# UMA IMPLEMENTAÇÃO DIDÁTICA DO ALGORÍTIMO KYBER

Rafael Hass - 103852

# AGENDA

Quem sou eu

Introdução

Implementação PKE

Implementação KEM

Resumo matemático

# QUEM SOU EU

- Formado em Licenciatura em Matemática em 2014
- Trabalhando com segurança da informação desde 2016
- Atualmente diretor de SOC e Threat Inteligence no banco BTG Pactual.

# INTRODUÇÃO

Algorítimo Kyber

# LINHA DO TEMPO

2017

O algoritmo CRYSTAL-Kyber é proposto como parte da iniciativa de submissão pós-quântica do NIST (National Institute of Standards and Technology) para algoritmos criptográficos.

2020

O NIST seleciona CRYSTAL-Kyber como um dos finalistas para o terceiro round da competição PQC.

2021

A terceira rodada da competição PQC está em andamento, com testes e avaliações mais aprofundados dos algoritmos finalistas, incluindo CRYSTAL-Kyber.

O NIST realiza um workshop público para discutir os algoritmos finalistas e obter feedback da comunidade criptográfica.

2022

O NIST anuncia os algoritmos pós-quânticos selecionados como padrões, incluindo o anúncio de CRYSTAL-Kyber como um dos algoritmos escolhidos.

# ALGUMAS CARACTERÍSTICAS DO KYBER

| Resistência algoritmos pós quânticos | Baseado em reticulados | MLWE | SVP |
|---|---|---|---|
| Resistência a algorítimos pós quânticos conhecidos | Grade multidimensional de pontos, onde a segurança é baseada na dificuldade de resolver problemas matemáticos complexos relacionados a essa estrutura. | LWE trabalha com vetores de inteiros<br>Ring LWE trabalha com polinômios<br>MLWE trabalha com vetores de polinômios inteiros módulo q | (Shortest Vector Problem), que envolve encontrar o vetor mais curto em um reticulado. |

# ALGUMAS CARACTERÍSTICAS DO KYBER

## Kyber PKE - IND-CPA

O IND-CPA (Indistinguishability under Chosen-Plaintext Attack) é um modelo de ataque onde um adversário tem a capacidade de escolher textos-claros e criptografá-los usando o esquema criptográfico que está sendo avaliado. Em seguida, o adversário recebe as cifras correspondentes e pode realizar operações e análises com base nessas cifras.

## Kyber KEM - IND-CCA2

O IND-CCA2 (Indistinguishability under Chosen-Ciphertext Attack) é um modelo de ataque mais forte em comparação ao IND-CPA. Nesse modelo, um adversário tem a capacidade de escolher pares de texto-claro e texto-cifrado, e também pode realizar operações e análises com base nessas informações.

Gerar chaves

s = Matriz de dimensões kx1 de polinômios de tamanho n

A = Matriz de dimensões kxk de polinômios de tamanho n

e = Matriz de dimensões kx1 de polinômios de tamanho n

Retornamos PK = (t= A*s + e,A) SK = s

Encriptar

m = mensagem em binário
r = Matriz de dimensões kx1 de polinômios de tamanho n
e1 = Matriz de dimensões kx1 de polinômios de tamanho n
e2 = Matriz de dimensões 1x1 de polinômios de tamanho n
Retornamos $u = A^T * r + e1$ e $v = t^T * r + e2 + m_u$

Decriptar

$$m^* = v - s^T * u = m + e2 + e^t * r + s^T * e1$$

E para achar o m, basta verificar para cada item do $m^*$, se está mais próximo de q/2 ou de (0 ou q)

# LINHA DO TEMPO

# IMPLEMENTAÇÃO DIDÁTICA

Algoritmo Kyber PKE

# O QUE É

Vou apresentar uma implementação simplificada do algoritmo CRYSTALS-Kyber, para exemplificar de forma didática alguns dos passos na execução, mantendo a lógica da implementação original, mas ignorando algumas etapas como compressão e descompressão, dado que não são relevantes para o entendimento do Kyber.

**Essa não é uma implementação que deve ser usada em qualquer aplicação do mundo real.**

# FUNÇÕES AUXILIARES USADAS

```python
def createRandomMatrixOfArraysofIntLimited(dimX: int, dimY: int, dimArrays: int, minValue: int, maxValue: int):
    matrix = np.empty((dimX, dimY), dtype=np.ndarray) # Create the matrix
    # Generate random values and assign them to each array
    for i in range(dimX):
        for j in range(dimY):
            random_values = np.random.randint(low=minValue, high=maxValue, size=dimArrays)
            matrix[i, j] = random_values
    return matrix
```

Gerar matriz de vetores com números aleatórios

```python
def setPolynomialRing(A: np.ndarray):
    rows = A.shape[0]
    columns = A.shape[1]
    matrix = np.zeros((rows, columns), dtype=np.ndarray)

    for i in range(rows):
        for j in range(columns):

            div, d = np.polynomial.polynomial.polydiv(A[i][j], cofdiv)
            matrix[i][j] = d % q

    return matrix
```

Fazer os polinômios módulo $x^{256} + 1$ e módulo q

# FUNÇÕES AUXILIARES USADAS

```python
def sum_arrays(A: np.ndarray,B: np.ndarray):

    if A.shape != B.shape:
        raise Exception("Dimensions incompatible:" + str( A.shape)+'x'+str(B.shape))
    rows = A.shape[0]
    columns = A.shape[1]
    matrix = np.zeros((rows, columns), dtype=np.ndarray)
    for i in range(rows):
        for j in range(columns):
            poly_sum = np.polynomial.polynomial.polyadd(A[i][j], B[i][j])
            matrix[i][j] = poly_sum

    return(setPolynomialRing(matrix))
```

Fazer a soma de matriz de polinômios

```python
def mult_matrix_polinomyals(A: np.ndarray,B: np.ndarray):

    rows = A.shape[0]
    columns = B.shape[1]
    matrix = np.zeros((rows, columns), dtype=np.ndarray)
    if A.shape[0] % B.shape[1] != 0:
        raise Exception("Dimensions incompatible:" + str( A.shape[0])+'x'+str(B.shape[1]))

    # Perform matrix multiplication
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            for k in range(B.shape[0]):
                resultMult = np.polynomial.polynomial.polymul(A[i][k], B[k][j]) # Line A x Column B
                matrix[i, j] = np.polynomial.polynomial.polyadd(matrix[i,j],resultMult)

    return(setPolynomialRing(matrix))
```

Fazer a multiplicação de matriz de polinômios

# PARÂMETROS A SEREM UTILIZADOS

Table 1: Parameter sets for Kyber

| | $n$ | $k$ | $q$ | $\eta_1$ | $\eta_2$ | $(d_u, d_v)$ | $\delta$ |
|---|---|---|---|---|---|---|---|
| Kyber512 | 256 | 2 | 3329 | 3 | 2 | (10, 4) | $2^{-139}$ |
| Kyber768 | 256 | 3 | 3329 | 2 | 2 | (10, 4) | $2^{-164}$ |
| Kyber1024 | 256 | 4 | 3329 | 2 | 2 | (11, 5) | $2^{-174}$ |

```
import numpy as np
import math

n = 256 # 256
q = 3329 # 3329
k = 2
n1 = 3
n2 = 2
cofdiv = np.zeros(n+1, dtype=int)
cofdiv[0] = 1
cofdiv[-1] = 1
```

- $n$ representa o tamanho em bits, da mensagem a ser encapsulada.
- $K$ representa o número de polinômios por vetor.
- $q$ representa um primo pequeno que satisfaz n|(q-1). É utilizado também para fazer o módulo das multiplicações de polinônios.
- $n_1, n_2, d_u, d_v$ representa um balanço entre segurança, tamanho do texto cifrado e chance de erro. Em nossa implementação, não usaremos $d_u, d_v$ pois são utilizados na compressão e descompressão das chaves e textos cifrados.
- $\delta$ representa a chance de erro.

# FUNÇÃO DE GERAR CHAVES

```python
def generateKey():

    s = createRandomMatrixOfArraysofIntLimited(k,1,n,-n1,n1+1)
    A = createRandomMatrixOfArraysofIntLimited(k,k,n,-q,q)
    e = createRandomMatrixOfArraysofIntLimited(k,1,n,-n1,n1+1)

    multA_S= mult_matrix_polinomyals(A,s)
    t = sum_arrays(multA_S, e)
    return ((A,t),s)
```

**Algorithm 4** KYBER.CPAPKE.KeyGen(): key generation

**Output:** Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
**Output:** Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$

$d \leftarrow \mathcal{B}^{32}$
$(\rho, \sigma) := G(d)$
$N := 0$
**for** $i$ from $0$ to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k \times k}$ in NTT domain
  **for** $j$ from $0$ to $k-1$ **do**
    $\hat{\mathbf{A}}[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, j, i))$
  **end for**
**end for**
**for** $i$ from $0$ to $k-1$ **do** ▷ Sample $\mathbf{s} \in R_q^k$ from $B_{\eta_1}$
  $\mathbf{s}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
  $N := N + 1$
**end for**
**for** $i$ from $0$ to $k-1$ **do** ▷ Sample $\mathbf{e} \in R_q^k$ from $B_{\eta_1}$
  $\mathbf{e}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(\sigma, N))$
  $N := N + 1$
**end for**
$\hat{\mathbf{s}} := \mathsf{NTT}(\mathbf{s})$
$\hat{\mathbf{e}} := \mathsf{NTT}(\mathbf{e})$
$\hat{\mathbf{t}} := \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$
$pk := (\mathsf{Encode}_{12}(\hat{\mathbf{t}} \bmod^{+} q) \| \rho)$ ▷ $pk := \mathbf{As} + \mathbf{e}$
$sk := \mathsf{Encode}_{12}(\hat{\mathbf{s}} \bmod^{+} q)$ ▷ $sk := \mathbf{s}$
**return** $(pk, sk)$

# FUNÇÃO DE ENCRIPTAR

```python
def encription(arrayMessageB, SK):

    A = SK[0]
    t = SK[1]
    r = createRandomMatrixOfArraysofIntLimited(k,1,n,-n1,n1+1)
    e1 = createRandomMatrixOfArraysofIntLimited(k,1,n,-n2,n2+1)
    e2 = createRandomMatrixOfArraysofIntLimited(1,1,n,-n2,n2+1)
    m = math.ceil(q/2)*arrayMessageB

    #u = A.T + e1
    mult_At_r = mult_matrix_polinomyals(A.T,r)
    u = sum_arrays(mult_At_r, e1)

    #v = t_t*r + e2 + m

    mult_tt_r = mult_matrix_polinomyals(t.T,r)

    sumE2M = sum_arrays(e2, m)
    v = sum_arrays(mult_tt_r,sumE2M)

    return (u,v)
```

**Algorithm 5** KYBER.CPAPKE.Enc$(pk, m, r)$: encryption

**Input:** Public key $pk \in \mathcal{B}^{12\cdot k\cdot n/8+32}$
**Input:** Message $m \in \mathcal{B}^{32}$
**Input:** Random coins $r \in \mathcal{B}^{32}$
**Output:** Ciphertext $c \in \mathcal{B}^{d_u\cdot k\cdot n/8+d_v\cdot n/8}$

$N := 0$
$\hat{\mathbf{t}} := \mathsf{Decode}_{12}(pk)$
$\rho := pk + 12 \cdot k \cdot n/8$
**for** $i$ from 0 to $k-1$ **do** ▷ Generate matrix $\hat{\mathbf{A}} \in R_q^{k\times k}$ in NTT domain
  **for** $j$ from 0 to $k-1$ **do**
    $\hat{\mathbf{A}}^T[i][j] := \mathsf{Parse}(\mathsf{XOF}(\rho, i, j))$
  **end for**
**end for**
**for** $i$ from 0 to $k-1$ **do** ▷ Sample $\mathbf{r} \in R_q^k$ from $B_{\eta_1}$
  $\mathbf{r}[i] := \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, N))$
  $N := N + 1$
**end for**
**for** $i$ from 0 to $k-1$ **do** ▷ Sample $\mathbf{e}_1 \in R_q^k$ from $B_{\eta_2}$
  $\mathbf{e}_1[i] := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
  $N := N + 1$
**end for**
$e_2 := \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$ ▷ Sample $e_2 \in R_q$ from $B_{\eta_2}$
$\hat{\mathbf{r}} := \mathsf{NTT}(\mathbf{r})$
$\mathbf{u} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$ ▷ $\mathbf{u} := \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$
$v := \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$ ▷ $v := \mathbf{t}^T\mathbf{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
$c_1 := \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\mathbf{u}, d_u))$
$c_2 := \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
**return** $c = (c_1 \| c_2)$ ▷ $c := (\mathsf{Compress}_q(\mathbf{u}, d_u), \mathsf{Compress}_q(v, d_v))$

# FUNÇÃO DE DECRIPTAR

```python
def decription(s, cm):
    u = cm[0]
    v = cm[1]

    #mn = v - s_t*u
    mult_st_u = mult_matrix_polinomyals(s.T,u)

    mn = sum_arrays(v, -mult_st_u)

    # print('mn: ' + str(mn))

    resultado = []
    for item in mn[0][0]:
        if abs(item - math.ceil(q/2)) < math.ceil(q/4):
            resultado.append(1)
        else:
            resultado.append(0)

    while len(resultado) < n:
        resultado.append(0)

    return [resultado]
```

**Algorithm 6** KYBER.CPAPKE.Dec($sk, c$): decryption

**Input:** Secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$
**Input:** Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
**Output:** Message $m \in \mathcal{B}^{32}$
1: $\mathbf{u} := \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
2: $v := \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n/8), d_v)$
3: $\hat{\mathbf{s}} := \mathsf{Decode}_{12}(sk)$
4: $m := \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^T \circ \mathsf{NTT}(\mathbf{u})), 1))$ $\triangleright$ $m := \mathsf{Compress}_q(v - \mathbf{s}^T\mathbf{u}, 1))$
5: **return** $m$

# IMPLEMENTAÇÃO DIDÁTICA

Algoritmo Kyber KEM

# FUNÇÃO DE GERAR CHAVES

```python
def KEM_KeyGen():
    return generateKey()
```

**Algorithm 7** KYBER.CCAKEM.KeyGen()

**Output:** Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
**Output:** Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8+96}$
1: $z \leftarrow \mathcal{B}^{32}$
2: $(pk, sk') := \text{KYBER.CPAPKE.KeyGen}()$
3: $sk := (sk' \| pk \| \mathrm{H}(pk) \| z)$
4: **return** $(pk, sk)$

# FUNÇÃO DE ENCRIPTAR

```python
def KEM_Encrypt(pk):

    m = createRandomMatrixOfArraysofIntLimited(1,1,n,0,2)
    ct = encription(m, pk)
    return ct,m
```

**Algorithm 8** $\text{Kyber.CCAKEM.Enc}(pk)$

**Input:** Public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8+32}$
**Output:** Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
**Output:** Shared key $K \in \mathcal{B}^*$

$m \leftarrow \mathcal{B}^{32}$
$m \leftarrow \mathrm{H}(m)$ ▷ Do not send output of system RNG
$(\bar{K}, r) := \mathrm{G}(m \| \mathrm{H}(pk))$
$c := \text{Kyber.CPAPKE.Enc}(pk, m, r)$
$K := \mathsf{KDF}(\bar{K} \| \mathrm{H}(c))$
**return** $(c, K)$

# FUNÇÃO DE DECRIPTAR

```python
def KEM_Decrypt(sk, ct):
    return decription(sk, ct)
```

**Algorithm 9** KYBER.CCAKEM.Dec$(c, sk)$

**Input:** Ciphertext $c \in \mathcal{B}^{d_u \cdot k \cdot n/8 + d_v \cdot n/8}$
**Input:** Secret key $sk \in \mathcal{B}^{24 \cdot k \cdot n/8 + 96}$
**Output:** Shared key $K \in \mathcal{B}^*$

$pk := sk + 12 \cdot k \cdot n/8$
$h := sk + 24 \cdot k \cdot n/8 + 32 \in \mathcal{B}^{32}$
$z := sk + 24 \cdot k \cdot n/8 + 64$
$m' :=$ KYBER.CPAPKE.Dec$(\mathbf{s}, (\mathbf{u}, v))$
$(\bar{K}', r') := \mathrm{G}(m' \| h)$
$c' :=$ KYBER.CPAPKE.Enc$(pk, m', r')$
**if** $c = c'$ **then**
    **return** $K :=$ KDF$(\bar{K}' \| \mathrm{H}(c))$
**else**
    **return** $K :=$ KDF$(z \| \mathrm{H}(c))$
**end if**
**return** $K$

# OBRIGADO

Rafael Hass

rafaelhass.92@gmail.com