UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

ESTRUTURAS CRIPTOGRÁFICAS

Trabalho 4

Grupo 10

Autores:

Joel Gama (A82202)

Tiago Pinheiro (A82491)

23 de Junho de 2020

Conteúdo

Introdução

O presente relatório surge no âmbito da Unidade Curricular de Estruturas Criptográficas integrada no perfil de Criptografia e Segurança da Informação.

Este trabalho é o último trabalho prático desta UC. O trabalho vai ter 2 implementações de duas categorias diferentes de esquemas. Cada tipo de esquema foi escolhido pelos elementos do grupo dentro de duas opções fornecidas no enunciado.

O primeiro esquema escolhido é o *Dilithium*, um esquema de assinatura digital.

O segundo é o *SPHINCS+* um esquema de assinatura baseado em *hash*.

Dilithium

Para este trabalho foi utilizada como base a implementação presente na Figura 1 do artigo *CRYSTALS-Dilithium: A Lattice-Based DigitalSignature Scheme* [1].

2.1 *Parameters*

Os parâmetros usados no trabalho foram definidos segundo o artigo [1]. No artigo existem 4 *sets* de parâmetros, de acordo com o nível de segurança *NIST*. O grupo escolheu usar os parâmetros referentes ao nível *weak* para ter uma compilação mais rápida.

Para além dos parâmetros também são gerados dois anéis polinomiais, **R** e **Rq**.

```
n = 256
q = 8380417
d = 14
peso = 60
y1 = (q-1)/16
y2 = y1/2
k = 3
l = 2
eta = 7
beta = 375
omega = 64

Zx.<x>  = ZZ[]
Gq.<z>  = PolynomialRing(GF(q))

R.<x>  = Zx.quotient(x^n+1)
Rq.<z> = Gq.quotient(z^n+1)
```

2.2 *Key Generation*

O primeiro passo é gerar uma matriz **A** com tamanho **k*l** (3*2) sendo todos os elementos pertences a **Rq**. Para isto, é criado um vetor com tamanho `k*l` e preenchido com elementos aleatórios de *Rq*. Depois basta utilizar a função `matrix` para criar e preencher a matriz.

Depois são criados os vetores **s1** e **s2** com tamanhos **l** e **k**, respetivamente, através da função `sam` - explicada mais à frente.

De seguida é calculado o **t**, que vai ser a segunda parte da chave pública. O **t** é o produto de `A` com `s1` mais `s2`.

No final, são definidas as chaves pública e privada. A chave privada é composta por **s1** e **s2**, por outro lado, a chave pública é composta por **A** e **t**.

```
def KeyGen():
    fillA = []
    for i in range(k*l):
        fillA.append(Rq.random_element())

    A = matrix(Rq,k,l,fillA)

    s1 = sam(eta,l)
    s2 = sam(eta,k)

    t = A*s1 + s2

    pk = (A,t)
    sk = (s1,s2)
    return {'pk': pk,'sk': sk }
```

2.3 *Signature*

Na assinatura é utilizado parte do retorno da `KeyGen` (A, s1 e s2). E no inicio é gerado o vetor **y** (tamanho tem que ser menor que y1) e feita a computação de **A** vezes **y**. Depois criado o vetor **w**, que contém todos os "*HighBits*"de **Ay**.

Os ciclos *for* vão fazer a parte de XOR (`w || m`) com a mensagem e no final é feita a *Hash* dessa string, **c**. O **c** é passado para **Rq** e passa a chamar-se **cQ**. Para terminar a primeira fase, é calculado o **z** (y + cQ*s1).

Para este **z** ser aceite, e seja considerado seguro, é preciso que se verifiquem duas condições:

- normaI(z) >= (y1-beta)
- LowBits(Ay-cQ*s2) >= (y2-beta)

Caso não se verifique, é repetido o processo até se verificar.

```
def Sign(text,keys):
    A = keys['pk'][0]
```

```
    s1 = keys['sk'][0]
    s2 = keys['sk'][1]

    y = sam(y1-1, l)
    Ay = A*y

    w = HighBits(Ay)

    string = ''
    string = string + text[2:]

    for i in range(len(w)):
        for j in range(len(w[i])):
            k = bin(w[i][j])
            if w[i][j] >= 0:
                string = string + k[2:]
            if w[i][j] < 0:
                string = string + k[3:]

    c = Hash(string)
    cQ = Rq(c)

    z = y + cQ*s1

    while (int(normaI(z)[0])) >= (y1-beta) and
            (normaI(LowBits(Ay-cQ*s2))) >= (y2-beta):
        ##(omitido)

    return {'z': z, 'c': c}
```

2.4 *Verify*

Na parte de verificação é apenas gerado o **w** (HighBits(A*z-cQ*t)) e calculado novamente o *Hash* de **c** (`c = Hash(m || w)`). Se tudo bater certo, é retornado sucesso, caso contrário, uma mensagem de erro e retorna-se o valor `-1`.

```
def Verify(text,cripto,keys):
    A = keys['pk'][0]
    t = keys['pk'][1]

    z = cripto['z']
    c = cripto['c']

    cQ = Rq(c)

    w = HighBits(A*z-cQ*t)
```

```
    string = ''
    string = string + text[2:]

    for i in range(len(w)):
        for j in range(len(w[i])):
            k = bin(w[i][j])
            if w[i][j] >= 0:
                string = string + k[2:]
            if w[i][j] < 0:
                string = string + k[3:]

    hashC = Hash(string)

    if (int(normaI(z)[0])) < (y1-beta) or hashC != c:
        print 'Denied'
        return -1
    else:
        print 'All good'
        return 1
```

2.5 Funções auxiliares

2.5.1 *Sam*

Preenche um array com elementos pertencentes a **Rq** e no final produz uma matriz `size*1`, sendo o *size* um valor dado.

```
def sam (limit,size):
    lista = []
    for i in range(size):
        poly = []
        for j in range(n):
            poly.append(randint(1,limit))
        lista.append(Rq(poly))

    res = matrix(Rq,size,1,lista)
    return res
```

2.5.2 *HighBits* e *LowBits*

```
def HighBits(poly):
    lista = poly.list()
    for i in range(len(lista)):
```

```
            f = lista[i]
            F = f.list()
            for j in range(len(F)):
                F[j] = HBAux(int(F[j]))

            lista[i] = F

        return lista

    def HBAux(C):
        res = Decompose(C,2*y2)
        return res[0]
```

2.5.3 *Decompose*

```
    def Decompose(C,alfa):
        r = mod(C,int(q))
        r0 = int(mod(r, int(alfa)))

        if (r-r0) == (q-1):
            r1 = 0
            r0 = r0 - 1
        else:
            r1 = (r-r0)/(int(alfa))

        return (r1,r0)
```

2.5.4 Norma

```
    def normaI(v):
        for i in range(v.nrows()):
            norma = normaIAux(v[i],q)
            v[i] = norma
        return max(v)

    def normaIAux(poly,number):
        lista = poly.list()
        for i in range(len(lista)):
            f = lista[i]
            F = f.list()
            for j in range(len(F)):
                F[j] = abs(int(F[j]))
            lista[i]=F
```

```
    List = []
    for i in range(len(lista)):
        List.append(max(lista[i]))

    return max(List)
```

2.5.5 *Hash*

A função *Hash* transforma uma certa *string* num vetor com **n** entradas, neste caso, são 256 elementos com, no máximo, **peso** (60) elementos igual a 1 ou -1 e os restantes igual a zero.

```
def Hash(value):
    H = []
    contador = 0
    contador_ = 0
    for i in range(0,n,2):
        u=value[i]+value[i+1]
        contador = contador + 1
        if u == '11':
            H.append(0)
        if u == '01':
            H.append(1)
            contador_ = contador_ + 1
        if u == '00':
            pass
        if u == '10':
            H.append(-1)
            contador_ = contador_ + 1
        if contador_ >= peso:
            break

    for i in range(n-contador):
        H.append(0)

    return H
```

2.6 Teste

```
def run():
    keys = KeyGen()
    text = bin(1024)
    cripto = Sign(text,keys)
    res = Verify(text,cripto,keys)
    return res
```

SPHINCS+

Para a implementação do algoritmo *SPHINCS+* foi utilizado como base o artigo *The SPHINCS+ Signature Framework*. [2] Nele estão presentes os vários componentes a implementar e os parâmetros necessários para cada um. Para além disso, possui uma explicação teórica de como o algoritmo funciona.

3.1 Classe *Connect*

A classe *Connect* é utilizada como o *address (ADRS)* referido no documento [2].

```
class Connect:
    # Types
    WOTSHASH = 0
    WOTSPK = 1
    TREE = 2
    FORSTREE = 3
    FORSROOTS = 4

        (...)
```

3.2 Parâmetros

```
_randomize = True

N = 16 # Security parameter
W = 4  # Winternitz parameter (usualy 4, 16 or 256)
H = 64 # HyperTree height
D = 8  # HyperTree layers
K = 10 # Fors trees
A = 15 # Fors tree height
T = 2 ** A # Fors tree leaves

L1 = math.ceil(8 * N /math.log(W,2)) # len 1
L2 = math.floor(math.log(L1 * (W - 1),2) /
                math.log(W,2)) + 1 # len 2
L0 = L1 + L2 # len 0
HPrime = H // D
```

3.3 *Tweakable Hash Funcions*

Nesta secção é feita são definidas as funções de *hash* a usar. Segundo o documento este tipo de funções de *hash* são recomendados para o contexto definido e tomam o lugar de *nonces* em termos de segurança. Para além disso, permitem serem chamadas em cada par de chaves do *SPHINCS+* e em cada posição da estrutura da árvore virtual do *SPHINCS+* de forma independente entre si.

```
# Hash (Sha256 based)
def hash(seed, conn: Connect, value, digestSize):
    h = hashlib.sha256()

    h.update(seed)
    h.update(conn.toBin())
    h.update(value)

    hashed = h.digest()[:digestSize]

    return hashed

# Pseudorandom Function
def prf(skSeed, conn, digestSize):
    random.seed(int.from_bytes(skSeed + conn.toBin(),"big"))
    return random.randint(0, 256 ** digestSize - 1)
                        .to_bytes(digestSize, byteorder='big')

# Message Hash
def hashMsg(r, pkSeed, pkRoot, value, digestSize):
    h = hashlib.sha256()

    h.update(r)
    h.update(pkSeed)
    h.update(pkRoot)
    h.update(value)

    hashed = h.digest()[:digestSize]

    i = 0
    while len(hashed) < digestSize:
        i += 1
        h = hashlib.sha256()

        h.update(r)
        h.update(pkSeed)
        h.update(pkRoot)
        h.update(value)
        h.update(bytes([i]))
```

```
        hashed += h.digest()[:digestSize - len(hashed)]

    return hashed

# Message Pseudorandom Function
def prfMsg(skSeed, opt, m, digestSize):
    random.seed(int.from_bytes(skSeed + opt +
                hashMsg(b'0', b'0', b'0', m, digestSize * 2),"big"))
    return random.randint(0, 256 ** digestSize - 1)
                         .to_bytes(digestSize, byteorder='big')
```

3.4 *Wots+*

É um esquema de assinatura *One-Time signature*, isto é, uma chave privada deve ser utilizada para assinar apenas uma mensagem.

O método *chain* é responsável por iterar o valor de *x s steps*. São utilizados o endereço *Connect* e a *public seed* para a operação.

```
# Chaining
def chain(x, i, s, pkSeed, conn: Connect):
    if s == 0:
        return bytes(x)

    if (i + s) > (W - 1):
        return -1
    tmp = chain(x, i, s - 1, pkSeed, conn)

    conn.setHashAdrs(i + s - 1)
    tmp = hash(pkSeed, conn, tmp, N)

    return tmp

# Public Key
def wotsGenPk(skSeed, pkSeed, conn: Connect):

    wotsPkConn = conn.copy()
    tmp = bytes()

    for i in range(0, L0):
        # Secret Key
        conn.setChainAdrs(i)
        conn.setHashAdrs(0)
        sk = prf(skSeed, conn.copy(), N)

        tmp += bytes(chain(sk, 0, W - 1, pkSeed, conn.copy()))

    wotsPkConn.setType(Connect.WOTSPK)
    wotsPkConn.setPairAdrs(conn.getPairAdrs())
```

```
    pk = hash(pkSeed, wotsPkConn, tmp, N)

    return pk

# Signature
def wotsSign(m, skSeed, pkSeed, conn):

    csum = 0

    msg = baseW(m, W, L1) # base W

    # checksum calculation
    for i in range(0, L1):
        csum += W -1 - msg[i]

    # checksum to base W
    padding = (L2 * math.floor(math.log(W,2))) % 8
               if (L2 * math.floor(math.log(W,2))) % 8 != 0 else 8

    csum = csum << (8 - padding)
    csumb = int(csum).to_bytes(math.ceil((L2 *
                               math.floor(math.log(W,2))) / 8),
                               byteorder='big')
    csumw = baseW(csumb, W, L2)
    msg += csumw

    sig = []
    for i in range(0, L0):
        conn.setChainAdrs(i)
        conn.setHashAdrs(0)
        sk = prf(skSeed, conn.copy(), N)
        sig += [chain(sk,0, msg[i], pkSeed, conn.copy())]

    return sig

# Pk from Signature
def wotsPkSig(sig, m, pkSeed, conn: Connect):
    csum = 0
    wotsPkConn = conn.copy()

    msg = baseW(m, W, L1)

    for i in range(0, L1):
        csum += W - 1 - msg[i]

    padding = (L2 * math.floor(math.log(W,2))) % 8
               if (L2 * math.floor(math.log(W,2))) % 8 != 0 else 8
```

```
    csum = csum << (8 - padding)
    csumb = int(csum).to_bytes(math.ceil((L2 *
                               math.floor(math.log(W,2))) / 8),
                               byteorder='big')
    csumw = baseW(csumb, W, L2)
    msg += csumw

    tmp = bytes()
    for i in range(0, L0):
        conn.setChainAdrs(i)
        tmp += chain(sig[i], msg[i], W - 1 - msg[i],
                     pkSeed, conn.copy())

    wotsPkConn.setType(Connect.WOTSPK)
    wotsPkConn.setPairAdrs(conn.getPairAdrs())

    pkSig = hash(pkSeed, wotsPkConn, tmp, N)

    return pkSig
```

3.5 *Hypertree (XMSS)*

Para a implementação do *Hypertree* são primeiro definidos métodos do tipo *single tree*, neste caso *XMSS*. Desta forma conseguimos combinar o *WOTS+* com as árvores binárias de *hash* o que resulta numa versão com *input* fixo. O *XMSS (eXtended Markle Signature Scheme)* permite assinar um número fixo de mensagens baseado no esquema de assinaturas *Merkle*. Cada nodo da árvore é um valor de n-bytes representado pela *teakable hash* da concatenação dos nodos dos filhos. Neste caso de implementação do *SPHINCS+* apenas é utilizada a *secret seed* para gerar todas as *secrets keys* do *WOTS+*.

```
# Return the root node (n-byte)
def treeHash(skSeed, s, z, pkSeed, conn: Connect):
    if s % (1 << z) != 0:
        return -1

    stack = []

    for i in range(0, 2 ** z):
        conn.setType(Connect.WOTSHASH)
        conn.setPairAdrs(s + i)
        node = wotsGenPk(skSeed, pkSeed, conn.copy())

        conn.setType(Connect.TREE)
        conn.setTreeHeight(1)
        conn.setTreeIndex(s+i)
```

```
        if len(stack) > 0:
            while stack[len(stack)-1]['height'] ==
                  conn.getTreeHeight():

                conn.setTreeIndex((conn.getTreeIndex() - 1) // 2)
                node = hash(pkSeed, conn.copy(),
                            stack.pop()['node'] + node, N)
                conn.setTreeHeight(conn.getTreeHeight() + 1)

                if len (stack) <= 0:
                    break

        stack.append({'node': node, 'height': conn.getTreeHeight()})

    return stack.pop()['node']

# Public Key
def xmssPkGen(skSeed, publicKey, conn: Connect):
    pk = treeHash(skSeed, 0, HPrime, publicKey, conn.copy())

    return pk

# Sign
def xmssSign(m, skSeed, idx, pkSeed, conn):
    auth = []

    # Authentication path
    for j in range(0, HPrime):
        ki = math.floor(idx // 2 ** j)
        if ki % 2 == 1: # XOR idx / 2**j with 1
            ki -= 1
        else:
            ki += 1

        auth += [treeHash(skSeed, ki * 2 **  j, j,
                 pkSeed, conn.copy())]

    conn.setType(Connect.WOTSHASH)
    conn.setPairAdrs(idx)

    sig = wotsSign(m, skSeed, pkSeed, conn.copy())
    sigXmss = sig + auth


    return sigXmss

# Pk from Signature
def xmssPkSig(idx, sigXmss, m, pkSeed, conn):
```

```
    # WOTS+ pk from WOTS+ signature
    conn.setType(Connect.WOTSHASH)
    conn.setPairAdrs(idx)
    sig = sigWotsSigXmss(sigXmss)
    auth = authSigXmss(sigXmss)

    nodo0 = wotsPkSig(sig, m, pkSeed, conn.copy())
    nodo1 = 0

    # Root from WOTS+ pk and authentication
    conn.setType(Connect.TREE)
    conn.setTreeIndex(idx)
    for i in range(0, HPrime):
        conn.setTreeHeight(i + 1)

        if math.floor(idx / 2 ** i) % 2 == 0:
            conn.setTreeIndex(conn.getTreeIndex() // 2)
            nodo1 = hash(pkSeed, conn.copy(), nodo0 + auth[i], N)
        else:
            conn.setTreeIndex((conn.getTreeIndex() - 1) // 2)
            nodo1 = hash(pkSeed, conn.copy(), auth[i] + nodo0, N)

        nodo0 = nodo1

    return nodo0
```

3.6 *Hypertree HT*

Hypertree HT é uma variante do *XMSS*. É implementada uma *tree of tree*s (árvore de árvores). Basicamente é uma árvore com várias camadas de árvores. Sendo utilizadas as camadas superiores e intermédias para assinar as chaves públicas.

```
# Public Key (unic superior layer XMSS tree root)
def htGenPk(skSeed, pkSeed):
    conn = Connect()
    conn.setLayerAdrs(D - 1)
    conn.setTreeAdrs(0)
    root = xmssPkGen(skSeed, pkSeed, conn.copy())

    return root

# Signature
def htSing(m, skSeed, pkSeed, indexTree, indexLeaf):
    # inicialization
    conn = Connect()
    conn.setLayerAdrs(0)
    conn.setTreeAdrs(indexTree)
```

```
    # sign
    sigTmp = xmssSign(m, skSeed, indexLeaf, pkSeed, conn.copy())
    sigHt = sigTmp
    root = xmssPkSig(indexLeaf, sigTmp, m, pkSeed, conn.copy())

    for j in range(1, D):
        indexLeaf = indexTree % 2 ** HPrime
        indexTree = indexTree >> HPrime

        conn.setLayerAdrs(j)
        conn.setTreeAdrs(indexTree)

        sigTmp = xmssSign(root, skSeed, indexLeaf,
                          pkSeed, conn.copy())
        sigHt = sigHt + sigTmp

        if j < D - 1:
            root = xmssPkSig(indexLeaf, sigTmp, root,
                             pkSeed, conn.copy())

    return sigHt

# Verify Signature
def htVerify(m, sigHt, pkSeed, indexTree, indexLeaf, publicKeyHt):

    conn = Connect()

    # verification
    sigsXmss = sigWotsSigXmss(sigHt)
    sigTmp = sigsXmss[0]

    conn.setLayerAdrs(0)
    conn.setTreeAdrs(indexTree)
    node = xmssPkSig(indexLeaf, sigTmp, m, pkSeed, conn)

    for j in range(1, D):
        indexLeaf = indexTree % 2 ** HPrime
        indexTree = indexTree >> HPrime

        sigTmp = sigsXmss[j]

        conn.setLayerAdrs(j)
        conn.setTreeAdrs(indexTree)

        node = xmssPkSig(indexLeaf, sigTmp, node, pkSeed, conn)

    if node == publicKeyHt:
```

```
        return True
    else:
        return False
```

3.7 *FORS*

A *Hypertree HT* é utilizada para assinar as chaves públicas das instâncias *FORS* e não as mensagens. São as instâncias *FORS* que são responsáveis por assinar as mensagens. Uma mensagem privada é formada por *kt* strings de n-bytes aleatórias, agrupadas em *k sets*.

A verificação das chaves publicas é feita implicitamente, pois existe apenas um método para calcular uma chave publica candidata.

```
# Private Key
def forsGenSk(skSeed, conn: Connect, idx):
    conn.setTreeHeight(0)
    conn.setTreeIndex(idx)
    sk = prf(skSeed, conn.copy(), N)

    return sk

# TreeHash (change the leaf calculation and address management)
def forsTreehash(skSeed, s, z, pkSeed, conn):
    if s % (1 << z) != 0:
        return -1

    stack = []

    for i in range(0, 2 ** z):
        conn.setTreeHeight(0)
        conn.setTreeIndex(s+i)
        sk = prf(skSeed, conn.copy(), N)
        node = hash(pkSeed, conn.copy(), sk, N)

        conn.setTreeHeight(1)
        conn.setTreeIndex(s+i)

        if len(stack) > 0:
            while stack[len(stack)-1]['height'] ==
                  conn.getTreeHeight():

                conn.setTreeIndex((conn.getTreeIndex() - 1) // 2)
                node = hash(pkSeed, conn.copy(),
                            stack.pop()['node'] + node, N)
                conn.setTreeHeight(conn.getTreeHeight() + 1)

                if len (stack) <= 0:
                    break
```

```
        stack.append({'node': node, 'height': conn.getTreeHeight()})

    return stack.pop()['node']

# Public Key
def forsGenPk(skSeed, pkSeed, conn: Connect):

    forsConn = conn.copy()

    root = bytes()
    for i in range(0, K):
        root += forsTreehash(skSeed, i * T, A, pkSeed, conn)

    forsConn.setType(Connect.FORSROOTS)
    forsConn.setPairAdrs(conn.getPairAdrs())
    pk = hash(pkSeed, forsConn, root, N)

    return pk

# Signature
def forsSign(m, skSeed, pkSeed, conn):
    mInt = int.from_bytes(m, 'big')
    sigFors = []

    # Signature elements calculation
    for i in range(0, K):

        # next index
        idx = (mInt >> (K -1 -i) * A) % T

        # private key element
        conn.setTreeHeight(0)
        conn.setTreeIndex(i * T + idx)
        sigFors += [prf(skSeed, conn.copy(), N)]

        auth = []

        # authentication path
        for j in range(0, A):
            s = math.floor(idx // 2 ** j)
            if s % 2 == 1:
                s -= 1
            else:
                s += 1

            auth += [forsTreehash(skSeed, i * T + s * 2 ** j, j,
                    pkSeed, conn.copy())]
```

```
        sigFors += auth

    return sigFors

# Pk from Signature
def forsPkSig(sigFors, m, pkSeed, conn:Connect):

    mInt = int.from_bytes(m, 'big')

    sigs = authsSigFors(sigFors)
    root = bytes()

    # roots calculation
    for i in range(0, K):

        #next index
        idx = (mInt >> (K -1 -i) * A) % T

        # leaf calculation
        sk = sigs[i][0]
        conn.setTreeHeight(0)
        conn.setTreeIndex(i * T + idx)
        nodo0 = hash(pkSeed, conn.copy(), sk, N)
        nodo1 = 0

        # root from authentication and leaf
        auth = sigs[i][1]
        conn.setTreeIndex(i * T + idx)

        for j in range(0, A):
            conn.setTreeHeight(j + 1)

            if math.floor(idx / 2 ** i) % 2 == 0:
                conn.setTreeIndex(conn.getTreeIndex() // 2)
                nodo1 = hash(pkSeed, conn.copy(), nodo0 + auth[j], N)
            else:
                conn.setTreeIndex((conn.getTreeIndex() - 1) // 2)
                nodo1 = hash(pkSeed, conn.copy(), auth[j] + nodo0, N)

            nodo0 = nodo1

        root += nodo1

    forsConn = conn.copy()
    forsConn.setType(Connect.FORSROOTS)
    forsConn.setPairAdrs(conn.getPairAdrs())
```

```
    pk = hash(pkSeed, forsConn, root, N)
    return pk
```

3.8 *SPHINCS+*

Na implementação do algoritmo *SPHINCS+* propriamente dito são utilizados todos os pontos anteriores para realizar a geração de chaves, assinatura e verificação.

```
(...)
# Key pair generation
def genKeyPair():

    sk, pk = shpKeygen()
    sk_0, pk_0 = bytes(), bytes()

    for i in sk:
        sk_0 += i

    for i in pk:
        pk_0 += i

    return sk_0, pk_0

# Key pair generation aux
def shpKeygen():

    skSeed = os.urandom(N)
    skPrf  = os.urandom(N)
    pkSeed = os.urandom(N)

    pkRoot = htGenPk(skSeed, pkSeed)

    return [skSeed, skPrf, pkSeed, pkRoot], [pkSeed, pkRoot]

# Signature
def sign(m, sk):
    skTab = []

    for i in range(0, 4):
        skTab.append(sk[(i * N):((i + 1) * N)])

    sigTab = shpSign(m, skTab)

    sig = sigTab[0]

    for i in sigTab[1]:
        sig += i
    for i in sigTab[2]:
```

```
        sig += i

    return sig

# Signature aux
def shpSign(m, secretKey):
    conn = Connect()

    skSeed = secretKey[0]
    skPrf  = secretKey[1]
    pkSeed = secretKey[2]
    pkRoot = secretKey[3]

    opt = bytes(N)
    if _randomize:
        opt = os.urandom(N)
    r = prfMsg(skPrf, opt, m, N)
    sig = [r]

    sizeMd = math.floor((K * A + 7) / 8)
    sizeIndexTree = math.floor((H - H // D + 7) / 8)
    sizeIndexLeaf = math.floor((H // D + 7) / 8)

    digest = hashMsg(r, pkSeed, pkRoot, m,
                     sizeMd + sizeIndexTree + sizeIndexLeaf)
    tmpMd = digest[:sizeMd]
    tmpIndexTree = digest[sizeMd:(sizeMd + sizeIndexTree)]
    tmpIndexLeaf = digest[(sizeMd + sizeIndexTree):len(digest)]

    mdInt = int.from_bytes(tmpMd, 'big') >> (len(tmpMd) * 8 - K * A)
    md = mdInt.to_bytes(math.ceil(K * A / 8), 'big')

    indexTree = int.from_bytes(tmpIndexTree, 'big') >>
                    (len(tmpIndexTree) * 8 - (H - H // D))
    indexLeaf = int.from_bytes(tmpIndexLeaf, 'big') >>
                    (len(tmpIndexLeaf) * 8 - (H // D))

    conn.setLayerAdrs(0)
    conn.setTreeAdrs(indexTree)
    conn.setType(Connect.FORSTREE)
    conn.setPairAdrs(indexLeaf)

    sigFors = forsSign(md, skSeed, pkSeed, conn.copy())
    sig += [sigFors]

    pkFors = forsPkSig(sigFors, md, pkSeed, conn.copy())

    conn.setType(Connect.TREE)
```

```
    sigHt = htSing(pkFors, skSeed, pkSeed, indexTree, indexLeaf)
    sig += [sigHt]

    return sig

# Verify
def verify(m, sig, pk):
    pkTab = []

    for i in range(0, 2):
        pkTab.append(pk[(i * N):((i + 1) * N)])

    sigTab = []

    sigTab += [sig[:N]]
    sigTab += [[]]
    for i in range(N, N + K * (A + 1) * N, N):
        sigTab[1].append(sig[i:(i + N)])

    sigTab += [[]]
    for i in range(N + K * (A + 1) * N,
                   N + K * (A + 1) * N + (H + D * L0) * N,
                   N):
        sigTab[2].append(sig[i:(i + N)])

    return shpVerify(m, sigTab, pkTab)

# Verify aux
def shpVerify(m, sig, publicKey):
    conn = Connect()

    r = sig[0]
    sigFors = sig[1]
    sigHt = sig[2]

    pkSeed = publicKey[0]
    pkRoot = publicKey[1]

    sizeMd = math.floor((K * A + 7) / 8)
    sizeIndexTree = math.floor((H - H // D + 7) / 8)
    sizeIndexLeaf = math.floor((H // D + 7) / 8)

    digest = hashMsg(r, pkSeed, pkRoot, m,
                     sizeMd + sizeIndexTree + sizeIndexLeaf)
    tmpMd = digest[:sizeMd]
    tmpIndexTree = digest[sizeMd:(sizeMd + sizeIndexTree)]
    tmpIndexLeaf = digest[(sizeMd + sizeIndexTree):len(digest)]
```

```
    mdInt = int.from_bytes(tmpMd, 'big') >>
                          (len(tmpMd) * 8 - K * A)
    md = mdInt.to_bytes(math.ceil(K * A / 8), 'big')

    indexTree = int.from_bytes(tmpIndexTree, 'big') >>
                    (len(tmpIndexTree) * 8 - (H - H // D))
    indexLeaf = int.from_bytes(tmpIndexLeaf, 'big') >>
                    (len(tmpIndexLeaf) * 8 - (H // D))

    conn.setLayerAdrs(0)
    conn.setTreeAdrs(indexTree)
    conn.setType(Connect.FORSTREE)
    conn.setPairAdrs(indexLeaf)

    pkFors = forsPkSig(sigFors, md, pkSeed, conn)

    conn.setType(Connect.TREE)

    return htVerify(pkFors, sigHt, pkSeed,
                    indexTree, indexTree, pkRoot)
```

3.9 Teste

Por fim, na parte final é feito o teste da implementação. Começa-se por gerar as chaves, depois é assinada uma mensagem predefinida e por fim é feita a verificação da assinatura.

```
sk, pk = genKeyPair()

m = b'Test message'
print(m)

signature = sign(m, sk)
print(signature)

result = verify(m, signature, pk)
print(result)
```

Conclusão

Este trabalho foi, sem dúvida, o mais difícil realizado até ao momento. O grupo sentiu imensas dificuldades com os tipos e cálculos necessários a efetuar.

Foi possível aplicar conceitos relacionados *Criptosistemas pós-quânticos*, *PKE* e *KEM*. Mais especificamente os esquemas *NTRU-Prime* e *New Hope*.

Em ambos os casos, *NTRU-Prime* e *New Hope*, apenas foi possível concluir com sucesso o exercício 1, ou seja, a implementação do *KEM*.

No *NTRU-Prime* ficou por fazer o exercício 2, algo que o grupo quer concluir ainda no decorre dos próximos dias, se for possível.

Na parte do *New Hope*, foi feito o exercício 2 mas como tem erros, não foi possível atingir a solução final.

Bibliografia

[1] CRYSTALS-Dilithium: A Lattice-Based DigitalSignature Scheme, https://eprint.iacr.org/2017/633.pdf.

[2] The SPHINCS+ Signature Framework, https://eprint.iacr.org/2019/1086.pdf