Prototype SPHINCS+

Enunciado do Problema

Neste trabalho pretende-se implementar em Sagemath de algumas dos candidatos a “standartização” ao concurso NIST Post-Quantum Cryptography na categoria de esquemas de assinatura digital. Ver também a diretoria com a documentação. Construa

- Um protótipo Sagemath do algoritmo Sphincs+.

Descrição do Problema

O problema consiste em implementar um protótipo do algoritmo **SPHINCS+.**

Abordagem

Através da documentação disponibilizada no site oficial do [SPHINCS+](), foi possível dividir o problema em várias partes:

1. Definição dos parâmetros
2. Funções matemáticas necessárias
3. Funções auxiliares necessárias
4. Definição da classe ADRS
5. Funções de hash
6. Funções de WOTS+
7. Funções de XMSS
8. Funções do Hypertree
9. Funções de FORS
10. Funções de SPHINCS+

Código

Segue-se o código desenvolvido para este protótipo do algoritmo **SPHINCS+**

Definição dos parâmetros

Foram definidos os parâmetros da variante **SPHINCS+-128s.**

In [1]:

```
RANDOMIZE = True

n = 16  # The security parameter in bytes
w = 16  # The Winternitz parameter
h = 63  # The height of the hypertree
d = 7  # The number of layers in the hypertree
k = 14  # The number of trees in FORS
a = 12  # The number of leaves of a FORS tree (lg t)
t = 2 ^ a  # The number of leaves of a FORS tree
```

Math Functions needed

In [2]:

```
def lg(x: int) -> float:
    return math.log(x, 2)
```

```
def ceil(x: float) -> int:
    return math.ceil(x)


def floor(x: float) -> int:
    return math.floor(x)
```

Compute the message digest lenght and the number of n-byte string elements in WOTS+ private key, public key and signature values

In [3]:

```
m_digest_len = floor((k * lg(t) + 7) / 8) + floor((h - h / d + 7) / 8) + floor((h / d +
7) / 8)

len_1 = ceil(8 * n / lg(w))
len_2 = floor(lg(len_1 * (w - 1)) / lg(w)) + 1
_len = len_1 + len_2
```

Function toByte

In [4]:

```
def toByte(x: int, y: int):
    """
    x, y non-negative integers
    x - number to be converted
    y - number of bytes in the output
    Z = toByte(x, y) is the y-byte string representation of x
    """

    return int(x).to_bytes(int(y), 'big')


x = 255
y = 2

result = toByte(x, y)
print(result)   # Output: b'\x00\xff'
```

```
b'\x00\xff'
```

Function base_w

In [5]:

```
from typing import List


def base_w(X: str, w: int, out_len: int) -> List[int]:
    """
    Input: len_X-byte string X, int w, output length out_len
    Output: out_len int array basew
    """
    _in = 0
    out = 0
    total = 0
    bits = 0
    basew = []

    for consumed in range(out_len):
        if bits == 0:
            total = X[_in]
            _in += 1
            bits += 8
        bits -= int(lg(w))
```

```
        basew.append((total >> bits) & (w - 1))
        out += 1

    return basew


X = b'\x12\x34'
w = 16
out_len = 4

result = base_w(X, w, out_len)
print(result)   # Output: [1, 2, 3, 4]
```

```
[1, 2, 3, 4]
```

ADRS

In [6]:

```
class ADRS:
    # Types
    TYPE_WOTS = 0
    TYPE_WOTSPK = 1
    TYPE_HASHTREE = 2
    TYPE_FORSTREE = 3
    TYPE_FORSPK = 4

    def __init__(self):
        self.layer_address = 0
        self.tree_address = 0
        self.type = 0

        # Differs from type to type
        self.word1 = 0
        self.word2 = 0
        self.word3 = 0

    def copy(self):
        adrs = ADRS()
        adrs.layer_address = self.layer_address
        adrs.tree_address = self.tree_address

        adrs.type = self.type
        adrs.word1 = self.word1
        adrs.word2 = self.word2
        adrs.word3 = self.word3
        return adrs

    def to_bytes(self):
        adrs = toByte(self.layer_address, 4)
        adrs += toByte(self.tree_address, 12)

        adrs += toByte(self.type, 4)
        adrs += toByte(self.word1, 4)
        adrs += toByte(self.word2, 4)
        adrs += toByte(self.word3, 4)

        return adrs

    def reset_words(self):
        self.word1 = 0
        self.word2 = 0
        self.word3 = 0

    def set_type(self, val):
        self.type = val

        self.word2 = 0
        self.word3 = 0
        self.word1 = 0
```

```python
    def set_layer_address(self, val):
        self.layer_address = val

    def set_tree_address(self, val):
        self.tree_address = val

    def set_key_pair_address(self, val):
        self.word1 = val

    def get_key_pair_address(self):
        return self.word1

    def set_chain_address(self, val):
        self.word2 = val

    def set_hash_address(self, val):
        self.word3 = val

    def set_tree_height(self, val):
        self.word2 = val

    def get_tree_height(self):
        return self.word2

    def set_tree_index(self, val):
        self.word3 = val

    def get_tree_index(self):
        return self.word3
```

F, H, T Functions (Tweakable Hash Functions)

For the simple variant, we instead define the tweakable hash functions as

```
math
F(PK.seed, ADRS, M1) = SHAKE256(PK.seed || ADRS|| M1, 8n),
```

```
math
H(PK.seed, ADRS, M1||M2) = SHAKE256(PK.seed || ADRS || M1 || M2, 8n),
```

```
math
T(PK.seed, ADRS, M) = SHAKE256(PK.seed || ADRS || M, 8n)
```

In [7]:

```python
def F(pk_seed: bytes, adrs: ADRS, m: bytes) -> bytes:
    """
    Input: pk_seed, adrs, m
    Output: m hash
    """
    sha256 = hashlib.sha256()

    sha256.update(pk_seed)
    sha256.update(adrs.to_bytes())
    sha256.update(m)

    pre_hashed = sha256.digest()
    hashed = pre_hashed[:m_digest_len]

    return hashed



def H(seed, adrs: ADRS, value, digest_size=n):
    """
    Input: seed, adrs, value
    Output: digest_size-byte hash
    """
```

```python
    m = hashlib.sha256()

    m.update(seed)
    m.update(adrs.to_bytes())
    m.update(value)

    pre_hashed = m.digest()
    hashed = pre_hashed[:digest_size]

    return hashed


def PRF(secret_seed, adrs):
    random.seed(int.from_bytes(secret_seed + adrs.to_bytes(), "big"))
    return int(random.randint(0, 256 ^ n)).to_bytes(n, byteorder='big')


def hash_msg(r, public_seed, public_root, value, digest_size=n):
    m = hashlib.sha256()

    m.update(str(r).encode('ASCII'))
    m.update(public_seed)
    m.update(public_root)
    m.update(value)

    pre_hashed = m.digest()
    hashed     = pre_hashed[:digest_size]
    i = 0
    while len(hashed) < digest_size:
        i += 1
        m = hashlib.sha256()

        m.update(str(r).encode('ASCII'))
        m.update(public_seed)
        m.update(public_root)
        m.update(value)
        m.update(bytes([i]))

        hashed += m.digest()[:digest_size - len(hashed)]

    return hashed


def PRF_msg(secret_seed, opt, m):
    random.seed(int.from_bytes(secret_seed + opt + hash_msg(b'0', b'0', b'0', m, n*2), "
big"))
    return int(random.randint(0, 256 ^ n)).to_bytes(n, byteorder='big')

def mgf1_sha256(seed, mask_len):
    output = b""
    counter = int(0)

    while len(output) < mask_len:
        c = int.to_bytes(counter, length=4, byteorder='big')
        data = seed + c
        hash_value = hashlib.sha256(data).digest()
        output += hash_value
        counter += 1

    return output[:mask_len]


def H_msg(R, PK_seed, PK_root, M, mask_len=m_digest_len):
    input_data = R + PK_seed + PK_root + M
    hash_input = hashlib.sha256(input_data).digest()
    mask = mgf1_sha256(hash_input, mask_len)
    return mask
```

WOTS+ (Winternitz One Time Signature Plus)

Foram definidas as seguintes funções para o esquema WOTS+:

Foram definidas as seguintes funções para o esquema WOTS+:

- `chain`: **função que itera a função F s vezes em uma string X**
- `wots_sk_gen`: **gera a chave privada WOTS+ a partir de uma seed secreta SK.seed (não é usada no algoritmo em si, apenas para ajudar a perceber o funcionamento dos algoritmos)**
- `wots_pk_gen`: **gera a chave pública WOTS+ a partir de uma chave privada sk**
- `wots_sign`: **gera uma assinatura WOTS+ a partir de uma mensagem M e uma chave privada sk**
- `wots_pk_from_sig`: **calcula a chave pública WOTS+ a partir de uma assinatura sig e uma mensagem M**

In [8]:

```
from typing import Optional


def chain(x, i, s, pk_seed, adrs: ADRS) -> Optional[bytes]:
    """
    Input: Input string X, start index i, number of steps s, public seed PK.seed, address
ADRS
    Output: value of F iterated s times on X
    """
    if s == 0:
        return bytes(x)

    if (i + s) > (w - 1):
        return None

    tmp = chain(x, i, s - 1, pk_seed, adrs)

    adrs.set_hash_address(i + s - 1)
    tmp = F(pk_seed, adrs, tmp)

    return tmp


def wots_sk_gen(sk_seed, adrs: ADRS):
    """
    Input: secret seed SK.seed, address ADRS
    Output: WOTS+ private key sk
    """
    sk = []
    for i in range(0, _len):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk.append(PRF(sk_seed, adrs))
    return sk


def wots_pk_gen(sk_seed, pk_seed, adrs: ADRS):
    """
    Input: secret seed SK.seed, address ADRS, public seed PK.seed
    Output: WOTS+ public key pk
    """
    wots_pk_adrs = adrs.copy()
    tmp = bytes()
    for i in range(0, _len):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk = PRF(sk_seed, adrs)
        tmp += bytes(chain(sk, 0, w - 1, pk_seed, adrs.copy()))

    wots_pk_adrs.set_type(ADRS.TYPE_WOTSPK)
    wots_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())

    pk = F(pk_seed, wots_pk_adrs, tmp)  # T_len(PK.seed, wotspkADRS, tmp);

    return pk


def wots_sign(m, sk_seed, pk_seed, adrs):
    """
```

```python
    Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
    Output: WOTS+ signature sig
    """
    csum = 0

    # convert message to base w
    msg = base_w(m, w, len_1)

    # compute checksum
    for i in range(len_1):
        csum = csum + w - 1 - msg[i]

    # convert checksum to base w
    if (lg(w) % 8) != 0:
        csum = csum << (8 - ((len_2 * lg(w)) % 8))

    len_2_bytes = ceil((len_2 * lg(w)) / 8)
    msg = msg + base_w(toByte(csum, len_2_bytes), w, len_2)

    sig = []
    for i in range(0, _len):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk = PRF(sk_seed, adrs.copy())
        sig.append(chain(sk, 0, msg[i], pk_seed, adrs.copy()))

    return sig


def wots_pk_from_sig(sig, m, pk_seed, adrs: ADRS):
    """
    Input: Message M, WOTS+ signature sig, address ADRS, public seed PK.seed
    Output: WOTS+ public key pk_sig derived from sig
    """
    csum = 0
    wots_pk_adrs = adrs.copy()

    # convert message to base w
    msg = base_w(m, w, len_1)

    # compute checksum
    for i in range(0, len_1):
        csum += w - 1 - msg[i]

    # convert checksum to base w
    csum = csum << (8 - ((len_2 * lg(w)) % 8))
    len_2_bytes = ceil((len_2 * lg(w)) / 8)
    msg = msg + base_w(toByte(csum, len_2_bytes), w, len_2)

    tmp = bytes()
    for i in range(0, _len):
        adrs.set_chain_address(i)
        tmp += chain(sig[i], msg[i], w - 1 - msg[i], pk_seed, adrs.copy())

    wots_pk_adrs.set_type(ADRS.TYPE_WOTSPK)
    wots_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())
    pk_sig = F(pk_seed, wots_pk_adrs, tmp)

    return pk_sig
```

XMSS (eXtended Merkle Signature Scheme)

As principais funções definidas para o esquema XMSS foram:

- `treehash`**: calcula o nodo raiz de uma árvore de altura** `h` **com a folha mais à esquerda com a chave** `pk` **no index** `s`**.**
- `xmss_pk_gen`**: gera a chave pública do esquema XMSS.**
- `xmss_sign`**: gera uma assinatura para a mensagem** `m` **com a chave privada** `sk`**.**
- `xmss_pk_from_sig`**: calcula a chave pública de uma assinatura**

In [9]:

```python
# XMSS Sub-Trees height
_h = h // d

def sig_wots_from_sig_xmss(sig):
    return sig[0:_len]


def auth_from_sig_xmss(sig):
    return sig[_len:]


def sigs_xmss_from_sig_ht(sig):
    sigs = []
    for i in range(0, d):
        sigs.append(sig[i * (_h + _len):(i + 1) * (_h + _len)])

    return sigs


def auths_from_sig_fors(sig):
    sigs = []
    for i in range(0, k):
        sigs.append([])
        sigs[i].append(sig[(a + 1) * i])
        sigs[i].append(sig[((a + 1) * i + 1):((a + 1) * (i + 1))])

    return sigs
```

In [10]:

```python
def treehash(secret_seed, s, z, public_seed, adrs: ADRS):
    """
    Input: Secret seed SK.seed, start index s, target node height z, public seed PK.seed,
address ADRS
    Output: n-byte root node - top node on Stack
    """
    if s % (1 << z) != 0:
        return -1

    stack = []

    for i in range(0, 2^z):
        adrs.set_type(ADRS.TYPE_WOTS)
        adrs.set_key_pair_address(s + i)
        node = wots_pk_gen(secret_seed, public_seed, adrs.copy())

        adrs.set_type(ADRS.TYPE_HASHTREE)
        adrs.set_tree_height(1)
        adrs.set_tree_index(s + i)

        if len(stack) > 0:
            while stack[len(stack) - 1]['height'] == adrs.get_tree_height():
                adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                node = H(public_seed, adrs.copy(), stack.pop()['node'] + node, n)
                adrs.set_tree_height(adrs.get_tree_height() + 1)

                if len(stack) <= 0:
                    break

        stack.append({'node': node, 'height': adrs.get_tree_height()})

    return stack.pop()['node']


def xmss_pk_gen(secret_seed, public_key, adrs: ADRS):
    """
    Input: Secret seed SK.seed, public seed PK.seed, address ADRS
    Output: XMSS public key PK
```

```python
    """
    pk = treehash(secret_seed, 0, _h, public_key, adrs.copy())
    return pk


def xmss_sign(m, secret_seed, idx, public_seed, adrs):
    """
    Input: n-byte message M, secret seed SK.seed, index idx, public seed PK.seed, address
ADRS
    Output: XMSS signature SIG_XMSS = (sig || AUTH)
    """
    auth = []
    for j in range(0, _h):
        ki = floor(idx // 2^j)
        if ki % 2 == 1: # XOR
            ki -= 1
        else:
            ki += 1

        auth.append(treehash(secret_seed, ki * 2^j, j, public_seed, adrs.copy()))

    adrs.set_type(ADRS.TYPE_WOTS)
    adrs.set_key_pair_address(idx)

    sig = wots_sign(m, secret_seed, public_seed, adrs.copy())
    sig_xmss = sig + auth
    return sig_xmss


def xmss_pk_from_sig(idx, sig_xmss, m, pk_seed, adrs):
    """
    Input: index idx, XMSS signature SIG_XMSS = (sig || AUTH), n-byte message M, public s
eed PK.seed, address ADRS
    Output: n-byte root value node[0]
    """
    adrs.set_type(ADRS.TYPE_WOTS)
    adrs.set_key_pair_address(idx)
    sig = sig_wots_from_sig_xmss(sig_xmss)
    auth = auth_from_sig_xmss(sig_xmss)

    node_0 = wots_pk_from_sig(sig, m, pk_seed, adrs.copy())

    adrs.set_type(ADRS.TYPE_HASHTREE)
    adrs.set_tree_index(idx)
    for i in range(0, _h):
        adrs.set_tree_height(i + 1)

        if math.floor(idx / 2^i) % 2 == 0:
            adrs.set_tree_index(adrs.get_tree_index() // 2)
            node_1 = H(pk_seed, adrs.copy(), node_0 + auth[i], n)
        else:
            adrs.set_tree_index( (adrs.get_tree_index() - 1) // 2)
            node_1 = H(pk_seed, adrs.copy(), auth[i] + node_0, n)

        node_0 = node_1

    return node_0
```

HyperTree

É essencialmente uma árvore de certificação de instâncias XMSS.

Foram definidas as seguintes funções:

- `ht_pkGen`**: gera a chave pública a partir da seed privada e pública.**
- `ht_sign`**: gera a assinatura a partir da mensagem, seed privada e pública, índice da árvore e índice da folha.**
- `ht_verify`**: verifica a assinatura a partir da mensagem, chave pública, índice da árvore e índice da folha.**

In [11]:

```
def ht_pkGen(sk_seed, pk_seed):
    """
    Input: Private seed SK.seed, public seed PK.seed
    Output: HT public key PK_HT
    """
    adrs = ADRS()  # Análogo a initializar 32 bytes a 0 (toBytes(0, 32) na especificação
)
    adrs.set_layer_address(d - 1)
    adrs.set_tree_address(0)
    root = xmss_pk_gen(sk_seed, pk_seed, adrs.copy())
    return root

def ht_sign(m, secret_seed, public_seed, idx_tree, idx_leaf):
    """
    Input: Message M, private seed SK.seed, public seed PK.seed, tree index idx_tree, lea
f index idx_leaf
    Output: HT signature SIG_HT
    """
    # init
    adrs = ADRS()  # Análogo a initializar 32 bytes a 0 (toBytes(0, 32) na especificação
)

    # sign
    adrs.set_layer_address(0)
    adrs.set_tree_address(idx_tree)
    sig_tmp = xmss_sign(m, secret_seed, idx_leaf, public_seed, adrs.copy())
    sig_ht = sig_tmp
    root = xmss_pk_from_sig(idx_leaf, sig_tmp, m, public_seed, adrs.copy())

    for j in range(1, d):
        idx_leaf = idx_tree % 2 ^ _h
        idx_tree = idx_tree >> _h

        adrs.set_layer_address(j)
        adrs.set_tree_address(idx_tree)

        sig_tmp = xmss_sign(root, secret_seed, idx_leaf, public_seed, adrs.copy())
        sig_ht = sig_ht + sig_tmp

        if j < d - 1:
            root = xmss_pk_from_sig(idx_leaf, sig_tmp, root, public_seed, adrs.copy())

    return sig_ht

def ht_verify(m, sig_ht, public_seed, idx_tree, idx_leaf, public_key_ht):
    """
    Input: Message M, signature SIG_HT, public seed PK.seed, tree index idx_tree, leaf in
dex idx_leaf, HT public key PK_HT
    Output: Boolean
    """
    # init
    adrs = ADRS()  # Análogo a initializar 32 bytes a 0 (toBytes(0, 32) na especificação
)

    # verify
    sigs_xmss = sigs_xmss_from_sig_ht(sig_ht)
    sig_tmp = sigs_xmss[0]
    adrs.set_layer_address(0)
    adrs.set_tree_address(idx_tree)
    node = xmss_pk_from_sig(idx_leaf, sig_tmp, m, public_seed, adrs)

    for j in range(1, d):
        idx_leaf = idx_tree % 2 ^ _h
        idx_tree = idx_tree >> _h

        sig_tmp = sigs_xmss[j]

        adrs.set_layer_address(j)
        adrs.set_tree_address(idx_tree)
```

```
        node = xmss_pk_from_sig(idx_leaf, sig_tmp, node, public_seed, adrs)

    if node == public_key_ht:
        return True
    else:
        return False
```

FORS (Forest of Random Subsets)

Função fors_SKgen:

- `fors_SKgen`**: gera a chave privada a partir da seed privada, endereço e índice.**
- `fors_treehash`**: calcula o nodo raiz da árvore de altura z com a folha mais à esquerda com o hash da chave privada no indice s.**
- `fors_PKgen`**: gera a chave pública a partir da seed privada e pública (não é usada no algoritmo em si, apenas para ajudar a perceber o funcionamento dos algoritmos).**
- `fors_sign`**: gera a assinatura a partir da mensagem, seed privada e pública, índice da árvore e índice da folha. -** `fors_pkFromSig`**: gera a chave pública a partir da assinatura, mensagem, índice da árvore, índice da folha, seed pública e endereço.**

In [12]:

```
def fors_SKgen(secret_seed, adrs: ADRS, idx):
    """
    Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
    Output: FORS private key sk
    """
    adrs.set_tree_height(0)
    adrs.set_tree_index(idx)
    sk = PRF(secret_seed, adrs.copy())

    return sk


def fors_treehash(secret_seed, s, z, public_seed, adrs):
    """
    Input: Secret seed SK.seed, start index s, target node height z, public seed PK.seed,
address ADRS
    Output: n-byte root node - top node on Stack
    """
    if s % (1 << z) != 0:
        return -1

    stack = []

    for i in range(0, 2^z):
        adrs.set_tree_height(0)
        adrs.set_tree_index(s + i)
        sk = PRF(secret_seed, adrs.copy())
        node = H(public_seed, adrs.copy(), sk, n)

        adrs.set_tree_height(1)
        adrs.set_tree_index(s + i)
        if len(stack) > 0:
            while stack[len(stack) - 1]['height'] == adrs.get_tree_height():
                adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                node = H(public_seed, adrs.copy(), stack.pop()['node'] + node, n)

                adrs.set_tree_height(adrs.get_tree_height() + 1)

                if len(stack) <= 0:
                    break
        stack.append({'node': node, 'height': adrs.get_tree_height()})

    return stack.pop()['node']
```

```
def fors_PKgen(secret_seed, public_seed, adrs: ADRS):
    """
    Input: Secret seed SK.seed, public seed PK.seed, address ADRS
    Output: FORS public key PK
    """
    fors_pk_adrs = adrs.copy()

    root = bytes()
    for i in range(0, k):
        root += fors_treehash(secret_seed, i * t, a, public_seed, adrs)

    fors_pk_adrs.set_type(ADRS.TYPE_FORSPK)
    fors_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())
    pk = H(public_seed, fors_pk_adrs, root)
    return pk

def fors_sign(m, secret_seed, public_seed, adrs):
    """
    Input: Bit string M, secret seed SK.seed, address ADRS, public seed PK.seed
    Output: FORS signature SIG_FORS
    """
    m_int = int.from_bytes(m, 'big')
    sig_fors = []

    for i in range(0, k):
        idx = (m_int >> (k - 1 - i) * a) % t

        adrs.set_tree_height(0)
        adrs.set_tree_index(i * t + idx)
        sig_fors += [PRF(secret_seed, adrs.copy())]

        auth = []

        for j in range(0, a):
            s = math.floor(idx // 2 ^ j)
            if s % 2 == 1:  # XOR 1
                s -= 1
            else:
                s += 1

            auth += [fors_treehash(secret_seed, i * t + s * 2^j, j, public_seed, adrs.co
py())]

        sig_fors += auth

    return sig_fors

def fors_pkFromSig(sig_fors, m, public_seed, adrs: ADRS):
    """
    Input: FORS signature SIG_FORS, (k lg t)-bit string M, public seed PK.seed, address A
DRS
    Output: FORS public key
    """
    m_int = int.from_bytes(m, 'big')

    sigs = auths_from_sig_fors(sig_fors)
    root = bytes()

    for i in range(0, k):
        idx = (m_int >> (k - 1 - i) * a) % t

        sk = sigs[i][0]
        adrs.set_tree_height(0)
        adrs.set_tree_index(i * t + idx)
        node_0 = H(public_seed, adrs.copy(), sk)

        auth = sigs[i][1]
        adrs.set_tree_index(i * t + idx)

        for j in range(0, a):
            adrs.set_tree_height(j+1)
```

```
            if math.floor(idx / 2^j) % 2 == 0:
                adrs.set_tree_index(adrs.get_tree_index() // 2)
                node_1 = H(public_seed, adrs.copy(), node_0 + auth[j], n)
            else:
                adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                node_1 = H(public_seed, adrs.copy(), auth[j] + node_0, n)

            node_0 = node_1

        root += node_0

    fors_pk_adrs = adrs.copy()
    fors_pk_adrs.set_type(ADRS.TYPE_FORSPK)
    fors_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())

    pk = H(public_seed, fors_pk_adrs, root, n)
    return pk
```

SPHINCS+

As principais funções desenvolvidas foram:

- `spx_keygen()` **: Gera um par de chaves SPHINCS+. Retorna um tuplo** `(SK, PK)` **, onde** `SK` **é a chave privada e** `PK` **é a chave pública.**
- `spx_sign(m, SK, PK)` **: Gera uma assinatura SPHINCS+ para a mensagem** `m` **com a chave privada** `SK` **e a chave pública** `PK` **. Retorna a assinatura correspondente.**
- `spx_verify(m, sig, PK)` **: Verifica se a assinatura** `sig` **é válida para a mensagem** `m` **e a chave pública** `PK` **. Retorna** `True` **se a assinatura for válida e** `False` **caso contrário.**

In [13]:

```
def sec_rand(n):
    return os.urandom(n)
```

Types used in SPHINCS+

In [14]:

```
from typing import Tuple

SK = Tuple[bytes, bytes, bytes, bytes]  # (SK.seed, SK.prf, PK.seed, PK.root) (n bytes, n bytes, n bytes, n bytes)

PK = Tuple[bytes, bytes]  # (PK.seed, PK.root) (n bytes, n bytes)

SIG = Tuple[bytes, bytes, bytes]  # (R, SIG_FORS, SIG_HT) (n bytes, k*(a+1)*n bytes, (h+d*len)*n bytes)
```

In [15]:

```
import math
import hashlib
import random  # Only for Pseudo-randoms
import os  # Secure Randoms


# Input: (none)
# Output: SPHINCS+ key pair (SK, PK)
def spx_keygen() -> (SK, PK):
    sk_seed = sec_rand(n)
    sk_prf = sec_rand(n)
    pk_seed = sec_rand(n)

    pk_root = ht_pkGen(sk_seed, pk_seed)

    sk: SK = (sk_seed, sk_prf, pk_seed, pk_root)
```

```
    pk: PK = (pk_seed, pk_root)

    return sk, pk
```

In [16]:

```
# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS+ signature SIG
def spx_sign(m: bytes, sk: SK) -> SIG:
    # init
    adrs = ADRS()   # Análogo a initializar 32 bytes a 0 (toBytes(0, 32) na especificação
)

    (sk_seed, sk_prf, pk_seed, pk_root) = sk

    # Generate randomizer
    opt = toByte(0, n)
    if RANDOMIZE:
        opt = sec_rand(n)
    R = PRF_msg(sk_prf, opt, m)
    sig = [R]   # SIG = SIG || R

    # Tamanhos a usar no digest
    size_md = floor((k * a + 7) / 8)
    size_idx_tree = floor((h - h / d + 7) / 8)

    # compute message digest and index

    digest = H_msg(R, pk_seed, pk_root, m)
    tmp_md = digest[:size_md]
    tmp_idx_tree = digest[size_md:(size_md + size_idx_tree)]
    tmp_idx_leaf = digest[(size_md + size_idx_tree):len(digest)]

    md_int = int.from_bytes(tmp_md, 'big') >> (len(tmp_md) * 8 - k * a)
    md = int(md_int).to_bytes(math.ceil(k * a / 8), 'big')

    idx_tree = int.from_bytes(tmp_idx_tree, 'big') >> (len(tmp_idx_tree) * 8 - (h - h //
d))
    idx_leaf = int.from_bytes(tmp_idx_leaf, 'big') >> (len(tmp_idx_leaf) * 8 - (h // d))

    # FORS sign
    adrs.set_layer_address(0)
    adrs.set_tree_address(idx_tree)
    adrs.set_type(ADRS.TYPE_FORSTREE)
    adrs.set_key_pair_address(idx_leaf)

    sig_fors = fors_sign(md, sk_seed, pk_seed, adrs.copy())
    sig += [sig_fors]

    # Get FORS public key
    pk_fors = fors_pkFromSig(sig_fors, md, pk_seed, adrs.copy())

    # Sign FORS public key with HT
    adrs.set_type(ADRS.TYPE_HASHTREE)
    sig_ht = ht_sign(pk_fors, sk_seed, pk_seed, idx_tree, idx_leaf)
    sig += [sig_ht]

    return sig
```

In [17]:

```
# Input: Message M, signature SIG, public key PK
# Output: Boolean
def spx_verify(m: bytes, sig: SIG, public_key: PK) -> bool:
    # init
    adrs = ADRS()   # Análogo a initializar 32 bytes a 0 (toBytes(0, 32) na especificação
)
    (r, sig_fors, sig_ht) = sig

    public_seed = public_key[0]
    public_root = public_key[1]
```

```
    # Tamanhos a usar no digest
    size_md = math.floor((k * a + 7) / 8)
    size_idx_tree = math.floor((h - h // d + 7) / 8)

    # Compute message digest and index
    digest = H_msg(r, public_seed, public_root, m)
    tmp_md = digest[:size_md]
    tmp_idx_tree = digest[size_md:(size_md + size_idx_tree)]
    tmp_idx_leaf = digest[(size_md + size_idx_tree):len(digest)]

    md_int = int.from_bytes(tmp_md, 'big') >> (len(tmp_md) * 8 - k * a)
    md = int(md_int).to_bytes(math.ceil(k * a / 8), 'big')

    idx_tree = int.from_bytes(tmp_idx_tree, 'big') >> (len(tmp_idx_tree) * 8 - (h - h //
d))
    idx_leaf = int.from_bytes(tmp_idx_leaf, 'big') >> (len(tmp_idx_leaf) * 8 - (h // d))

    # Compute FORS public key
    adrs.set_layer_address(0)
    adrs.set_tree_address(idx_tree)
    adrs.set_type(ADRS.TYPE_FORSTREE)
    adrs.set_key_pair_address(idx_leaf)

    pk_fors = fors_pkFromSig(sig_fors, md, public_seed, adrs)

    # Verify HT signature
    adrs.set_type(ADRS.TYPE_HASHTREE)
    return ht_verify(pk_fors, sig_ht, public_seed, idx_tree, idx_leaf, public_root)
```

Testes/Exemplos

Por fim, foi testada a implementação do SPHINCS+, com casos de teste bastante simples:

- **Geração de chaves**
- **Assinatura da mensagem**
- **Verificação da assinatura**
- **Verificação de uma assinatura com uma mensagem diferente (de modo, a falhar a verificação)**

In [18]:

```
%%time

# Gerar as chaves pública e privada
sk, pk = spx_keygen()
print("Par de chaves gerado com sucesso!")
```

```
Par de chaves gerado com sucesso!
CPU times: user 1.22 s, sys: 0 ns, total: 1.22 s
Wall time: 1.21 s
```

In [19]:

```
%%time
m = b"Grande SPHINCS+!"

print("Mensagem:", m)

s = spx_sign(m, sk)
```

```
Mensagem: b'Grande SPHINCS+!'
CPU times: user 9.61 s, sys: 15 ms, total: 9.63 s
Wall time: 9.66 s
```

In [20]:

```
%%time
print("Mensagem assinada com sucesso!")
```

```
print("Assinatura correta!" if spx_verify(m, s, pk) else "Assinatura INCORRETA!")
```

```
Mensagem assinada com sucesso!
Assinatura correta!
CPU times: user 15 ms, sys: 0 ns, total: 15 ms
Wall time: 9.6 ms
```

In [21]:

```
%%time
print("Mensagem assinada com sucesso!")

m = b"Grande SPHINCS++!"  # Mensagem diferente
print("Assinatura correta!" if spx_verify(m, s, pk) else "Assinatura INCORRETA!")
```

```
Mensagem assinada com sucesso!
Assinatura INCORRETA!
CPU times: user 16 ms, sys: 0 ns, total: 16 ms
Wall time: 10.1 ms
```