Sphincs+

May 10, 2023

1 Estruturas Criptográficas: Trabalho Prático 4

Grupo 13, constituído por: – Rodrigo Rodrigues, PG50726
– Rui Guilherme Monteiro, PG50739

1.1 SPHINCS+

Referências: https://sphincs.org/data/sphincs+-round2-specification.pdf

Parâmetros SPHINCS+ n: Parâmetro de segurança
w: Parâmetro de Winternitz
h: Altura da Hypertree
d: Número de camadas da Hypertree
k: Número de árvores na FORS
t: Número de folhas de uma árvore FORS

1.1.1 Inicialização dos parâmetros

[1]:
```
RANDOMIZE = True
n = 32
w = 256
h = 12
d = 3
k = 8
a = 4

# Message Length for WOTS
len_1 = math.ceil(8 * n / math.log(w, 2))
len_2 = math.floor(math.log(len_1 * (w - 1), 2) / math.log(w, 2)) + 1
len_0 = len_1 + len_2

# XMSS Sub-Trees height
h_prime = h // d

# FORS trees leaves number
t = 2^a
```

1.1.2 Classe ADRS (Hash Function Address Scheme)

[2]:
```python
class ADRS:
    # TYPES
    WOTS_HASH = 0
    WOTS_PK = 1
    TREE = 2
    FORS_TREE = 3
    FORS_ROOTS = 4

    def __init__(self):
        self.layer = 0
        self.tree_address = 0

        self.type = 0

        # Words for which role can change depending on ADRS.type
        self.word_1 = 0
        self.word_2 = 0
        self.word_3 = 0

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

        adrs.type = self.type
        adrs.word_1 = self.word_1
        adrs.word_2 = self.word_2
        adrs.word_3 = self.word_3
        return adrs

    def to_bin(self):
        adrs = int(self.layer).to_bytes(4, byteorder='big')
        adrs += int(self.tree_address).to_bytes(12, byteorder='big')

        adrs += int(self.type).to_bytes(4, byteorder='big')
        adrs += int(self.word_1).to_bytes(4, byteorder='big')
        adrs += int(self.word_2).to_bytes(4, byteorder='big')
        adrs += int(self.word_3).to_bytes(4, byteorder='big')

        return adrs

    def reset_words(self):
        self.word_1 = 0
        self.word_2 = 0
        self.word_3 = 0
```

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

        self.word_2 = 0
        self.word_3 = 0
        self.word_1 = 0


    # Getters & Setters

    def set_layer_address(self, val):
        self.layer = val

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

    def set_key_pair_address(self, val):
        self.word_1 = val

    def get_key_pair_address(self):
        return self.word_1

    def set_chain_address(self, val):
        self.word_2 = val

    def set_hash_address(self, val):
        self.word_3 = val

    def set_tree_height(self, val):
        self.word_2 = val

    def get_tree_height(self):
        return self.word_2

    def set_tree_index(self, val):
        self.word_3 = val

    def get_tree_index(self):
        return self.word_3
```

1.1.3 Métodos auxiliares

Strings of Base-w Numbers Na função base_w, são passados a string x, o inteiro w e o comprimento do output, out_len, a função retorna um array base-w de inteiros de comprimento out_len.

[3]:
```python
def base_w(x, w, out_len):
    """
        Parâmetros:
        ----------
        x: string de len_X bytes
        w: int
        out_len: length do output

        Retorna: array de int basew com comprimento out_len
    """
    vin = 0
    vout = 0
    total = 0
    bits = 0
    basew = []

    for consumed in range(0, out_len):
        if bits == 0:
            total = x[vin]
            vin += 1
            bits += 8
        bits -= math.floor(math.log(w, 2))
        basew.append((total >> bits) % w)
        vout += 1

    return basew
```

1.1.4 Tweakable Hash Functions & UTILS

As tweakable hash functions permitem tornar as chamadas das funções de hash independentes entre cada par e posição na árvore virtual da estrutura do SPHINCS+.

[5]:
```python
def hash(seed, adrs: ADRS, value, digest_size=n):
    m = hashlib.sha256()

    m.update(seed)
    m.update(adrs.to_bin())
    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_bin(), "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 sig_wots_from_sig_xmss(sig):
    return sig[0:len_0]


def auth_from_sig_xmss(sig):
    return sig[len_0:]


def sigs_xmss_from_sig_ht(sig):
    sigs = []
    for i in range(0, d):
        sigs.append(sig[i*(h_prime + len_0):(i+1)*(h_prime + len_0)])
```

```
    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
```

1.1.5 FORS

Esta classe usa as suas chaves públicas para assinar os "digests" das mensagens.

Esta classe usa os parâmetros os seguintes parâmetros:

n - Parâmetro de segurança. É o tamanho da chave privada, da chave pública ou da assinatura em bytes.
k - Número de sets da chave privada, de árvores e de indices computados da string de input.
a - Número usado para computar o parâmetro.

Valor computado pela classe:
t - Número de elementos em cada set da chave privada, número de folhas por hash tree e valor superior a todos os valores de index. Tem de ser múltiplo de 2, como as árvores têm altura e a string de input é dividida em strings de tamanho.

Funções 1. **FORS Private Key** Esta função recebe como parâmetros uma secret seed, um endereço FORS ADRS e o indice da folha a ser usada idx. Esta função gera a chave privada da classe FORS om recurso á função de hash definida na secção e a um endereço FORS.

2. **FORS TreeHash** Esta função recebe como parâmetros uma secret seed, o index de partida s, a altura do nodo a calcular z, uma public seed e um endereço que codifica o par de chaves FORS a usar adrs. Esta função retorna o "root node" de uma árvore de altura z com a folha mais á esquerda sendo a chave pública FORS no index s.

3. **FORS Public Key** Esta função recebe como parâmetros uma secret seed, uma public seed e um edereço FORS adrs. Esta função gera uma chave pública FORS com recurso á função e á aplicação de uma função de hash definida na secção sobre o resultado da função anterior.

4. **FORS Signature Generation** Esta função recebe como parâmetros uma menagem m, uma secret seed, uma public seed e um endereço adrs. Esta função tem como objetivo gerar uma assinatura de tamanho com strings de tamanho. Essa assinatura contem k valores da chave privada, com n bytes cada, e os respetivos caminhos de autenticação AUTH.

5. **FORS Compute Public Key from Signature** Esta função recebe como parâmetros uma sssinatura sig_fors, uma mensagem m, uma public seed e um endereço adrs. Esta função obtem a chave pública a partir da ssinatura.

Notas Importantes

Por estarmos a trabalhar no Sagemath precisamos de obter resultados mais rápidos por isso fizemos uso de uma dica fornecida na página 37 do pdf do SPHINCS+, que está transcrita abaixo:

"Shorter Outputs. If a parameter set requires an output length n < 32-bytes for F, H, PRF, and PRFmsg we take the first n bytes of the output and discard the remaining."

[10]:
```
def fors_sk_gen(secret_seed, adrs: ADRS, idx):
    """
        Parâmetros:
        -----------
        secret_seed: SK.seed
        adrs: address
        idx: secret key index = it + j

        Retorna: 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):
    """
        Parâmetros:
        -----------
        secret_seed: SK.seed
        s: start index
        z: target node height
        public_seed: PK.seed
        adrs: address

        Retorna: 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 = hash(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 = hash(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_pk_gen(secret_seed, public_seed, adrs: ADRS):
    """
        Parâmetros:
        -----------
        secret_seed: SK.seed
        public_seed: PK.seed
        adrs: address

        Retorna: 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.FORS_ROOTS)
    fors_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())
    pk = hash(public_seed, fors_pk_adrs, root)
    return pk

def fors_sign(m, secret_seed, public_seed, adrs):
    """
        Parâmetros:
        -----------
        m: Bit string
        secret_seed: SK.seed
        public_seed: PK.seed
        adrs: address

        Retorna: 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:  # XORING idx/ 2**j with 1
                s -= 1
            else:
                s += 1

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

        sig_fors += auth

    return sig_fors

def fors_pk_from_sig(sig_fors, m, public_seed, adrs: ADRS):
    """
        Parâmetros:
        -----------
        sig_fors: FORS signature
        m: (k lg t)-bit string
        public_seed: PK.seed
        adrs: address

        Retorna: 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 = hash(public_seed, adrs.copy(), sk)
        node_1 = 0

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

        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 = hash(public_seed, adrs.copy(), node_0 + auth[j], n)
            else:
                adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                node_1 = hash(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.FORS_ROOTS)
    fors_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())

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

1.1.6 HyperTree

Para construir a hypertree do SPHINCS+ é inicialmente combinado o WOTS+ com uma árvore binária de hash, obtendo assim uma versão com input de tamanho fixo do eXtended Merkle Signature Scheme (XMSS).

Esta classe contem várias instancias de todas com a mesma altura, equipara-se portanto a uma árvore em que em cada nodo tem árvores XMSS.

Esta classe usa os parâmetros h, d, h' e w:

h - Altura da árvore HyperTree.
d - Número de camandas de árvores XMSS existentes na HyperTree.
h' - Altura das árvores XMSS.
w - Parâmetro de Winternitz. Usado por todas as camandas de árvores XMSS.

Funções 1. **HT Key Generation**

2. **HT Signature Generation**

3. **HT Signature Verification**

[19]:
```
def ht_pk_gen(secret_seed, public_seed):
    """
        Esta função tem como chave secreta a secret seed que será usada para gerar todas as chaves privadas WOTS+ da HyperTree.
        Já a chave pública é o "root node" da árvore XMSS no topo da HyperTree.

        Parâmetros:
        -----------
        secret_seed: private seed SK.seed
        public_seed: public seed PK.seed

        Retorna: HT public key
    """
    adrs = ADRS()
    adrs.set_layer_address(d - 1)
    adrs.set_tree_address(0)
    root = xmss_pk_gen(secret_seed, public_seed, adrs.copy())
    return root

def ht_sign(m, secret_seed, public_seed, idx_tree, idx_leaf):
    """
        Recebe uma mensagem m, uma secret seed, uma public seed, o index da árvore a ser usada para assinar a mensagem idx_tree e o index da folha da árvore XMSS a ser usada para assinar a mensagem idx_leaf.
        Esta função tem como objetivo assinar uma mensagem m. A assinatura resultante será uma string de bytes de tamanho (h + d*len0)*n
        Consistindo estas de 'd' assinaturas XMSS de tamanho ((h/d)+len0)*n.
        Esta função começa por definir o endereço da árvore escolhida para assinar a mensagem, assinando-a em seguida.
        Depois aplica assinaturas consecutivas de árvores XMSS entre a árvore escolhida e a raiz da HyperTree.

        Parâmetros:
        -----------
        m: mensagem
        secret_seed: private seed SK.seed
        public_seed: public seed PK.seed
        idx_tree: tree index
        idx_leaf: leaf index

        Retorna: HT signature SIG_HT
    """
    adrs = ADRS()
    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_prime
        idx_tree = idx_tree >> h_prime

        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):
    """
        Esta função recebe como parâmetros uma mensagem m, uma assinatura␣
↪sig_ht, uma public seed, o index da folha da HyperTree (árvore XMSS) a ser␣
↪usada para assinar a mensagem idx_tree, o index da folha da árvore XMSS a␣
↪ser usada para assinar a mensagem idx_leaf e a chave pública da HyperTree␣
↪public_key_ht.
        Verifica as assinaturas feitas pelas árvores XMSS na função `ht_sign`.
        Processo: Começamos por obter a chave pública da assinatura recebida e␣
↪posteriormente realizamos 'd' vezes a mesma operação para cada árvore XMSS␣
↪usada no sign entre a árvore na folha da HyperTree e a árvore XMSS na raiz␣
↪da HyperTree.

        Parâmetros:
        -----------
        m: mensagem
        public_seed: public seed PK.seed
        idx_tree: tree index
        idx_leaf: leaf index
        public_key_ht: HT public key

        Retorna: Boolean
    """
    adrs = ADRS()

    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_prime
        idx_tree = idx_tree >> h_prime

        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
```

1.1.7 WOTS+

Esta classe usa os parâmetros n e w, que usa para calcular as variáveis len_0, len1 e len2.

n - Parâmetro de Segurança. Trata-se do tamanho da mensagem, bem como da chave privada, pública ou assinatura em bytes.
w - Parâmetro de Winternitz. Corresponde a um dos 3 elementos {4,16,256}.

Funções:
1. **WOTS+ Chaining Function**
2. **WOTS+ Private Key Generation**
3. **WOTS+ Public Key Generation**
4. **WOTS+ Signature Generation**
5. **WOTS+ Compute Public Key from Signature**

[27]:
```
def chain(x, i, s, public_seed, adrs: ADRS):
    """
        WOTS+ Chaining Function

        Esta função computa uma iteração de um hash definido em 'Tweakables' sobre um input de n bytes usando um endereço de hash WOTS+ ao qual se dá o nome de ADRS e a chave pública.
        Parâmetros:
        -----------
        x: string
```

```
        i: posição inicial
        s: numero de iterações
        public_seed: public seed
        adrs: endereço WOTS+ ADRS

        Retorna: value of F iterated s times on X
    """
    if s == 0:
        return bytes(x)

    if (i + s) > (w - 1):
        return -1

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

    adrs.set_hash_address(i + s - 1)
    #F
    tmp = hash(public_seed, adrs, tmp, n)

    return tmp

def wots_sk_gen(secret_seed, adrs: ADRS):
    """
        WOTS+ Private Key Generation

        Esta função gera uma chave privada WOTS+ de n bytes fazendo uso da␣
 ↪função de hash que se encontra definida na secção 'Tweakables'.

        Parâmetros:
        -----------
        secret_seed
        adrs: endereço WOTS+ ADRS

        Retorna: WOTS+ private key sk
    """
    sk = []
    for i in range(0, len_0):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk.append(prf(secret_seed, adrs.copy()))
    return sk

def wots_pk_gen(secret_seed, public_seed, adrs: ADRS):
    """
        WOTS+ Public Key Generation

        Gera uma chave pública que consiste em n chains de tamanho w.
```

```
        Cada string que compõe a chave secreta é usada com ponto de partida␣
 ↪para a aplicação da função chain.
        É aplicada posterirmente uma função de hash definida na secção␣
 ↪'Tweakables'.
.
        Parâmetros:
        -----------
        secret_seed
        public_seed
        adrs: endereço WOTS+ ADRS

        Retorna: WOTS+ public key pk
    """
    wots_pk_adrs = adrs.copy()
    tmp = bytes()
    for i in range(0, len_0):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk = prf(secret_seed, adrs.copy())
        tmp += bytes(chain(sk, 0, w - 1, public_seed, adrs.copy()))

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

    #T_len
    pk = hash(public_seed, wots_pk_adrs, tmp)
    return pk


def wots_sign(m, secret_seed, public_seed, adrs):
    """
        WOTS+ Signature Generation

        Esta função mapeia a mensagem m para n inteiros entre 0 e w-1.
        Esta mensagem m começa por ser transformada em números em base w usando␣
 ↪a função definida na secção de Métodos Auxiliares.
        Depois, realiza-se um checksum à mensagem original e após transformar o␣
 ↪checksum numa mensagem de len2 número em base w este é somado à mensagem␣
 ↪trasformada.
        Depois usam-se os números na mensagem final para selecionar um nodo␣
 ↪diferente na função 'chain', a assinatura será composta pela concatenação de␣
 ↪todos estes nodos.
        Parâmetros:
        -----------
        m: mensagem
        secret_seed
        public_seed
```

```
        adrs: endereço WOTS+ ADRS

        Retorna: WOTS+ signature sig
    """
    csum = 0

    msg = base_w(m, w, len_1)

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

    padding = (len_2 * math.floor(math.log(w, 2))) % 8 if (len_2 * math.
↪floor(math.log(w, 2))) % 8 != 0 else 8
    csum = csum << (8 - padding)
    csumb = int(csum).to_bytes(math.ceil((len_2 * math.floor(math.log(w, 2))) /␣
↪8), byteorder='big')
    csumw = base_w(csumb, w, len_2)
    msg += csumw

    sig = []
    for i in range(0, len_0):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk = prf(secret_seed, adrs.copy())
        sig += [chain(sk, 0, msg[i], public_seed, adrs.copy())]

    return sig


def wots_pk_from_sig(sig, m, public_seed, adrs: ADRS):
    """
        WOTS+ Compute Public Key from Signature

        Computa a chave pública WOTS+ através da assinatura sig "completando"␣
↪as computações usando a função `chain`.
        Começa por emular o funcionamento da função `wots_sign` até ao for loop␣
↪par a geração da assinatura.
        Em seguida usa a assinatura e, em vez começar as iterações da chain na␣
↪posição 0 começa na posição indicada pelo número na posição 'i' da mensagem␣
↪trasformada e itera  w - 1 - msg[i] vezes.

        Parâmetros:
        -----------
        m: mensagem
        secret_seed
        public_seed
        adrs: endereço WOTS+ ADRS
```

```
        Retorna: WOTS+ signature sig
    """
    csum = 0
    wots_pk_adrs = adrs.copy()

    msg = base_w(m, w, len_1)

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

    padding = (len_2 * math.floor(math.log(w, 2))) % 8 if (len_2 * math.
↪floor(math.log(w, 2))) % 8 != 0 else 8
    csum = csum << (8 - padding)
    csumb = int(csum).to_bytes(math.ceil((len_2 * math.floor(math.log(w, 2))) /␣
↪8), byteorder='big')
    csumw = base_w(csumb, w, len_2)
    msg += csumw

    tmp = bytes()
    for i in range(0, len_0):
        adrs.set_chain_address(i)
        tmp += chain(sig[i], msg[i], w - 1 - msg[i], public_seed, adrs.copy())

    wots_pk_adrs.set_type(ADRS.WOTS_PK)
    wots_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())
    pk_sig = hash(public_seed, wots_pk_adrs, tmp)
    return pk_sig
```

1.1.8 XMSS

Esta classe usa os parâmetros:

h - Define a altura da árvore - 1 (número de níveis -1).
n - Parâmetro de segurança. Tamanho em bytes das mensagensbem como de cada nodo.
w - Parâmetro de Winternitz. Corresponde a um dos 3 elementos {4,16,256}.
A árvore tem 2^h folhas (as folhas são as chaves públicas WOTS+), ou seja, um par de chaves XMSS para uma altura pode ser usado para assinar 2^h mensagens diferentes.

Funções
1. **TreeHash**
Retorna o "root node" de uma árvore de altura z com a folha mais à esquerda sendo a chave pública WOTS+ no index s.

2. **XMSS Public Key Generation** Usa função usa a função `treehash` para gerar a chave pública.

3. **XMSS Signature Generation**
Retorna uma assinatura XMSS. Estas assinaturas têm tamanho (len0 + h)*n e são compostas

por uma assinatura WOTS+ de tamanho len_0 e o caminho de autenticação AUTH para a folha associada com o par de chaves usad na assinatura WOTS+.

4. **XMSS Compute Public Key from Signature**
   Esta função computa um "root node". Esta função começa por chamar a função `wots_pk_from_sig` para obter um candidato a chave pública WOTS+. Posteriormente este resultado é usado juntamente com os valores do AUTH para obter o "root node".

[28]:
```
def treehash(secret_seed, s, z, public_seed, adrs: ADRS):
    """
    TreeHash

    Parâmetros:
    -----------
    secret_seed
    s: posição inicial
    z: altura do nodo alvo
    public_seed
    adrs: address

    Retorna: "root node" de n-bytes da árvore de altura z (top node on Stack)
    """
    if s % (1 << z) != 0:
        return -1

    stack = []

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

        adrs.set_type(ADRS.TREE)
        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 = hash(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):
    """
    XMSS Public Key Generation

    Parâmetros:
    -----------
    secret_seed
    public_key
    adrs: address

    Retorna: XMSS public key PK
    """
    pk = treehash(secret_seed, 0, h_prime, public_key, adrs.copy())
    return pk

def xmss_sign(m, secret_seed, idx, public_seed, adrs):
    """
    XMSS Signature Generation

    Parâmetros:
    -----------
    m: mensagem
    secret_seed
    idx
    public seed
    adrs: address

    Retorna: XMSS signature SIG_XMSS = (sig || AUTH)
    """
    auth = []
    for j in range(0, h_prime):
        ki = math.floor(idx // 2^j)
        if ki % 2 == 1: # XOR
            ki -= 1
        else:
            ki += 1

        auth += [treehash(secret_seed, ki * 2^j, j, public_seed, adrs.copy())]

    adrs.set_type(ADRS.WOTS_HASH)
    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, public_seed, adrs):
    """
    XMSS Compute public key from signature

    Parâmetros:
    -----------
    idx
    sig_xmss: assinatura xmss
    m: mensagem de n-bytes
    public seed
    adrs: address

    Retorna: n-byte root value node[0]
    """
    adrs.set_type(ADRS.WOTS_HASH)
    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, public_seed, adrs.copy())
    node_1 = 0

    adrs.set_type(ADRS.TREE)
    adrs.set_tree_index(idx)
    for i in range(0, h_prime):
        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 = hash(public_seed, adrs.copy(), node_0 + auth[i], n)
        else:
            adrs.set_tree_index( (adrs.get_tree_index() - 1) // 2)
            node_1 = hash(public_seed, adrs.copy(), auth[i] + node_0, n)

        node_0 = node_1

    return node_0
```

1.1.9 SPHINCS+

Esta classe usa os parâmetros:

n - Parâmetro de segurança.
w - Parâmetro de Winternitz.
h - Altura da HyperTree.
d - Número de camadas da HyperTree.

k - Número de árvores no FORS.
t - Número de folhas de uma árvore FORS.

[31]:
```
import math
import hashlib
import random  # Pseudo-randoms
import os      # Secure Randoms

class Sphincs:

    def generate_key_pair(self):
        """
        Esta função gera os parâmetros: secret_seed, secret_prf e public_seed.
        Retorna: Key Pair (SK,PK), ou seja, dois arrays
                 - o primeiro corresponde à secret key e é composto por␣
 ↪secret_seed, secret_prf, public_seed e public_root
                 - segundo corresponde à public key e é composto por␣
 ↪public_seed e public_root.
        """
        secret_seed = os.urandom(n)
        secret_prf = os.urandom(n)
        public_seed = os.urandom(n)

        public_root = ht_pk_gen(secret_seed, public_seed)

        return [secret_seed, secret_prf, public_seed, public_root],␣
 ↪[public_seed, public_root]

    def sign(self, m, secret_key):
        """
        Sign

        Parâmetros:
        -----------
        m: mensagem
        secret_key

        Retorna: assinatura SPHINCS+
        """
        # Init
        adrs = ADRS()
        secret_seed = secret_key[0]
        secret_prf = secret_key[1]
        public_seed = secret_key[2]
        public_root = secret_key[3]

        # Generate randomizer
```

```
        opt = bytes(n)
        if RANDOMIZE:
            opt = os.urandom(n)
        r = prf_msg(secret_prf, opt, m)
        sig = [r]

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

        digest = hash_msg(r, public_seed, public_root, m, size_md +␣
↪size_idx_tree + size_idx_leaf)

        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)]

        # Conversões
        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.FORS_TREE)
        adrs.set_key_pair_address(idx_leaf)

        sig_fors = fors_sign(md, secret_seed, public_seed, adrs.copy())
        sig += [sig_fors]

        # Get FORS public key
        pk_fors = fors_pk_from_sig(sig_fors, md, public_seed, adrs.copy())

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

        return sig

    def verify(self, m, sig, public_key):
```

```
        """
        Verify
        Parâmetros:
        -----------
        m: mensagem
        sig: assinatura
        public_key

        Retorna: Boolean
        """
        # Init
        adrs = ADRS()
        r = sig[0]
        sig_fors = sig[1]
        sig_ht = sig[2]

        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)
        size_idx_leaf = math.floor((h // d + 7) / 8)

        # Compute message digest and index
        digest = hash_msg(r, public_seed, public_root, m, size_md +␣
↪size_idx_tree + size_idx_leaf)
        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.FORS_TREE)
        adrs.set_key_pair_address(idx_leaf)

        pk_fors = fors_pk_from_sig(sig_fors, md, public_seed, adrs)
```

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

1.1.10 Teste

[32]:
```
sphincs = Sphincs()

secret_key, public_key = sphincs.generate_key_pair()

print("Secret key: ", secret_key)
print("Public key: ", public_key)


message = b"SPHINCS+ do grupo 13 de EC 22/23."
print("Mensagem a ser assinada: ", message)

signature = sphincs.sign(message, secret_key)

print("Assinatura correta? ", sphincs.verify(message, signature, public_key))
```

```
Secret key:  [b'\x9fb\xb8\x81&\xbd\x89\x0f\x84Bsu3\xfb\x08\xf0q\xe3\xcez\x14\xe1
4\xf8\x88\x0b\x91\x8bj\xea\t\xff', b'\x9b\x8aS\xe6\xe4t\rq\x8fA\x03\x060P\xfdx\x
18\xde\x12n\xd6\xeeY\xbb\x87\xf7\xc0b\xff\x94]\xd8',
b'\x88\xa2\xec\xb6\xfar\xc84\x0e\xd3\xae\x15\x99g\x9fM\xabL\xb6\xe2$O\xa2(
O\x17\xb3\xa6\x1e\xbd\x05', b'\xe8~xsf\x95\x11\xcc\xe6\xe4\x08\xe7>\xea\x87\xc7`
\x82\x04\xf4\xa4\x19z4\x84\xdd\xca\x1c\xc8=\xde-']
Public key:
[b'\x88\xa2\xec\xb6\xfar\xc84\x0e\xd3\xae\x15\x99g\x9fM\xabL\xb6\xe2$O\xa2(
O\x17\xb3\xa6\x1e\xbd\x05', b'\xe8~xsf\x95\x11\xcc\xe6\xe4\x08\xe7>\xea\x87\xc7`
\x82\x04\xf4\xa4\x19z4\x84\xdd\xca\x1c\xc8=\xde-']
Mensagem a ser assinada:  b'SPHINCS+ do grupo 13 de EC 22/23.'
Assinatura correta?  True
```