TP4_Exercicio2

May 25, 2024

1 TP4 - Exercício 2

1.0.1 Autores

Afonso Ferreira - pg52669
Tiago Rodrigues - pg52705

1.0.2 Enunciado

Implemente um protótipo do esquema descrito na norma FIPS 205 que deriva do algoritmo SPHINCS+.

1.1 Parâmetros

Este documento descreve a implementação do esquema de assinatura SPHINCS+ em conformidade com o padrão FIPS 205. SPHINCS+ é um esquema de assinatura pós-quântica baseado em funções hash, projetado para resistir a ataques de computadores quânticos. A implementação utiliza os seguintes parâmetros:

Parâmetros de Segurança e Configuração:

- `n = 32`: Parâmetro de segurança, que define o tamanho das assinaturas e a força de segurança geral.
- `w = 256`: Parâmetro de Winternitz, que controla o trade-off entre tamanho da assinatura e velocidade de assinatura/verificação.
- `len_1`, `len_2`, `len_0`: Parâmetros que definem o comprimento das mensagens para o esquema WOTS+ (Winternitz One-Time Signature Plus) usado dentro do SPHINCS+.
- `h = 12`: Altura da hiperárvore, que determina o número de níveis na estrutura da árvore usada para gerar assinaturas.
- `d = 3`: Número de camadas na hiperárvore.
- `h_prime = h // d`: Altura das subárvores XMSS (eXtended Merkle Signature Scheme).
- `k = 8`: Número de árvores FORS (Forest of Random Subsets).
- `a = 4`: Número de folhas nas árvores FORS.
- `t = 2^a`: Número total de folhas nas árvores FORS.

[ ]:
```
# The security parameter
n = 32

# Winternitz parameter
w = 256
```

```
# 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

# Hypertree height
h = 12

# Hypertree layers
d = 3

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

# FORS trees numbers
k = 8

# FORS trees leaves number
a = 4
t = 2^a
```

1.2 Classe ADRS

A classe ADRS (Address) é utilizada para gerir endereços dentro da estrutura da árvore do SPHINCS+

[ ]:
```
class ADRS:
    # TYPES
    WOTS_HASH = 0
    WOTS_PK = 1
    TREE = 2
    FORS_TREE = 3
    FORS_ROOTS = 4
    WOTS_PRF = 5
    FORS_PRF = 6

    def __init__(self):
        self.layer = 0
        self.tree_address = 0
        self.type = 0
        self.word_1 = 0
        self.word_2 = 0
        self.word_3 = 0

    def copy(self):
        adrs = ADRS()
```

```python
        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_1 = 0
        self.word_2 = 0
        self.word_3 = 0

    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):
```

```python
        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
```

1.3 Funções Auxiliares

[ ]:
```python
import random
import hashlib
```

`base_w(x, w, out_len)` - Converte uma sequência de bytes `x` numa representação numérica na base `w` (parâmetro de Winternitz). - `out_len` especifica o número de dígitos desejados na saída. - A função percorre os bytes de entrada, extraindo dígitos na base `w` até atingir o comprimento desejado.

[ ]:
```python
# Input: len_X-byte string X, int w, output length out_len
# Output: out_len int array basew
def base_w(x, w, out_len):
    v_in = 0
    v_out = 0
    total = 0
    bits = 0
    basew = []
    consumed = 0
    while (consumed < out_len):
        if bits == 0:
            total = x[v_in]
            v_in += 1
            bits += 8
        bits -= math.floor(math.log(w, 2))
        basew.append((total >> bits) % w)
        v_out += 1
        consumed += 1
    return basew
```

`hash(seed, adrs, value, digest_size=n)`

- Calcula o hash de uma mensagem utilizando a função hash SHA-512.
- `seed` é uma semente inicial para o hash.
- `adrs` é um objeto `ADRS` que contém informações de endereço.
- `value` é a mensagem a ser hasheada.
- `digest_size` especifica o tamanho desejado do hash (padrão é `n`).

[ ]:
```python
def hash(seed, adrs: ADRS, value, digest_size = n):
    m = hashlib.sha512()

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

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

    return hashed
```

`prf(public_seed, secret_seed, adrs)`

- Função pseudoaleatória que gera valores secretos com base nas sementes e no endereço.
- Utiliza a biblioteca `random` para gerar um número aleatório e converte-o em bytes.

[ ]:
```python
def prf(public_seed, secret_seed, adrs):
    random.seed(int.from_bytes(public_seed + secret_seed + adrs.to_bin(),␣
 ↪"big"))
    return int(random.randint(0, 256 ^ n)).to_bytes(n, byteorder='big')
```

`hash_msg(r, public_seed, public_root, value, digest_size=n)`

- Calcula o hash de uma mensagem com parâmetros adicionais, como o *randomizer* `r`, a `public_seed` e a `public_root`.
- Se o hash resultante for menor que `digest_size`, são adicionados bytes até atingir o tamanho desejado.

[ ]:
```python
def hash_msg(r, public_seed, public_root, value, digest_size=n):
    m = hashlib.sha512()

    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.sha512()

        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
```

`prf_msg(secret_seed, opt, m)`

Função pseudoaleatória que gera um valor aleatório a partir da `secret_seed`, um valor opcional `opt` e a mensagem `m`.

```
[ ]: 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')
```

1.4 Funções Auxiliares para Assinaturas XMSS

`sig_wots_from_sig_xmss(sig)`

- Extrai a assinatura WOTS+ (`sig`) da assinatura XMSS completa.
- A assinatura WOTS+ é a parte inicial da assinatura XMSS, com comprimento `len_0`.

`auth_from_sig_xmss(sig)`

- Extrai o caminho de autenticação da assinatura XMSS.
- O caminho de autenticação é a parte restante da assinatura, após a assinatura WOTS+.

`sigs_xmss_from_sig_ht(sig)`

- Extrai as assinaturas XMSS individuais da assinatura da Hypertree (`sig_ht`).
- Cada assinatura XMSS tem comprimento `h_prime + len_0`, e há `d` assinaturas no total.

`auths_from_sig_fors(sig)`

- Extrai os caminhos de autenticação das árvores FORS a partir da assinatura FORS (`sig_fors`).
- Cada caminho de autenticação FORS consiste numa folha (`sig[(a+1) * i]`) e um caminho na árvore (`sig[((a+1) * i + 1):((a+1) * (i+1))]`).

```
[ ]: 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
```

1.5 Funções WOTS+ (Winternitz One-Time Signature Plus)

`chain(x, i, s, public_seed, adrs)`

- Calcula o resultado da função hash iterada `s` vezes sobre a entrada `x`, começando no índice `i`.
- Utiliza a função `hash` para calcular os valores da cadeia.
- O endereço `adrs` é atualizado a cada iteração.

`wots_pk_gen(secret_seed, public_seed, adrs)`

- Gera a chave pública WOTS+ a partir da `secret_seed`, da `public_seed` e do `adrs`.
- Calcula os valores da cadeia para cada posição na assinatura WOTS+ e faz um hash destes para obter a chave pública.

`wots_sign(m, secret_seed, public_seed, adrs)`

- Gera a assinatura WOTS+ da mensagem `m` utilizando a `secret_seed`, a `public_seed` e o `adrs`.
- Converte a mensagem para a base `w`, calcula o checksum e gera os valores da cadeia para cada posição na assinatura.

`wots_pk_from_sig(sig, m, public_seed, adrs)`

- Reconstrói a chave pública WOTS+ a partir da assinatura `sig`, da mensagem `m`, da `public_seed` e do `adrs`.
- Calcula os valores da cadeia com base na assinatura e na mensagem, e faz um hash destes para obter a chave pública.

[ ]:
```
# 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
def chain(x, i, s, public_seed, adrs: ADRS):
    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)
    tmp = hash(public_seed, adrs, tmp, n)
    return tmp

# Input: secret seed SK.seed, address ADRS, public seed PK.seed
# Output: WOTS+ public key pk
```

```python
def wots_pk_gen(secret_seed, public_seed, adrs: ADRS):
    wots_pk_adrs = adrs.copy()
    wots_pk_adrs.set_type(ADRS.WOTS_PRF)
    wots_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())

    tmp = bytes()
    for i in range(0, len_0):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk = prf(public_seed, secret_seed, adrs.copy())
        tmp += bytes(chain(sk, 0, w - 1, public_seed, adrs.copy()))

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

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

# Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
# Output: WOTS+ signature sig
def wots_sign(m, secret_seed, public_seed, adrs):
    csum = 0

    # 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 csum to base w
    if (len_2 * math.floor(math.log(w, 2))) % 8 != 0:
        csum = csum << (8 - (len_2 * math.floor(math.log(w, 2))) % 8 )
    len2_bytes = math.ceil((len_2 * math.floor(math.log(w, 2))) / 8)
    msg += base_w(int(csum).to_bytes(len2_bytes, byteorder='big'), w, len_2)

    wots_pk_adrs = adrs.copy()
    wots_pk_adrs.set_type(ADRS.WOTS_PRF)
    wots_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())

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

```python
    return sig

def wots_pk_from_sig(sig, m, public_seed, adrs: ADRS):
    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 csum to base w
    if (len_2 * math.floor(math.log(w, 2))) % 8 != 0:
        padding = (len_2 * math.floor(math.log(w, 2))) % 8
    else:
        padding = 8
    csum = csum << (8 - padding)
    msg += base_w(int(csum).to_bytes(math.ceil((len_2 * math.floor(math.log(w,␣
 ↪2))) / 8), byteorder='big'), w, len_2)

    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
```

1.6 Funções XMSS (eXtended Merkle Signature Scheme)

`xmss_node(secret_seed, s, z, public_seed, adrs)`

- Calcula o nó raiz de uma árvore XMSS de altura `z`, começando no índice `s`.
- Utiliza a função `wots_pk_gen` para gerar as chaves públicas WOTS+ dos nós folha e a função `hash` para calcular os nós internos.

`xmss_sign(m, secret_seed, idx, public_seed, adrs)`

- Gera a assinatura XMSS da mensagem `m` utilizando a `secret_seed`, o índice do par de chaves `idx`, a `public_seed` e o `adrs`.
- Constrói o caminho de autenticação e a assinatura WOTS+ da mensagem.

`xmss_pk_from_sig(idx, sig_xmss, m, public_seed, adrs)`

- Reconstrói a chave pública XMSS (raiz da árvore) a partir da assinatura XMSS, da mensagem,

da `public_seed` e do `adrs`.
- Utiliza a função `wots_pk_from_sig` para obter a chave pública WOTS+ e o caminho de autenticação para calcular a raiz da árvore.

[ ]:
```
# 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
def xmss_node(secret_seed, s, z, public_seed, adrs: ADRS):
    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']

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

```
            ki += 1
        auth += [xmss_node(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

# Input: index idx, XMSS signature SIG_XMSS = (sig || AUTH), n-byte message M,␣
↪public seed PK.seed, address ADRS
# Output: n-byte root value node[0]
def xmss_pk_from_sig(idx, sig_xmss, m, public_seed, adrs):
    # compute WOTS+ pk from WOTS+ sig
    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)

    node0 = wots_pk_from_sig(sig, m, public_seed, adrs.copy())
    node1 = 0

    # compute root from WOTS+ pk and AUTH
    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)
            node1 = hash(public_seed, adrs.copy(), node0 + auth[i], n)
        else:
            adrs.set_tree_index( (adrs.get_tree_index() - 1) // 2)
            node1 = hash(public_seed, adrs.copy(), auth[i] + node0, n)
        node0 = node1

    return node0
```

1.7 Funções FORS (Forest of Random Subsets) e Hypertree

As funções `fors_sk_gen`, `fors_node`, `fors_sign`, `fors_pk_from_sig` e `ht_sign`, `ht_verify` são análogas às funções XMSS, mas operam nas árvores FORS e na Hypertree, respectivamente.

[ ]:
```
# Input: Message M, private seed SK.seed, public seed PK.seed, tree index␣
↪idx_tree, leaf index idx_leaf
# Output: HT signature SIG_HT
def ht_sign(m, secret_seed, public_seed, idx_tree, idx_leaf):
    # init
```

```python
    adrs = ADRS()

    # 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_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

# Input: Message M, signature SIG_HT, public seed PK.seed, tree index idx_tree,␣
 ↪leaf index idx_leaf, HT public key PK_HT
# Output: Boolean
def ht_verify(m, sig_ht, public_seed, idx_tree, idx_leaf, public_key_ht):
    # init
    adrs = ADRS()

    # 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_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
```

```python
# Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
# Output: FORS private key sk
def fors_sk_gen(secret_seed, public_seed, adrs: ADRS, idx):
    wots_pk_adrs = adrs.copy()
    wots_pk_adrs.set_type(ADRS.FORS_PRF)
    wots_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())

    adrs.set_tree_height(0)
    adrs.set_tree_index(idx)
    sk = prf(public_seed, secret_seed, adrs.copy())
    return sk

# 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
def fors_node(secret_seed, s, z, public_seed, adrs):
    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(public_seed, 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']

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

    sig_fors = []
    for i in range(0, k):
```

```
        # get next index
        idx = (m_int >> (k - 1 - i) * a) % t

        # pick private key element
        adrs.set_tree_height(0)
        adrs.set_tree_index(i * t + idx)
        sig_fors += [prf(public_seed, secret_seed, adrs.copy())]

        # compute auth path
        auth = []
        for j in range(0, a):
            s = math.floor(idx // 2 ^ j)
            if s % 2 == 1:
                s -= 1
            else:
                s += 1
            auth += [fors_node(secret_seed, i * t + s * 2^j, j, public_seed,
 ↪adrs.copy())]
        sig_fors += auth

    return sig_fors

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

    sigs = auths_from_sig_fors(sig_fors)
    root = bytes()

    # compute roots
    for i in range(0, k):
        # get next index
        idx = (m_int >> (k - 1 - i) * a) % t

        # compute leaf
        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

        # compute root from lead and AUTH
        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 = 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() # copy address to create FTS public key address
    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
```

1.8 Funções de Assinatura e Verificação

[ ]:
```
import os
```

`slh_keygen()`

- Gera um par de chaves (chave privada e chave pública).
- A chave privada consiste em *seeds* privadas e públicas, e a chave pública inclui a `public_seed` e a `public_root` da árvore XMSS.

[ ]:
```
# Input: (none)
# Output: key pair (SK,PK)
def slh_keygen():
    secret_seed = os.urandom(n)
    secret_prf = os.urandom(n)
    public_seed = os.urandom(n)

    adrs = ADRS()
    adrs.set_layer_address(d - 1)
    public_root = xmss_node(secret_seed, 0, h_prime, public_seed, adrs.copy())

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

`slh_sign(m, secret_key)`

- Assina uma mensagem `m` utilizando a chave privada `secret_key`.
- Gera um *randomizer* `r` e calcula o hash da mensagem.

- Utiliza as funções `fors_sign` e `ht_sign` para gerar as assinaturas FORS e HT, respectivamente.

[ ]:
```python
RANDOMIZE = True

# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS+ signature SIG
def slh_sign(m, secret_key):
    # 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 = public_seed
    if RANDOMIZE:
        opt = os.urandom(n)
    r = prf_msg(secret_prf, opt, m)
    sig = [r]

    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))

    # 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
```

`slh_verify(m, sig, public_key)`

- Verifica a assinatura `sig` de uma mensagem `m` utilizando a chave pública `public_key`.
- Calcula o hash da mensagem e extrai os índices da árvore.
- Utiliza as funções `fors_pk_from_sig` e `ht_verify` para verificar as assinaturas FORS e HT, respectivamente.

```
[ ]: # Input: Message M, signature SIG, public key PK
# Output: Boolean
def slh_verify(m, sig, public_key):
    # init
    adrs = ADRS()
    r = sig[0]
    sig_fors = sig[1]
    sig_ht = sig[2]

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

    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)
```

1.9 Teste

[ ]:
```
m = b'Hello there!'
print("\nMessage to be signed:\n", m)
```

```
Message to be signed:
 b'Hello there!'
```

[ ]:
```
# Generate key pair
sk, pk = slh_keygen()

print("Private key:\n", sk)
print("\nPublic key:\n", pk)
```

```
Private key:
 [b'\xb1\xff \xff\xb4\x98\x15\x03\xcc1\xab\xd1\xaf$c(\x8cu\rz\xdb\xa4:,\xafX\x1c
\xa5#\x19\xcd\x98', b'e\x921\x03y\xf3ok\x15KO\x0f\xeb\x9bu\x1e\x97\xff\x9f\xac<\
x96\x84p/\xc1\xdbB\xbd\x91\x9a\x95', b'i\x87\xbc\xe9)sO\x0f{\xc5\xc2\xe3/u\x9e\x
a8)<\x850\xbf\x9d<Hi\x8db\xa3V9@\xda', b'\x81\xc0.\x97KA\x84\xdb\x00\xf5\xc1
\x94\xcao\xb0{~\x8d\xb6\x1f\xad\xf5t\xa3*@\x16\x83\xe9-\xee']

Public key:
 [b'i\x87\xbc\xe9)sO\x0f{\xc5\xc2\xe3/u\x9e\xa8)<\x850\xbf\x9d<Hi\x8db\xa3V9@\xd
a', b'\x81\xc0.\x97KA\x84\xdb\x00\xf5\xc1
\x94\xcao\xb0{~\x8d\xb6\x1f\xad\xf5t\xa3*@\x16\x83\xe9-\xee']
```

[ ]:
```
s = slh_sign(m, sk)

print("\nVerifying signature...\n", slh_verify(m, s, pk))
```

```
Verifying signature…
 True
```