Estruturas Criptográficas 2022/23 — TP4. Problema 2

Grupo 7. Leonardo Berteotti e Paulo R. Pereira

May 30, 2023

O SPHINCS+ é um esquema de assinatura digital baseado em *hashes* criptográficos que é seguro contra ataques de computadores quânticos. Destaca-se que é um esquema *stateless*, ou seja, não necessita de registar informações após cada assinatura, torndo-o um bom candidato para o concurso.

Este documento segue a mesma estrutura que o *paper* fornecido na diretoria da UC, com a única diferença que os parâmetros são dados no início, já que são requeridos ao longo da implementação.

1 Parâmetros do esquema SPHINCS+

[126]:
```
# 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
```

2 Funções auxiliares

2.1 Strings of Base-w Numbers

[127]:
```
# 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
```

2.2 Hash Function Address Scheme (Structure of ADRS)

[128]:
```
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
        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

    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
```

2.3 Tweakable Hash Functions

[129]:
```
import random
import hashlib

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
```

3 WOTS+ One-Time Signatures

3.1 Parâmetros

1. n - parâmetro de Segurança: trata-se do tamanho da mensagem, bem como da chave privada, pública ou assinatura em bytes.
2. w - parâmetro de Winternitz: corresponde a um elemento do conjunto $\{4, 16, 256\}$.

3.2 WOTS+ Chaining Function (Function chain)

A função *chain* recebe uma string x, uma posição inicial i, um número de iterações a serem realizadas s, uma public *seed* e um endereço WOTS+ ADRS. A função retorna uma iteração de um *tweakables hash* sobre um *input* de n bytes usando um endereço de *hash* WOTS+ denominado ADRS e a chave pública.

[130]:
```python
# 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
```

3.3 WOTS+ Private Key (Function *wots_skGen*)

A função *wots_skGen* recebe uma *secret seed* e um endereço WOTS+ ADRS e retorna uma chave privada WOTS+ de n bytes que deve ser usada para assinar uma única mensagem. Assinar mais do que uma mensagem com a mesma chave compromete a segurança do esquema.

[131]:
```python
# Input: secret seed SK.seed, address ADRS
# Output: WOTS+ private key sk
def wots_sk_gen(secret_seed, adrs: ADRS):
    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
```

3.4 WOTS+ Public Key Generation (Function *wots_pkGen*)

A função *wots_pkGen* recebe uma *secret seed* uma *public seed* e um endereço WOTS+ ADRS. A função gera uma chave pública - n *chains* de tamanho w.

[132]:
```python
# Input: secret seed SK.seed, address ADRS, public seed PK.seed
# Output: WOTS+ public key pk
def wots_pk_gen(secret_seed, public_seed, adrs: ADRS):
    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())

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

3.5 WOTS+ Signature Generation (Function *wots_sign*)

A função *wots_sign* recebe uma mensagem m, uma *secret seed*, uma *public seed* e um endereço WOTS+ ADRS.
A função trata de maepar a mensagem m para n inteiros entre 0 e $w-1$. Começa-se por transformar a mensagem m em len_1 números de base w. Depois, é realizado um *checksum* à mensagem original, transforma-se o *checksum* numa mensagem de len_2 número em base w e somam-se as duas transformações. Por fim, os números na mensagem final são usados para selecionar um nodo *hash chain* diferente. A assinatura é a concatenação de todos os nodos.

[133]:
```
# 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)

    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
```

3.6 WOTS+ Compute Public Key from Signature (Function *wots_pkFromSig*)

A função *wots_pkFromSig* recebe uma assinatura *sig*, uma mensagem m, uma *public seed* e um endereço WOTS+ ADRS.
A função gera a chave pública WOTS+ através da assinatura *sig*. Diferente da função *wots_sign*, a

função *wots_ pkFromSig* usa a assinatura, ao invés da chave secreta, e começa a iterarar na posição i da mensagem trasformada $w - 1 - msg[i]$ vezes.

[134]:
```
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
```

4 The SPHINCS+ Hypertree

4.1 (Fixed Input-Length) XMSS

4.1.1 XMSS Parameters

1. h - Altura da árvore (número de níveis - 1);
2. n - Parâmetro de segurança, i.e., tamanho em bytes das mensagens bem como de cada nodo.
3. w - Parâmetro de Winternitz.

A árvore tem 2^h folhas (as folhas são as chaves públicas WOTS+), i.e., um par de chaves XMSS para uma altura h pode ser usado para assinar 2^h mensagens diferentes.

4.1.2 TreeHash (Function *treehash*)

A função *treehash* recebe uma *secret seed*, um índice inicial s, a altura do nodo escolhido z, uma *public seed* e um endereço que codifica a árvore ADRS.
A função retorna a raíz de uma árvore de altura z com a folha mais à esquerda, estando a chave pública WOTS+ no índice s.

[135]:
```
# 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 treehash(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']
```

4.1.3 XMSS Public Key Generation (Function *xmss_PKgen*)

A função *xmss_PKgen* recebe uma *secret seed*, uma *public key*, e um endereço ADRS. A função recorre à função *treehash* para gerar a chave pública.

[136]:
```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: XMSS public key PK
def xmss_pk_gen(secret_seed, public_key, adrs: ADRS):
    pk = treehash(secret_seed, 0, h_prime, public_key, adrs.copy())
```

```
    return pk
```

4.1.4 XMSS Signature Generation (Function xmss_sign)

A função *xmss_sign* recebe uma mensagem m, uma *secret seed*, o índice do par de chaves WOTS+ *idx*, uma *public seed* e um endereço ADRS. O objetivo é gerar uma assinatura XMSS. Estas assinaturas têm tamanho $(len_0 + h) * n$ e são compostas por uma assinatura WOTS+ de tamanho *len_0* e o caminho de autenticação AUTH para a folha associada ao par de chaves usadas na assinatura WOTS+.

[137]:
```
# 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 += [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
```

4.1.5 XMSS Compute Public Key from Signature (Function *xmss_pkFromSig*)

A função *xmss_pkFromSig* recebe o indice da assinatura *idx*, uma assinatura XMSS *sig_xmss*, uma mensagem m, uma *public seed* e um endereço ADRS.
A partir da função $wots_pk_from_sig$, obtém-se um candidato a chave pública WOTS+, que é usado com os valores do AUTH para obter o nodo da raiz.

[138]:
```
# 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
```

4.2 HT: The Hypertee

4.2.1 HT Parameters

1. h - Altura da árvore;
2. d - Número de camandas de árvores XMSS existentes;
3. h' - Altura das (sub-)árvores XMSS, que é igual a $h \,//\, d$;
4. w - Parâmetro de Winternitz.

4.2.2 HT Key Generation (Function *ht_PKgen*)

A função *ht_PKgen* recebe uma *secret seed* e uma *public seed*. A partir da *secret seed*, a função gera todas as chaves privadas WOTS+ da HyperTree.

[139]:
```
# Input: Private seed SK.seed, public seed PK.seed
# Output: HT public key PK_HT
def ht_pk_gen(secret_seed, public_seed):
    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
```

4.2.3 HT Signature (Function *ht_sign*)

A função *ht_sign* recebe uma mensagem m, uma *secret seed*, uma *public seed*, o índice *idx_tree* da sub-árvore da HyperTree (árvore XMSS) e o índice idx_leaf da folha da árvore XMSS. O objetivo é assinar a mensagem m. A assinatura será uma string de bytes de tamanho $(h + d * len_0) * n$, i.e. d assinaturas XMSS de tamanho $((h/d) + len_0) * n$.
Começa-se por assinar a mensagem a partir do endereço da árvore escolhida. Depois, assina-se

consecutivamente a partir das árvores XMSS que estão no caminho desde a árvore escolhida à raiz da HyperTree.

[140]:
```
# 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
    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
```

4.2.4 HT Signature Generation (Function *ht_sign*)

A função *ht_sign* recebe uma mensagem m, uma assinatura *sig_ht*, uma *public seed*, o índice *idx_tree* da sub-árvore da HyperTree (i.e. uma árvore XMSS) a ser usada para assinar a mensagem, o índice *idx_leaf* da folha da árvore XMSS a ser usada para assinar a mensagem e a chave pública da HyperTree *public_key_ht*. A verificação de assinatura HyperTree pode ser resumida em comparar o valor recebdio com as d iterações de *xmss_pkFromSig*.

[141]:
```
# 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
```

5 FORS: Forest Of Random Subsets

5.1 FORS Parameters

1. n - Parâmetro de segurança: tamanho da chave privada, da chave pública ou da assinatura em bytes;
2. k - Número de *sets* da chave privada, de árvores e de indices computados da string de input;
3. t - Número de elementos em cada *set* da chave privada, número de folhas por *hash tree* e o majorante de todos os índices. Tem de ser múltiplo de 2, portanto, $t = 2^a$ para um dado a. Escolheu-se $a = 4$.

5.2 FORS Private Key (Function *fors_SKgen*)

A função *fors_SKgen* recebe uma *secret_seed*, um endereço FORS ADRS, e o índice *idx* da folha a ser usada.
O objetivo é gerar a chave privada FORS.

[142]:
```
# Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
# Output: FORS private key sk
def fors_sk_gen(secret_seed, adrs: ADRS, idx):
    adrs.set_tree_height(0)
    adrs.set_tree_index(idx)
    sk = prf(secret_seed, adrs.copy())
    return sk
```

5.3 FORS TreeHash (Function *fors_treehash*)

A função *fors_treehash* recebe uma *secret_seed*, o índice de partida s, a altura do nodo a calcular z, uma *public_seed* e um endereço que codifica o par de chaves FORS. A função retorna a raiz de uma árvore de altura z, sendo a folha mais à esquerda no índice s a chave pública FORS.

[143]:
```
# 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_treehash(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(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']
```

5.4 FORS Public Key (Function *fors_PKgen*)

A função *fors_PKgen* recebe uma *secret_seed*, uma *public_seed* e um edereço FORS ADRS. O objetivo é gerar, com recurso à função $fors_treehash$ e uma *tweakable hash function*, uma chave pública FORS.

[144]:
```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: FORS public key PK
def fors_pk_gen(secret_seed, public_seed, adrs: ADRS):
    fors_pk_adrs = adrs.copy() # copy address to create FTS public key address

    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
```

5.5 FORS Signature Generation (Function *fors_sign*)

A função *fors_sign* recebe uma menagem m, uma *secret_seed*, uma *public_seed* e um endereço ADRS.
O objetivo é gerar uma assinatura de tamanho $k(log2(t) + 1)$ com strings de tamanho n. Essa assinatura contem k valores da chave privada, com n bytes cada, e os respetivos caminhos de autenticação AUTH.

[145]:
```
# 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(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_treehash(secret_seed, i * t + s * 2^j, j, public_seed,␣
 ↪adrs.copy())]
        sig_fors += auth

    return sig_fors
```

5.6 FORS Compute Public Key from Signature (Function *fors_pkFromSig*)

A função *fors_pkFromSig* recebe uma sssinatura *sig_fors*, uma mensagem m, uma *public_seed* e um endereço ADRS.
O objetivo é gerar a chave pública a partir da assinatura.

[146]:
```
# 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
```

6 SPHINCS+

6.1 SPHINCS+ Key Generation (Function *spx_keygen*)

São gerados: *secret_seed*, *secret_prf* e *public_seed*. A função retorna dois arrays: *[secret_seed, secret_prf, public_seed, public_root]* como sendo a *private key* e *[public_seed, public_root]* como sendo a *public key*.

[147]:
```
import os

# Input: (none)
# Output: SPHINCS+ key pair (SK,PK)
def spx_key_gen():
    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]
```

6.2 SPHINCS+ Signature Generation (Function *spx_sign*)

A função *spx_sign* recebe uma mensagem m, e a chave secreta *secret_key*. Ela gera uma assinatura de tamanho $(1 + k(a+1) + h + d * len_0) * n$ sobre a mensagem recebida. A assinatura é constituida por uma string R aleatória de n bytes, uma assinatura FORS com $k(a+1)$ strings de tamanho n e uma assinatura da HyperTree com $h + d * len_0$ strings de n bytes.

[148]:
```
RANDOMIZE = True

# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS+ signature SIG
def spx_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 = bytes(n)
    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
```

6.3 SPHINCS+ Signature Verification (Function *spx_verify*)

A função *spx_verify* recebe uma mensagem m, uma assinatura *sig* e uma chave publica *public_key*. A função calcula o *digest* da mensagem e recorre às funções $fors_pk_from_sig$ e ht_verify para verificar se a assinatura corresponde à mensagem.

[149]:
```
# Input: Message M, signature SIG, public key PK
# Output: Boolean
def spx_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)
```

7 Exemplo de teste

[150]:
```
# Generate key pair
sk, pk = spx_key_gen()

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

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

s = spx_sign(m, sk)

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

```
Private key:
 [b'5\xa8*\xa2dB\x87\xea\xec*\xa9F\xc6\x8e\x83\x10\xc7\xea\xd0\xcb\x7f\x9c\xba\x
e6\xc1\xb8\xd6\xb3\x1d2\xb9\xcf', b'\x99\x81\xa4\xe4\xb2\x05\xaf\xb9\xa4\xc0\x15
\xd1\xcb\x82\xc9T\x0c\xe7Up\xf3\x18\x91\x05\x94j\xd0\x9d\xeb\xaa3\xa2', b'\x98\x
00\xddk\x15\xcbh\xf7R.\x1b\xb5\x82\xe6\xa2V\x1aS\xf3\xc14\x08n\xbc\x9c>>cK\xb0>\
xb9', b't\xfc\xe4\xd7\xf3\xe07*\x15\xc5,\xc5\xb8\x0b\x93\xc8#\x13\xb3\\\t\x1f\xa
5m\xe2\xa2\xcd|\xc5\x8f\xbf\x0c']

Public key:
 [b'\x98\x00\xddk\x15\xcbh\xf7R.\x1b\xb5\x82\xe6\xa2V\x1aS\xf3\xc14\x08n\xbc\x9c
>>cK\xb0>\xb9', b't\xfc\xe4\xd7\xf3\xe07*\x15\xc5,\xc5\xb8\x0b\x93\xc8#\x13\xb3\
\\t\x1f\xa5m\xe2\xa2\xcd|\xc5\x8f\xbf\x0c']

Message to be signed:
 b'Hello there!'

Verifying signature...
 True
```