SPHINCS+

June 23, 2020

1 SPHINCS+

SPHINCS+ recebe os parâmetros:

- n: Pârametro de segurança
- w: Pârametro Winternitz
- h: Altura da Hypertree
- d: Nº de camadas da Hypertree
- k: Nº de árvores no FORS
- t: Nº de folhas de uma árvore do FORS

[1]:
```
import math
import hashlib
import random  # Only for Pseudo-randoms
import os # Secure Randoms
```

1.1 Classe ADRS - Hash Function Address Scheme

[2]:
```
class ADRS:
    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
```

```python
        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 = self.layer.to_bytes(4, byteorder='big')
        adrs += self.tree_address.to_bytes(12, byteorder='big')
        adrs += self.type.to_bytes(4, byteorder='big')
        adrs += self.word_1.to_bytes(4, byteorder='big')
        adrs += self.word_2.to_bytes(4, byteorder='big')
        adrs += 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
```

1.2 Strings of Base-w Numbers

Uma vez que uma string byte pode ser considerada uma string de números base *w*, com esta função é-lhe dado como input uma string x, um inteiro w bem como o comprimento do output. Neste sentido, a função implementada, retornar um array base_w de inteiro com o comprimento conhecido.

[3]:
```
def base_w(x, w, 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
```

1.3 Inicialização dos Parâmetros

[4]:
```
_randomize = True

_n = 16
_w = 16
_h = 64
_d = 8
_k = 10
_a = 15
_t = 2 ** _a

_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
_h_prime = _h // _d


def calculate_variables():
    _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
    _h_prime = _h // _d
    _t = 2 ** _a
```

1.4 Funções Auxiliares

De forma a auxiliar a nossa solução, são definidos os gets e sets que iriam permitir, como os próprios nomes indicam, obter o valor das variáveis e atualizá-las.

[5]:
```
def set_security( val):
    _n = val
    calculate_variables()

def set_n( val):
    _n = val
    calculate_variables()

def get_security():
    return _n

def set_W( val):
    if val == 4 or val == 16 or val == 256:
        _w = val
    calculate_variables()

def set_w( val):
    if val == 4 or val == 16 or val == 256:
        _w = val
    calculate_variables()

def get_winternitz():
    return _w

def set_hypertree_height( val):
    _h = val
    calculate_variables()

def set_h( val):
    _h = val
```

```
    calculate_variables()

def get_hypertree_height():
    return _h

def set_hypertree_layers( val):
    _d = val
    calculate_variables()

def set_d( val):
    _d = val
    calculate_variables()

def get_hypertree_layers():
    return _d

def set_fors_trees_number( val):
    _k = val
    calculate_variables()

def set_k( val):
    _k = val
    calculate_variables()

def get_fors_trees_number():
    return _k

def set_fors_trees_height( val):
    _a = val
    calculate_variables()

def set_a( val):
    _a = val
    calculate_variables()

def get_fors_trees_height():
    return _a
```

1.5 Tweakable Hash Functions

De acordo com o documento de auxílio, estas funções recebem uma public seed e um endereço **ADRS** em conjunto com a mensagem. Desta forma, é-nos possível tornar as chamadas das funções de hash independentes entre cada par e posição na árvore virtual da estrutura do **SPHINCS+**.

[6]:
```
def hash(seed, adrs: ADRS, value, digest_size):
    m = hashlib.sha256()
    m.update(seed)
```

```python
    m.update(adrs.to_bin())
    m.update(value)
    hashed = m.digest()[:digest_size]
    return hashed


def prf(secret_seed, adrs, digest_size):
    random.seed(int.from_bytes(secret_seed + adrs.to_bin(), "big"))
    return random.randint(0, 256 ** digest_size - 1).to_bytes(digest_size, byteorder='big')


def hash_msg(r, public_seed, public_root, value, digest_size):
    m = hashlib.sha256()
    m.update(r)
    m.update(public_seed)
    m.update(public_root)
    m.update(value)
    hashed = m.digest()[:digest_size]

    i = 0
    while len(hashed) < digest_size:
        i += 1
        m = hashlib.sha256()
        m.update(r)
        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, digest_size):
    random.seed(int.from_bytes(secret_seed + opt + hash_msg(b'0', b'0', b'0', m, digest_size * 2), "big"))
    return random.randint(0, 256 ** digest_size - 1).to_bytes(digest_size, byteorder='big')

def print_bytes_bit(value):
    array = []
    for val in value:
        for j in range(7, -1, -1):
            array.append((val >> j) % 2)
    print(array)
```

1.6 WOTS+

O WOTS+ significa que cada chave privada apenas pode ser utilizada para assinar uma única mensagem.

Esta função computa a iteração num input de n bytes, utilizando o endereço de hash WOTS+ $ADRS$ e a public seed.

[7]:
```python
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
```

1.6.1 Chave Privada e Pública

[8]:
```python
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(), _n))
    return sk

def wots_pk_gen( secret_seed, public_seed, adrs: ADRS):

    # copy address to create OTS public key address
    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(), _n)
        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, _n)
    return pk
```

1.6.2 Gerar Assinatura

[9]:
```
def wots_sign( m, secret_seed, public_seed, adrs):
    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 = 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(), _n)
        sig += [chain(sk, 0, msg[i], public_seed, adrs.copy())]

    return sig
```

1.6.3 Obter Chave Pública a partir da Assinatura

[10]:
```
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
    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 = 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, _n)
    return pk_sig
```

2 Hypertree

De forma a 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)**.

2.0.1 Threehash

[11]:
```
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']
```

2.0.2 Gerar Chave Pública

[12]:
```
def xmss_pk_gen( secret_seed, public_key, adrs: ADRS):
    pk = treehash(secret_seed, 0, _h_prime, public_key, adrs.copy())
    return pk
```

2.0.3 Gerar Assinatura

[13]:
```
def xmss_sign( m, secret_seed, idx, public_seed, adrs):
    auth = []
    # build authentication path
    for j in range(0, _h_prime):
        ki = math.floor(idx // 2 ** j)
        if ki % 2 == 1:  # XORING idx/ 2**j with 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
```

2.0.4 Obter Chave Pública através da Assinatura

[14]:
```
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)

    node_0 = wots_pk_from_sig(sig, m, public_seed, adrs.copy())
    node_1 = 0

    # compute root from WOTS+ pk and AUTH
    adrs.set_type(ADRS.TREE)
    adrs.set_tree_index(idx)
```

```python
    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
```

2.1 HT

A Hypertree HT é uma variante do XMSS, que é essencialmente uma árvore de certificação de instâncias XMSS, ou seja, é no fundo uma árvore de várias camadas de árvores XMSS.

2.1.1 Gerar Chave Pública

[15]:
```python
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
```

2.1.2 Gerar Assinatura

O índice identifica uma folha da hypertree que irá ser utilizada para assinar a mensagem. A assinatura consiste numa stack de assinaturas XMSS.

[16]:
```python
def ht_sign( m, secret_seed, public_seed, idx_tree, idx_leaf):
    # init
    adrs = ADRS()
    adrs.set_layer_address(0)
    adrs.set_tree_address(idx_tree)

    # sign
    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
```

2.1.3 Verificar Assinatura

[17]:
```
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
```

3 FORS - Forest Of Random Subsets

A Hypertree HT não é utilizada para assinar as mensagens mas sim as chaves públicas de instâncias FORS, que são, estas sim, utilizadas para assinar as mensagens.

O SPHINCS+ utiliza verificação implicita para FORS, em que utiliza apenas um método para calcular uma chave pública candidata de uma assinatura.

3.0.1 Gerar Chave Privada

[18]:
```
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(), _n)

    return sk
```

3.0.2 ThreeHash

[19]:
```
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(), _n)
        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']
```

3.0.3 Gerar Chave Pública

[20]:
```
def fors_pk_gen( secret_seed, public_seed, adrs: ADRS):

    # copy address to create FTS public key address
    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, _n)
    return pk
```

3.0.4 Gerar Assinatura

[21]:
```
def fors_sign( m, secret_seed, public_seed, adrs):
    m_int = int.from_bytes(m, 'big')
    sig_fors = []

    # compute signature elements
    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(), _n)]

        auth = []

        # compute auth path
        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
```

```python
    return sig_fors
```

3.0.5 Obter Chave Pública através da Assinatura

[22]:
```python
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, _n)
        node_1 = 0

        # compute root from leaf and AUTH
        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

    # copy address to create FTS public key address
    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())
```

```python
    pk = hash(public_seed, fors_pk_adrs, root, _n)
    return pk
```

4 Classe SHPINCS+

[23]:
```python
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
```

4.0.1 Gerar Chaves

[24]:
```python
def gerar_chaves():
    """
    Generate a key pair for sphincs signatures
    :return: secret key and public key
    """
    sk, pk = spx_keygen()
    sk_0, pk_0 = bytes(), bytes()

    for i in sk:
        sk_0 += i
    for i in pk:
        pk_0 += i

    return sk_0, pk_0

def spx_keygen():
    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]
```

4.0.2 Signature Generation

De forma a gerar a assinatura são percorridos quatro passos.

[25]:
```
def sign( m, sk):
    """
    Sign a message with sphincs algorithm
    :param m: Message to be signed
    :param sk: Secret Key
    :return: Signature of m with sk
    """
    sk_tab = []

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

    sig_tab = spx_sign(m, sk_tab)

    sig = sig_tab[0]  # R
    for i in sig_tab[1]:  # SIG FORS
        sig += i
    for i in sig_tab[2]:  # SIG Hypertree
        sig += i

    return 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, _n)
    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 = 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
```

4.0.3 Verificar Assinatura

[26]:
```
def verify( m, sig, pk):
    """
    Check integrity of signature
    :param m: Message signed
```

```
    :param sig: Signature of m
    :param pk: Public Key
    :return: Boolean True if signature correct
    """
    pk_tab = []

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

    sig_tab = []

    sig_tab += [sig[:_n]]  # R

    sig_tab += [[]]  # SIG FORS
    for i in range(_n,
                   _n + _k * (_a + 1) * _n,
                   _n):
        sig_tab[1].append(sig[i:(i + _n)])

    sig_tab += [[]]  # SIG Hypertree
    for i in range(_n + _k * (_a + 1) * _n,
                   _n + _k * (_a + 1) * _n + (_h + _d * _len_0) * _n,
                   _n):
        sig_tab[2].append(sig[i:(i + _n)])

    return spx_verify(m, sig_tab, pk_tab)

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 = 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)
```

5 TESTE

[29]:
```
set_W(4)

sk, pk = gerar_chaves()

mensagem = b'EstruturasCriptograficas'
assinatura = sign(mensagem, sk)

resultado = verify(mensagem, assinatura, pk)
print(resultado)
```

```
True
```