sphincs

June 23, 2020

1 SPHINCS+

1.1 Parâmetros

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

[1]:
```
import math
import hashlib
import random
import os
```

[2]:
```
def aux_bytes(value, length):
    result = []

    for i in range(0, length):
        result.append(value >> (i * 8) & 0xff)

    result.reverse()
    return bytes(result)
```

1.1.1 Classe ADRS (Hash Function Address Scheme)

[3]:
```
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 = aux_bytes(self.layer, 4)
        adrs +=aux_bytes(self.tree_address, 12)
        adrs +=aux_bytes(self.type, 4)
        adrs +=aux_bytes(self.word_1, 4)
        adrs +=aux_bytes(self.word_2, 4)
        adrs +=aux_bytes(self.word_3, 4)

        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
```

1.1.2 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*.

[4]:
```
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
```

1.1.3 Inicialização dos Parâmetros

[5]:
```
_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
```

1.2 Funções auxiliares

[6]:
```
def set_security( val):
    _n = val
    calculate_variables()

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

def get_security():
    return _n

def set_winternitz( 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
```

1.3 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+.

[7]:
```
def hash(seed, adrs: ADRS, value, digest_size):
    m = hashlib.sha256()

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

    hashed = m.digest()[:digest_size]

    return hashed

'''
PRF pseudorandom key generation
'''
def prf(secret_seed, adrs, digest_size):
    random.seed(int.from_bytes(secret_seed + adrs.to_bin(), "big"))
    return aux_bytes(random.randint(0, 256 ** digest_size - 1), digest_size)

'''
To compress the message to be signed, SPHINCS+
uses an additional keyed hash function Hmsg
that can process arbitrary length messages:
'''
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

'''
PRFmsg generate randomness for
the message compression
'''
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 aux_bytes(random.randint(0, 256 ** digest_size - 1), digest_size)

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

2 WOTS+

-

2.0.1 Parâmetros

O WOTS+ utiliza os parâmetros n e w que contém valores positivos, sendo que o **n** representa o tamanho da mensagem e da chave privada, pública ou do elemento da assinatura em bytes, enquanto que o **w** é um elemento do set $\{4, 16, 256\}$.

[8]:
```
# 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
```

[9]:
```
# 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(), _n))
    return sk
```

[10]:
```
# 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):

    # 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
```

[11]:
```
# 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
    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 = aux_bytes(csum, math.ceil((_len_2 * math.floor(math.log(_w, 2))) /␣
↪8))
    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
```

```
[12]: def wots_pk_from_sig( sig, m, public_seed, adrs: ADRS): #obter public key a␣
↪partir da assinatura
    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 = aux_bytes(csum, math.ceil((_len_2 * math.floor(math.log(_w, 2))) /␣
↪8))
    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
```

3 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)**.

3.1 XMSS (Tamanho de Input Fixo)

O $XMSS$ é um método para assinar um número fixo de mensagens baseado no esquema de assinaturas Merkle .

[13]:
```
# 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:
```

```python
                break

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

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

[14]:

```python
# 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
```

[15]:

```python
# 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):
    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
```

[16]:

```python
# 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)

    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)
    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
```

3.2 Hypertree HT

A Hypertree HT é uma variante do XMSS e serve como uma árvore de certificação de instâncias XMSS.

[17]:
```
# 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
```

[18]:
```
# 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()
    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
```

[19]:
```
# 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
```

4 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.

[20]:
```
# 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(), _n)

    return sk
```

[21]:
```
# 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(), _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']
```

[22]:
```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: FORS public key PK
```

```python
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
```

[23]:
```python
# 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):
    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

    return sig_fors
```

[24]:
```python
# 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, _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())

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

[25]:
```python
# UTILS
# =================================================

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 Key Generation

[26]:
```python
def generate_key_pair():
    """
    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

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

```python
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

[27]:
```python
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

# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS+ signature SIG
def spx_sign( m, secret_key):

    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 = aux_bytes(md_int, math.ceil(_k * _a / 8))

    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 Signature Verification

[28]:
```python
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)

# 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 = md_int.to_bytes(math.ceil(_k * _a / 8), 'big')
    md = aux_bytes(md_int, math.ceil(_k * _a / 8))

    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)
```

4.0.4 Teste

[29]:

```
set_winternitz(4)

sk, pk = generate_key_pair()

m = b'Vamos decifrar esta mensagem'
print(m)

signature = sign(m, sk)
#print(signature)

result = verify(m, signature, pk)
print(result)
```

```
b'Vamos decifrar esta mensagem'
True
```

[ ]:

```

```