```
import os
import math
import random
import hashlib
from adrs import ADRS


# TWEAKABLES & UTILS
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


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)


# 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):
    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



class Sphincs():
    def __init__(self):
        self._randomize = True
        self._n = 32
        self._w = 16
        self._h = 64
        self._d = 8
        self._k = 22
        self._a = 14
        self._len_1 = math.ceil(8 * self._n / math.log(self._w, 2))
        self._len_2 = math.floor(math.log(self._len_1 * (self._w - 1), 2) /
math.log(self._w, 2)) + 1
        self._len_0 = self._len_1 + self._len_2
        self._h_prime = self._h // self._d
        self._t = 2 ** self._a

    def calculate_variables(self):
        self._len_1 = math.ceil(8 * self._n / math.log(self._w, 2))
        self._len_2 = math.floor(math.log(self._len_1 * (self._w - 1), 2) /
math.log(self._w, 2)) + 1
        self._len_0 = self._len_1 + self._len_2
        self._h_prime = self._h // self._d
        self._t = 2 ** self._a

    # SPHINCS IMPLEMENTATION
    # ================================================
    def spx_keygen(self):
        # Gerar as seeds aleatorias
        secret_seed = os.urandom(self._n)
        secret_prf = os.urandom(self._n)
        public_seed = os.urandom(self._n)

        # Compute gerador do root que chama hipertree xmss depois wots+
        public_root = self.ht_pk_gen(secret_seed, public_seed)

        # pack das chhaves
        return secret_seed + secret_prf + public_seed + public_root , public_seed +
public_root


    def spx_sign(self, m, secret_key):
        sk_tab = []
        for i in range(0, 4):
            sk_tab.append(secret_key[(i * self._n):((i + 1) * self._n)])
        adrs = ADRS()
        # Separar a sk
        secret_seed = sk_tab[0]
        secret_prf = sk_tab[1]
        public_seed = sk_tab[2]
        public_root = sk_tab[3]

        # Generate r
        opt = bytes(self._n)
        if self._randomize:
            opt = os.urandom(self._n)
```

```
        r = prf_msg(secret_prf, opt, m, self._n)
        # inicializa sign with r
        sig = r

        # Compute valores dois tamanhos
        size_md = math.floor((self._k * self._a + 7) / 8)
        size_idx_tree = math.floor((self._h - self._h // self._d + 7) / 8)
        size_idx_leaf = math.floor((self._h // self._d + 7) / 8)

        digest = hash_msg(r, public_seed, public_root, m, size_md + size_idx_tree +
size_idx_leaf)
        tmp_md = digest[:size_md]
        tmp_idx_tree = digest[size_md:(size_md + size_idx_tree)]
        tmp_idx_leaf = digest[(size_md + size_idx_tree):len(digest)]

        md_int = int.from_bytes(tmp_md, 'big') >> (len(tmp_md) * 8 - self._k * self._a)
        md = md_int.to_bytes(math.ceil(self._k * self._a / 8), 'big')

        idx_tree = int.from_bytes(tmp_idx_tree, 'big') >> (len(tmp_idx_tree) * 8 -
(self._h - self._h // self._d))
        idx_leaf = int.from_bytes(tmp_idx_leaf, 'big') >> (len(tmp_idx_leaf) * 8 -
(self._h // self._d))

        # definir adrs
        adrs.set_layer_address(0)
        adrs.set_tree_address(idx_tree)
        adrs.set_type(ADRS.FORS_TREE)
        adrs.set_key_pair_address(idx_leaf)

        # gerar assinatura com fors
        sig_fors = self.fors_sign(md, secret_seed, public_seed, adrs.copy())
        for s in sig_fors:
            sig += s
        # gerar a chave publica fors
        pk_fors = self.fors_pk_from_sig(sig_fors, md, public_seed, adrs.copy())
        # carregar a hiper arvore
        adrs.set_type(ADRS.TREE)
        # gerar assinatura com hiper arvore
        sig_ht = self.ht_sign(pk_fors, secret_seed, public_seed, idx_tree, idx_leaf)
        for s in sig_ht:
            sig += s
        # Assinatura final com concatenação de tudo
        return sig

    def spx_verify(self, m, sign, pk):
        # descomprimir a assinatura
        r = sign[:self._n]
        sig_fors = []
        sig_ht = []
        for i in range(self._n,
                       self._n + self._k * (self._a + 1) * self._n,
                       self._n):
            sig_fors.append(sign[i:(i + self._n)])
        for i in range(self._n + self._k * (self._a + 1) * self._n,
                       self._n + self._k * (self._a + 1) * self._n + (self._h + self._d *
self._len_0) * self._n,
                       self._n):
            sig_ht.append(sign[i:(i + self._n)])
        adrs = ADRS()

        # extarir os valores da pk
        public_seed = pk[0:self._n]
        public_root = pk[self._n:self._n*2]

        size_md = math.floor((self._k * self._a + 7) / 8)
        size_idx_tree = math.floor((self._h - self._h // self._d + 7) / 8)
```

```
        size_idx_leaf = math.floor((self._h // self._d + 7) / 8)

        # compute de hash msg
        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)]

        # extrair valores
        md_int = int.from_bytes(tmp_md, 'big') >> (len(tmp_md) * 8 - self._k * self._a)
        md = md_int.to_bytes(math.ceil(self._k * self._a / 8), 'big')

        idx_tree = int.from_bytes(tmp_idx_tree, 'big') >> (len(tmp_idx_tree) * 8 -
(self._h - self._h // self._d))
        idx_leaf = int.from_bytes(tmp_idx_leaf, 'big') >> (len(tmp_idx_leaf) * 8 -
(self._h // self._d))

        adrs.set_layer_address(0)
        adrs.set_tree_address(idx_tree)
        adrs.set_type(ADRS.FORS_TREE)
        adrs.set_key_pair_address(idx_leaf)

        # Gerar Chave Pública FORS a partir da Assinatura
        pk_fors = self.fors_pk_from_sig(sig_fors, md, public_seed, adrs)
        # Verificação da Árvore de Hiperárvore:
        adrs.set_type(ADRS.TREE)
        # Retorna o resultado da verificação
        return self.ht_verify(pk_fors, sig_ht, public_seed, idx_tree, idx_leaf,
public_root)



    # UTILS
    # ================================================
    def sig_wots_from_sig_xmss(self, sig):
        return sig[0:self._len_0]

    def auth_from_sig_xmss(self, sig):
        return sig[self._len_0:]

    def sigs_xmss_from_sig_ht(self, sig):
        sigs = []
        for i in range(0, self._d):
            sigs.append(sig[i * (self._h_prime + self._len_0):(i + 1) * (self._h_prime +
self._len_0)])
        return sigs

    def auths_from_sig_fors(self, sig):
        sigs = []
        for i in range(0, self._k):
            sigs.append([])
            sigs[i].append(sig[(self._a + 1) * i])
            sigs[i].append(sig[((self._a + 1) * i + 1):((self._a + 1) * (i + 1))])
        return sigs

    # WOTS+
    # ================================================
    # 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(self, x, i, s, public_seed, adrs: ADRS):
        if s == 0:
            return bytes(x)
        if (i + s) > (self._w - 1):
            return -1
```

```
        tmp = self.chain(x, i, s - 1, public_seed, adrs)
        adrs.set_hash_address(i + s - 1)
        tmp = hash(public_seed, adrs, tmp, self._n)
        return tmp

    # Input: secret seed SK.seed, address ADRS
    # Output: WOTS+ private key sk
    def wots_sk_gen(self, secret_seed, adrs: ADRS):  # Not necessary
        sk = []
        for i in range(0, self._len_0):
            adrs.set_chain_address(i)
            adrs.set_hash_address(0)
            sk.append(prf(secret_seed, adrs.copy(), self._n))
        return sk

    # Input: secret seed SK.seed, address ADRS, public seed PK.seed
    # Output: WOTS+ public key pk
    def wots_pk_gen(self, secret_seed, public_seed, adrs: ADRS):
        wots_pk_adrs = adrs.copy()
        tmp = bytes()
        for i in range(0, self._len_0):
            adrs.set_chain_address(i)
            adrs.set_hash_address(0)
            sk = prf(secret_seed, adrs.copy(), self._n)
            tmp += bytes(self.chain(sk, 0, self._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, self._n)
        return pk

    # Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
    # Output: WOTS+ signature sig
    def wots_sign(self, m, secret_seed, public_seed, adrs):
        csum = 0
        msg = base_w(m, self._w, self._len_1)
        for i in range(0, self._len_1):
            csum += self._w - 1 - msg[i]
        padding = (self._len_2 * math.floor(math.log(self._w, 2))) % 8 if (self._len_2 *
math.floor(math.log(self._w, 2))) % 8 != 0 else 8
        csum = csum << (8 - padding)
        csumb = csum.to_bytes(math.ceil((self._len_2 * math.floor(math.log(self._w,
2))) / 8), byteorder='big')
        csumw = base_w(csumb, self._w, self._len_2)
        msg += csumw
        sig = []
        for i in range(0, self._len_0):
            adrs.set_chain_address(i)
            adrs.set_hash_address(0)
            sk = prf(secret_seed, adrs.copy(), self._n)
            sig += [self.chain(sk, 0, msg[i], public_seed, adrs.copy())]
        return sig

    def wots_pk_from_sig(self, sig, m, public_seed, adrs: ADRS):
        csum = 0
        wots_pk_adrs = adrs.copy()
        msg = base_w(m, self._w, self._len_1)
        for i in range(0, self._len_1):
            csum += self._w - 1 - msg[i]
        padding = (self._len_2 * math.floor(math.log(self._w, 2))) % 8 if (self._len_2 *
math.floor(math.log(self._w, 2))) % 8 != 0 else 8
        csum = csum << (8 - padding)
        csumb = csum.to_bytes(math.ceil((self._len_2 * math.floor(math.log(self._w,
2))) / 8), byteorder='big')
        csumw = base_w(csumb, self._w, self._len_2)
        msg += csumw
        tmp = bytes()
```

```
        for i in range(0, self._len_0):
            adrs.set_chain_address(i)
            tmp += self.chain(sig[i], msg[i], self._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, self._n)
        return pk_sig

    # XMSS
    # ================================================

    # 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(self, 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 = self.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,
self._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: Secret seed SK.seed, public seed PK.seed, address ADRS
    # Output: XMSS public key PK
    def xmss_pk_gen(self, secret_seed, public_key, adrs: ADRS):
        pk = self.treehash(secret_seed, 0, self._h_prime, public_key, adrs.copy())
        return pk

    # 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(self, m, secret_seed, idx, public_seed, adrs):
        auth = []
        for j in range(0, self._h_prime):
            ki = math.floor(idx // 2 ** j)
            if ki % 2 == 1:  # XORING idx/ 2**j with 1
                ki -= 1
            else:
                ki += 1
            auth += [self.treehash(secret_seed, ki * 2 ** j, j, public_seed, adrs.copy())]
        adrs.set_type(ADRS.WOTS_HASH)
        adrs.set_key_pair_address(idx)
        sig = self.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(self, idx, sig_xmss, m, public_seed, adrs):
```

```
        adrs.set_type(ADRS.WOTS_HASH)
        adrs.set_key_pair_address(idx)
        sig = self.sig_wots_from_sig_xmss(sig_xmss)
        auth = self.auth_from_sig_xmss(sig_xmss)
        node_0 = self.wots_pk_from_sig(sig, m, public_seed, adrs.copy())
        node_1 = 0
        adrs.set_type(ADRS.TREE)
        adrs.set_tree_index(idx)
        for i in range(0, self._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], self._n)
            else:
                adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                node_1 = hash(public_seed, adrs.copy(), auth[i] + node_0, self._n)
            node_0 = node_1
        return node_0

    # HYPERTREE XMSS
    # ================================================
    # Input: Private seed SK.seed, public seed PK.seed
    # Output: HT public key PK_HT
    def ht_pk_gen(self, secret_seed, public_seed):
        adrs = ADRS()
        adrs.set_layer_address(self._d - 1)
        adrs.set_tree_address(0)
        root = self.xmss_pk_gen(secret_seed, public_seed, adrs.copy())
        return root

    # 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(self, m, secret_seed, public_seed, idx_tree, idx_leaf):
        adrs = ADRS()
        adrs.set_layer_address(0)
        adrs.set_tree_address(idx_tree)
        sig_tmp = self.xmss_sign(m, secret_seed, idx_leaf, public_seed, adrs.copy())
        sig_ht = sig_tmp
        root = self.xmss_pk_from_sig(idx_leaf, sig_tmp, m, public_seed, adrs.copy())
        for j in range(1, self._d):
            idx_leaf = idx_tree % 2 ** self._h_prime
            idx_tree = idx_tree >> self._h_prime
            adrs.set_layer_address(j)
            adrs.set_tree_address(idx_tree)
            sig_tmp = self.xmss_sign(root, secret_seed, idx_leaf, public_seed,
adrs.copy())
            sig_ht = sig_ht + sig_tmp
            if j < self._d - 1:
                root = self.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(self, m, sig_ht, public_seed, idx_tree, idx_leaf, public_key_ht):
        adrs = ADRS()
        sigs_xmss = self.sigs_xmss_from_sig_ht(sig_ht)
        sig_tmp = sigs_xmss[0]
        adrs.set_layer_address(0)
        adrs.set_tree_address(idx_tree)
        node = self.xmss_pk_from_sig(idx_leaf, sig_tmp, m, public_seed, adrs)
        for j in range(1, self._d):
            idx_leaf = idx_tree % 2 ** self._h_prime
            idx_tree = idx_tree >> self._h_prime
```

```
            sig_tmp = sigs_xmss[j]
            adrs.set_layer_address(j)
            adrs.set_tree_address(idx_tree)
            node = self.xmss_pk_from_sig(idx_leaf, sig_tmp, node, public_seed, adrs)
        if node == public_key_ht:
            return True
        else:
            return False

    # FORS
    # ================================================

    # Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
    # Output: FORS private key sk
    def fors_sk_gen(self, secret_seed, adrs: ADRS, idx):
        adrs.set_tree_height(0)
        adrs.set_tree_index(idx)
        sk = prf(secret_seed, adrs.copy(), self._n)
        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_treehash(self, 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(), self._n)
            node = hash(public_seed, adrs.copy(), sk, self._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,
self._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: Secret seed SK.seed, public seed PK.seed, address ADRS
    # Output: FORS public key PK
    def fors_pk_gen(self, secret_seed, public_seed, adrs: ADRS):
        fors_pk_adrs = adrs.copy()
        root = bytes()
        for i in range(0, self._k):
            root += self.fors_treehash(secret_seed, i * self._t, self._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, self._n)
        return pk

    # Input: Bit string M, secret seed SK.seed, address ADRS, public seed PK.seed
    # Output: FORS signature SIG_FORS
    def fors_sign(self, m, secret_seed, public_seed, adrs):
        m_int = int.from_bytes(m, 'big')
        sig_fors = []
        for i in range(0, self._k):
            idx = (m_int >> (self._k - 1 - i) * self._a) % self._t
            adrs.set_tree_height(0)
```

```
            adrs.set_tree_index(i * self._t + idx)
            sig_fors += [prf(secret_seed, adrs.copy(), self._n)]
            auth = []
            for j in range(0, self._a):
                s = math.floor(idx // 2 ** j)
                if s % 2 == 1:  # XORING idx/ 2**j with 1
                    s -= 1
                else:
                    s += 1
                auth += [self.fors_treehash(secret_seed, i * self._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(self, sig_fors, m, public_seed, adrs: ADRS):
        m_int = int.from_bytes(m, 'big')
        sigs = self.auths_from_sig_fors(sig_fors)
        root = bytes()
        for i in range(0, self._k):
            idx = (m_int >> (self._k - 1 - i) * self._a) % self._t
            sk = sigs[i][0]
            adrs.set_tree_height(0)
            adrs.set_tree_index(i * self._t + idx)
            node_0 = hash(public_seed, adrs.copy(), sk, self._n)
            node_1 = 0
            auth = sigs[i][1]
            adrs.set_tree_index(i * self._t + idx)  # Really Useful?
            for j in range(0, self._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], self._n)
                else:
                    adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                    node_1 = hash(public_seed, adrs.copy(), auth[j] + node_0, self._n)
                node_0 = node_1
            root += node_0
        fors_pk_adrs = adrs.copy()
        fors_pk_adrs.set_type(ADRS.FORS_ROOTS)
        fors_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())
        pk = hash(public_seed, fors_pk_adrs, root, self._n)
        return pk

sphincs = Sphincs()
```