Estruturas Criptográficas

Trabalho Prático 4 - Exercício 2

José de Matos Moreira - PG53963

Pedro Freitas - PG52700

Enunciado do problema

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

Resolução

Em primeiro lugar, mostram-se os diversos *imports* efetuados.

In [22]:

```
from Crypto.Hash import SHA256, SHA512, SHAKE256
import os
```

Código auxiliar

Em primeiro lugar, apresenta-se o código relativo à classe **ADRS**. Um **ADRS** consiste em valores públicos que indicam a posição do valor a ser utilizado pela função. Deste modo, todas as funções presentes nessa mesma classe permitem manipular um endereço, sendo, portanto, fundamentais naquilo que é o desenvolvimento do protótipo pretendido.

In [23]:

```
class ADRS:

    WOTS_HASH   = 0
    WOTS_PK     = 1
    TREE        = 2
    FORS_TREE   = 3
    FORS_ROOTS  = 4
    WOTS_PRF    = 5
    FORS_PRF    = 6


    def __init__(self, a=32):
        self.a = bytearray(a)
    def copy(self):
        return ADRS(self.a)


    def set_layer_address(self, x):
        self.a[0 : 4] = x.to_bytes(4, byteorder='big')


    def set_tree_address(self, x):
        self.a[4 : 16] = x.to_bytes(12, byteorder='big')


    def set_key_pair_address(self, x):
        self.a[20 : 24] = x.to_bytes(4, byteorder='big')

    def get_key_pair_address(self):
        return int.from_bytes(self.a[20 : 24], byteorder='big')


    def set_tree_height(self, x):
        self.a[24 : 28] = x.to_bytes(4, byteorder='big')


    def set_chain_address(self, x):
        self.a[24 : 28] = x.to_bytes(4, byteorder='big')


    def set_tree_index(self, x):
        self.a[28 : 32] = x.to_bytes(4, byteorder='big')


    def get_tree_index(self):
        return int.from_bytes(self.a[28 : 32], byteorder='big')


    def set_hash_address(self, x):
        self.a[28 : 32] = x.to_bytes(4, byteorder='big')


    def set_type_and_clear(self, t):
        self.a[16 : 20] = t.to_bytes(4, byteorder='big')
        for i in range(12):
            self.a[20 + i] = 0


    def adrs(self):
        return self.a


    def adrsc(self):
        return self.a[3 : 4] + self.a[8 : 16] + self.a[19 : 20] + self.a[20:
```

Algoritmos

Agora, apresentam-se os diversos algoritmos (e outras funções úteis) que constituem o corpo do protótipo apresentado no **FIPS 205**. Adianta-se que todos os algoritmos referidos pertencem à classe **SLHDSA** e que a inicialização da mesma se efetua tendo em conta os parâmetros estabelecidos. Apesar da classe possuir diversas outras funções (algoritmos de *hash*, autenticação, etc.), passam-se a explicar apenas os algoritmos que constituem o mecanismo principal:

- **to_int**: função que converte uma *byte string* num inteiro
- **to_byte**: função que converte um inteiro numa *byte string*
- **base_2b**: algoritmo que computa a representação em base 2^b de uma *byte string*
- **chain**: função de *chain* usada no **WOTS+**
- **wots_pkgen**: algoritmo que gera uma chave pública **WOTS+**
- **wots_sign**: algoritmo que gera uma assinatura **WOTS+**, através de uma mensagem, em *bytes*
- **wots_pk_from_sig**: algoritmo que calcula uma chave pública **WOTS+** através de uma mensagem e da sua assinatura
- **xmss_node**: função que calcula a raiz de uma subárvore **Merkle** de chaves públicas **WOTS+**
- **xmss_sign**: algoritmo que gera uma assinatura **XMSS**
- **xmss_pk_from_sig**: algoritmo que computa uma chave pública **XMSS**, através de uma assinatura **XMSS**
- **ht_sign**: algoritmo que gera uma *hypertree signature*
- **ht_verify**: função que verifica uma *hypertree signature*
- **fors_sk_gen**: função que gera *byte strings* da chave privada **FORS**
- **fors_node**: função que calcula a raiz de uma subárvore **Merkle** de valores públicos **FORS**
- **fors_sign**: algoritmo que gera uma assinatura **FORS**
- **fors_pk_from_sig**: algoritmo que computa uma chave pública **FORS** através de uma assinatura **FORS**
- **slh_keygen**: função que gera um par de chaves **SLH-DSA**, privada e pública
- **slh_sign**: algoritmo que gera uma assinatura **SLH-DSA**
- **slh_verify**: algoritmo que verifica uma assinatura **SLH-DSA**

In [24]:

```python
class SLHDSA:

    def __init__(self, hashname, paramid, n, h, d, hp, a, k, lg_w, m, rbg=os
        self.hashname = hashname
        self.paramid = paramid
        self.n = n
        self.h = h
        self.d = d
        self.hp = hp
        self.a = a
        self.k = k
        self.lg_w = lg_w
        self.m = m
        self.rbg = rbg
        self.algname = 'SPHINCS+'
        self.stdname = f'SLH-DSA-{self.hashname}-{8 * self.n}{self.paramid}'

        if hashname == 'SHAKE':
            self.h_msg = self.shake_h_msg
            self.prf = self.shake_prf
            self.prf_msg = self.shake_prf_msg
            self.h_f = self.shake_f
            self.h_h = self.shake_f
            self.h_t = self.shake_f

        elif hashname == 'SHA2' and self.n == 16:
            self.h_msg = self.sha256_h_msg
            self.prf = self.sha256_prf
            self.prf_msg = self.sha256_prf_msg
            self.h_f = self.sha256_f
            self.h_h = self.sha256_f
            self.h_t = self.sha256_f

        elif hashname == 'SHA2' and self.n > 16:
            self.h_msg = self.sha512_h_msg
            self.prf = self.sha256_prf
            self.prf_msg = self.sha512_prf_msg
            self.h_f = self.sha256_f
            self.h_h = self.sha512_h
            self.h_t = self.sha512_h

        self.w = 2 ** self.lg_w
        self.len1 = (8 * self.n + (self.lg_w - 1)) // self.lg_w
        self.len2 = (self.len1 * (self.w - 1)).bit_length() // self.lg_w + 1
        self.len = self.len1 + self.len2
        self.pk_sz = 2 * self.n
        self.sk_sz = 4 * self.n
        self.sig_sz = (1 + self.k*(1 + self.a) + self.h + self.d * self.len)


    def shake256(self, x, l):
        return SHAKE256.new(x).read(l)


    def shake_h_msg(self, r, pk_seed, pk_root, m):
        return self.shake256(r + pk_seed + pk_root + m, self.m)


    def shake_prf(self, pk_seed, sk_seed, adrs):
        return self.shake256(pk_seed + adrs.adrs() + sk_seed, self.n)


    def shake_prf_msg(self, sk_prf, opt_rand, m):
        return self.shake256(sk_prf + opt_rand + m, self.n)
    def shake_f(self, pk_seed, adrs, m1):
        return self.shake256(pk_seed + adrs.adrs() + m1, self.n)


    def sha256(self, x, n=32):
        return SHA256.new(x).digest()[0 : n]


    def sha512(self, x, n=64):
        return SHA512.new(x).digest()[0 : n]


    def mgf(self, hash_f, hash_l, mgf_seed, mask_len):
        t = b''

        for c in range((mask_len + hash_l - 1) // hash_l):
            t += hash_f(mgf_seed + c.to_bytes(4, byteorder='big'))

        return t[0 : mask_len]


    def mgf_sha256(self, mgf_seed, mask_len):
        return self.mgf(self.sha256, 32, mgf_seed, mask_len)


    def mgf_sha512(self, mgf_seed, mask_len):
        return self.mgf(self.sha512, 64, mgf_seed, mask_len)


    def hmac(self, hash_f, hash_l, hash_b, k, text):
        if len(k) > hash_b:
            k = hash_f(k)

        ipad = bytearray(hash_b)
        ipad[0 : len(k)] = k
        opad = bytearray(ipad)

        for i in range(hash_b):
            ipad[i] ^= 0x36
            opad[i] ^= 0x5C

        return hash_f(opad + hash_f(ipad + text))


    def hmac_sha256(self, k, text, n=32):
        return self.hmac(self.sha256, 32, 64, k, text)[0 : n]


    def hmac_sha512(self, k, text, n=64):
        return self.hmac(self.sha512, 64, 128, k, text)[0 : n]


    def sha256_h_msg(self, r, pk_seed, pk_root, m):
        return self.mgf_sha256(r + pk_seed + self.sha256(r + pk_seed + pk_ro
    def sha256_prf(self, pk_seed, sk_seed, adrs):
        return self.sha256(pk_seed + bytes(64 - self.n) + adrs.adrsc() + sk_


    def sha256_prf_msg(self, sk_prf, opt_rand, m):
        return self.hmac_sha256(sk_prf, opt_rand + m, self.n)


    def sha256_f(self, pk_seed, adrs, m1):
        return self.sha256(pk_seed + bytes(64 - self.n) + adrs.adrsc() + m1,


    def sha512_h_msg(self, r, pk_seed, pk_root, m):
        return self.mgf_sha512( r + pk_seed + self.sha512(r + pk_seed + pk_r


    def sha512_prf_msg(self, sk_prf, opt_rand, m):
        return self.hmac_sha512(sk_prf, opt_rand + m, self.n)


    def sha512_h(self, pk_seed, adrs, m2):
        return self.sha512(pk_seed + bytes(128 - self.n) + adrs.adrsc() + m2


    def split_digest(self, digest):
        ka1 = (self.k * self.a + 7) // 8
        md = digest[0 : ka1]
        hd = self.h // self.d
        hhd = self.h - hd
        ka2 = ka1 + ((hhd + 7) // 8)
        i_tree = self.to_int(digest[ka1 : ka2], (hhd + 7) // 8) % (2 ** hhd)
        ka3 = ka2 + ((hd + 7) // 8)
        i_leaf = self.to_int(digest[ka2 : ka3], (hd + 7) // 8) % (2 ** hd)

        return md, i_tree, i_leaf


    def to_int(self, X, n):
        total = 0

        for i in range(n):
            total = (total << 8) + int(X[i])

        return total


    def to_byte(self, x, n):
        total = x
        S = bytearray(n)

        for i in range(n):
            S[n - 1 - i] = total & 0xFF
            total >>= 8

        return S
    def base_2b(self, X, b, out_len):
        i = 0
        bits = 0
        total = 0
        baseb = []
        m = (1 << b) - 1

        for _ in range(out_len):
            while bits < b:
                total = (total << 8) + int(X[i])
                i += 1
                bits += 8
            bits -= b
            baseb += [(total >> bits) & m]

        return baseb


    def chain(self, X, i, s, PK_seed, ADRS):
        if i + s >= self.w:
            return None

        tmp = X
        for j in range(i, i + s):
            ADRS.set_hash_address(j)
            tmp = self.h_f(PK_seed, ADRS, tmp)

        return tmp


    def wots_pkgen(self, SK_seed, PK_seed, adrs):
        skADRS = adrs.copy()
        skADRS.set_type_and_clear(ADRS.WOTS_PRF)
        skADRS.set_key_pair_address(adrs.get_key_pair_address())

        tmp = b''
        for i in range(self.len):
            skADRS.set_chain_address(i)
            sk = self.prf(PK_seed, SK_seed, skADRS)
            adrs.set_chain_address(i)
            tmp += self.chain(sk, 0, self.w - 1, PK_seed, adrs)

        wotspkADRS = adrs.copy()
        wotspkADRS.set_type_and_clear(ADRS.WOTS_PK)
        wotspkADRS.set_key_pair_address(adrs.get_key_pair_address())
        pk = self.h_t(PK_seed, wotspkADRS, tmp)

        return pk


    def wots_sign(self, m, SKseed, PKseed, adrs):
        csum =   0
        msg = self.base_2b(m, self.lg_w, self.len1)

        for i in range(self.len1):
            csum += self.w - 1 - msg[i]

        csum <<= ((8 - ((self.len2 * self.lg_w) % 8)) % 8)
        msg += self.base_2b(self.to_byte(csum, (self.len2 * self.lg_w + 7) /

        skADRS = adrs.copy()
        skADRS.set_type_and_clear(ADRS.WOTS_PRF)
        skADRS.set_key_pair_address(adrs.get_key_pair_address())

        sig = b''
        for i in range(self.len):
            skADRS.set_chain_address(i)
            sk = self.prf(PKseed, SKseed, skADRS)
            adrs.set_chain_address(i)
            sig += self.chain(sk, 0, msg[i], PKseed, adrs)

        return sig


    def wots_pk_from_sig(self, sig, m, PKseed, adrs):
        csum = 0
        msg = self.base_2b(m, self.lg_w, self.len1)

        for i in range(self.len1):
            csum += self.w - 1 - msg[i]

        csum <<= ((8 - ((self.len2 * self.lg_w) % 8)) % 8)
        msg +=  self.base_2b(self.to_byte(csum, (self.len2 * self.lg_w + 7)

        tmp = b''
        for i in range(self.len):
            adrs.set_chain_address(i)
            tmp +=  self.chain(sig[i*self.n:(i+1)*self.n], msg[i], self.w -

        wotspkADRS = adrs.copy()
        wotspkADRS.set_type_and_clear(ADRS.WOTS_PK)
        wotspkADRS.set_key_pair_address(adrs.get_key_pair_address())

        pksig = self.h_t(PKseed, wotspkADRS, tmp)

        return  pksig


    def xmss_node(self, SKseed, i, z, PKseed, adrs):
        if z > self.hp or i >= 2 ** (self.hp - z):
            return None

        if z == 0:
            adrs.set_type_and_clear(ADRS.WOTS_HASH)
            adrs.set_key_pair_address(i)
            node = self.wots_pkgen(SKseed, PKseed, adrs)

        else:
            lnode = self.xmss_node(SKseed, 2 * i, z - 1, PKseed, adrs)
            rnode = self.xmss_node(SKseed, 2 * i + 1, z - 1, PKseed, adrs)
            adrs.set_type_and_clear(ADRS.TREE)
            adrs.set_tree_height(z)
            adrs.set_tree_index(i)

            node = self.h_h(PKseed, adrs, lnode + rnode)

        return node


    def xmss_sign(self, m, SKseed, idx, PKseed, adrs):
        auth = b''
        for j in range(self.hp):
            k = (idx >> j) ^ 1
            auth += self.xmss_node(SKseed, k, j, PKseed, adrs)

        adrs.set_type_and_clear(ADRS.WOTS_HASH)
        adrs.set_key_pair_address(idx)

        sig = self.wots_sign(m, SKseed, PKseed, adrs)
        SIGxmss = sig + auth

        return SIGxmss


    def xmss_pk_from_sig(self, idx, SIGxmss, m, PKseed, adrs):
        adrs.set_type_and_clear(ADRS.WOTS_HASH)
        adrs.set_key_pair_address(idx)

        sig = SIGxmss[0 : self.len * self.n]
        AUTH = SIGxmss[self.len * self.n:]
        node_0 = self.wots_pk_from_sig(sig, m, PKseed, adrs)

        adrs.set_type_and_clear(ADRS.TREE)
        adrs.set_tree_index(idx)

        for k in range(self.hp):
            adrs.set_tree_height(k + 1)
            auth_k = AUTH[k * self.n : (k + 1) * self.n]

            if (idx >> k) & 1 == 0:
                adrs.set_tree_index(adrs.get_tree_index() // 2)
                node_1 = self.h_h(PKseed, adrs, node_0 + auth_k)

            else:
                adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                node_1 = self.h_h(PKseed, adrs, auth_k + node_0)
            node_0 = node_1

        return node_0


    def ht_sign(self, m, SKseed, PKseed, i_tree, i_leaf):
        adrs = ADRS()
        adrs.set_tree_address(i_tree)

        SIGtmp = self.xmss_sign(m, SKseed, i_leaf, PKseed, adrs)
        SIGht  = SIGtmp
        root = self.xmss_pk_from_sig(i_leaf, SIGtmp, m, PKseed, adrs)
        hp_m = ((1 << self.hp) - 1)

        for j in range(1, self.d):
            i_leaf = i_tree & hp_m
            i_tree = i_tree >> self.hp

            adrs.set_layer_address(j)
            adrs.set_tree_address(i_tree)

            SIGtmp = self.xmss_sign(root, SKseed, i_leaf, PKseed, adrs)
            SIGht += SIGtmp

            if j < self.d - 1:
                root = self.xmss_pk_from_sig(i_leaf, SIGtmp, root, PKseed, a

        return SIGht


    def ht_verify(self, m, SIGht, PKseed, i_tree, i_leaf, PKroot):
        adrs = ADRS()
        adrs.set_tree_address(i_tree)

        sig_tmp = SIGht[0 : (self.hp + self.len) * self.n]
        node = self.xmss_pk_from_sig(i_leaf, sig_tmp, m, PKseed, adrs)

        hp_m = ((1 << self.hp) - 1)
        for j in range(1, self.d):
            i_leaf = i_tree & hp_m
            i_tree = i_tree >> self.hp

            adrs.set_layer_address(j)
            adrs.set_tree_address(i_tree)

            sig_tmp = SIGht[j*(self.hp + self.len) * self.n : (j + 1) * (sel
            node = self.xmss_pk_from_sig(i_leaf, sig_tmp, node, PKseed, adrs

        return node == PKroot


    def fors_sk_gen(self, SKseed, PKseed, adrs, idx):
        sk_adrs = adrs.copy()
        sk_adrs.set_type_and_clear(ADRS.FORS_PRF)
        sk_adrs.set_key_pair_address(adrs.get_key_pair_address())
        sk_adrs.set_tree_index(idx)

        return self.prf(PKseed, SKseed, sk_adrs)


    def fors_node(self, SKseed, i, z, PKseed, adrs):
        if z > self.a or i >= (self.k << (self.a - z)):
            return None

        if z == 0:
            sk = self.fors_sk_gen(SKseed, PKseed, adrs, i)
            adrs.set_tree_height(0)
            adrs.set_tree_index(i)
            node = self.h_f(PKseed, adrs, sk)

        else:
            lnode = self.fors_node(SKseed, 2 * i, z - 1, PKseed, adrs)
            rnode = self.fors_node(SKseed, 2 * i + 1, z - 1, PKseed, adrs)
            adrs.set_tree_height(z)
            adrs.set_tree_index(i)
            node = self.h_h(PKseed, adrs, lnode + rnode)

        return node


    def fors_sign(self, md, SKseed, PKseed, adrs):
        sig_fors = b''
        indices = self.base_2b(md, self.a, self.k)

        for i in range(self.k):
            sig_fors += self.fors_sk_gen(SKseed, PKseed, adrs, (i << self.a)

            for j in range(self.a):
                s = (indices[i] >> j) ^ 1
                sig_fors += self.fors_node(SKseed, (i << (self.a - j)) + s,

        return sig_fors


    def fors_pk_from_sig(self, SIGfors, md, PKseed, adrs):

        def get_sk(sig_fors, i):
            return sig_fors[i * (self.a + 1) * self.n : (i * (self.a + 1) +

        def get_auth(sig_fors, i):
            return sig_fors[(i * (self.a + 1) + 1) * self.n : (i + 1) * (sel

        indices = self.base_2b(md, self.a, self.k)

        root = b''
        for i in range(self.k):
            sk = get_sk(SIGfors, i)
            adrs.set_tree_height(0)
            adrs.set_tree_index((i << self.a) + indices[i])
            node_0 = self.h_f(PKseed, adrs, sk)

            auth = get_auth(SIGfors, i)
            for j in range(self.a):
                auth_j = auth[j * self.n : (j + 1) * self.n]
                adrs.set_tree_height(j + 1)

                if (indices[i] >> j) & 1 == 0:
                    adrs.set_tree_index(adrs.get_tree_index() // 2)
                    node_1 = self.h_h(PKseed, adrs, node_0 + auth_j)

                else:
                    adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                    node_1 = self.h_h(PKseed, adrs, auth_j + node_0)

                node_0 = node_1

            root += node_0

        fors_pk_adrs = adrs.copy()
        fors_pk_adrs.set_type_and_clear(ADRS.FORS_ROOTS)
        fors_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())

        pk = self.h_t(PKseed, fors_pk_adrs, root)

        return pk


    def slh_keygen(self):
        seed = self.rbg(3 * self.n)
        sk_seed = seed[0 : self.n]
        sk_prf = seed[self.n : 2 * self.n]
        pk_seed = seed[2 * self.n:]
        adrs = ADRS()
        adrs.set_layer_address(self.d - 1)
        pk_root = self.xmss_node(sk_seed, 0, self.hp, pk_seed, adrs)
        sk = sk_seed + sk_prf + pk_seed + pk_root
        pk = pk_seed + pk_root

        return sk, pk


    def slh_sign(self, m, sk, randomize=True):
        adrs = ADRS()
        sk_seed = sk[0 : self.n]
        sk_prf  = sk[self.n : 2 * self.n]
        pk_seed = sk[2 * self.n : 3 * self.n]
        pk_root = sk[3 * self.n:]

        opt_rand = pk_seed
        if randomize:
            opt_rand = self.rbg(self.n)

        r = self.prf_msg(sk_prf, opt_rand, m)
        sig = r

        digest = self.h_msg(r, pk_seed, pk_root, m)
        md, i_tree, i_leaf = self.split_digest(digest)

        adrs.set_tree_address(i_tree)
        adrs.set_type_and_clear(ADRS.FORS_TREE)
        adrs.set_key_pair_address(i_leaf)

        sig_fors = self.fors_sign(md, sk_seed, pk_seed, adrs)
        sig += sig_fors

        pk_fors = self.fors_pk_from_sig(sig_fors, md, pk_seed, adrs)
        sig_ht = self.ht_sign(pk_fors, sk_seed, pk_seed, i_tree, i_leaf)
        sig += sig_ht
        return  sig


    def slh_verify(self, m, sig, pk):
        if len(sig) != self.sig_sz or len(pk) != self.pk_sz:
            return False

        pk_seed = pk[:self.n]
        pk_root = pk[self.n:]

        adrs = ADRS()
        r = sig[0 : self.n]
        sig_fors = sig[self.n : (1 + self.k * (1 + self.a)) * self.n]
        sig_ht = sig[(1 + self.k * (1 + self.a)) * self.n:]

        digest  = self.h_msg(r, pk_seed, pk_root, m)
        (md, i_tree, i_leaf) = self.split_digest(digest)

        adrs.set_tree_address(i_tree)
        adrs.set_type_and_clear(ADRS.FORS_TREE)
        adrs.set_key_pair_address(i_leaf)

        pk_fors = self.fors_pk_from_sig(sig_fors, md, pk_seed, adrs)

        return self.ht_verify(pk_fors, sig_ht, pk_seed, i_tree, i_leaf, pk_r
```

Testes de aplicação

Para efeitos de teste, desenvolveu-se a função **slh_dsa_test**, responsável por, através dos respetivos parâmetros, dependendo do conjunto pretendido, geras as chaves, privada e pública, assinar uma mensagem e verificar a respetiva assinatura.

In [25]:

```
def slh_dsa_test(security):
    alg = None

    if security == 'SLH-DSA-SHA2-128s':
        alg = SLHDSA('SHA2', 's', 16, 63, 7, 9, 12, 14, 4, 30)

    elif security == 'SLH-DSA-SHAKE-128s':
        alg = SLHDSA('SHAKE', 's', 16, 63, 7, 9, 12, 14, 4, 30)

    elif security == 'SLH-DSA-SHA2-128f':
        alg = SLHDSA('SHA2', 'f', 16, 66, 22, 3, 6, 33, 4, 34)

    elif security == 'SLH-DSA-SHAKE-128f':
        alg = SLHDSA('SHAKE', 'f', 16, 66, 22, 3, 6, 33, 4, 34)

    elif security == 'SLH-DSA-SHA2-192s':
        alg = SLHDSA('SHA2', 's', 24, 63, 7, 9, 14, 17, 4, 39)

    elif security == 'SLH-DSA-SHAKE-192s':
        alg = SLHDSA('SHAKE', 's', 24, 63, 7, 9, 14, 17, 4, 39)

    elif security == 'SLH-DSA-SHA2-192f':
        alg = SLHDSA('SHA2', 'f', 24, 66, 22, 3, 8, 33, 4, 42)

    elif security == 'SLH-DSA-SHAKE-192f':
        alg = SLHDSA('SHAKE', 'f', 24, 66, 22, 3, 8, 33, 4, 42)

    elif security == 'SLH-DSA-SHA2-256s':
        alg = SLHDSA('SHA2', 's', 32, 64, 8, 8, 14, 22, 4, 47)

    elif security == 'SLH-DSA-SHAKE-256s':
        alg = SLHDSA('SHAKE', 's', 32, 64, 8, 8, 14, 22, 4, 47)

    elif security == 'SLH-DSA-SHA2-256f':
        alg = SLHDSA('SHA2', 'f', 32, 68, 17, 4, 9, 35, 4, 49)

    elif security == 'SLH-DSA-SHAKE-256f':
        alg = SLHDSA('SHAKE', 'f', 32, 68, 17, 4, 9, 35, 4, 49)

    else:
        print('[SLH-DSA] invalid call')
        return


    sk, pk = alg.slh_keygen()
    m = b"Messi, the GOAT!"
    sig = alg.slh_sign(m, sk)
    verify = alg.slh_verify(m, sig, pk)

    if verify == True:
        print(f'[SLH-DSA] ({security}) valid signature')

    else:
        print(f'[SLH-DSA] ({security}) invalid signature')
```

SLH-DSA-SHA2-128s

In [26]:

```
slh_dsa_test('SLH-DSA-SHA2-128s')
```

```
[SLH-DSA] (SLH-DSA-SHA2-128s) valid signature
```

SLH-DSA-SHAKE-128s

In [27]:

```
slh_dsa_test('SLH-DSA-SHAKE-128s')
```

```
[SLH-DSA] (SLH-DSA-SHAKE-128s) valid signature
```

SLH-DSA-SHA2-128f

In [28]:

```
slh_dsa_test('SLH-DSA-SHA2-128f')
```

```
[SLH-DSA] (SLH-DSA-SHA2-128f) valid signature
```

SLH-DSA-SHAKE-128f

In [29]:

```
slh_dsa_test('SLH-DSA-SHAKE-128f')
```

```
[SLH-DSA] (SLH-DSA-SHAKE-128f) valid signature
```

SLH-DSA-SHA2-192s

In [30]:

```
slh_dsa_test('SLH-DSA-SHA2-192s')
```

```
[SLH-DSA] (SLH-DSA-SHA2-192s) valid signature
```

SLH-DSA-SHAKE-192s

In [31]:

```
slh_dsa_test('SLH-DSA-SHAKE-192s')
```

```
[SLH-DSA] (SLH-DSA-SHAKE-192s) valid signature
```

SLH-DSA-SHA2-192f

In [32]:

```
slh_dsa_test('SLH-DSA-SHA2-192f')
```

```
[SLH-DSA] (SLH-DSA-SHA2-192f) valid signature
```

SLH-DSA-SHAKE-192f

In [33]:

```
slh_dsa_test('SLH-DSA-SHAKE-192f')
```

```
[SLH-DSA] (SLH-DSA-SHAKE-192f) valid signature
```

SLH-DSA-SHA2-256s

In [34]:

```
slh_dsa_test('SLH-DSA-SHA2-256s')
```

```
[SLH-DSA] (SLH-DSA-SHA2-256s) valid signature
```

SLH-DSA-SHAKE-256s

In [35]:

```
slh_dsa_test('SLH-DSA-SHAKE-256s')
```

```
[SLH-DSA] (SLH-DSA-SHAKE-256s) valid signature
```

SLH-DSA-SHA2-256f

In [36]:

```
slh_dsa_test('SLH-DSA-SHA2-256f')
```

```
[SLH-DSA] (SLH-DSA-SHA2-256f) valid signature
```

SLH-DSA-SHAKE-256f

In [37]:

```
slh_dsa_test('SLH-DSA-SHAKE-256f')
```

```
[SLH-DSA] (SLH-DSA-SHAKE-256f) valid signature
```

Invalid parameter set

```
In [38]: slh_dsa_test('SLH-DSA-LIONEL-MESSI')
```

```
[SLH-DSA] invalid call
```