Ex2

May 27, 2024

1 Trabalho Prático 4

André Freitas PG54707

Bruna Macieira PG54467

1.1 Exercício 2

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

Este padrão especifica um esquema de assinatura digital baseado em hash, chamado SLH-DSA, para aplicações que requerem uma assinatura digital em vez de uma assinatura escrita.

SLH-DSA é um esquema de assinatura baseado em hash que é construído usando outros esquemas de assinatura baseados em hash como componentes: um esquema de assinatura de poucas vezes- floresta de subconjuntos aleatórios (FORS); e um esquema de assinatura de múltiplas vezes- o Esquema de Assinatura Merkle Estendido (XMSS). O XMSS é construído usando o esquema de assinatura Winternitz One-Time Signature Plus (WOTS+) como um componente.

[ ]:
```
%pip install pycryptodome
#%pip install sagemath-standard

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

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: pycryptodome in
/home/fura/.local/lib/python3.10/site-packages (3.20.0)
Note: you may need to restart the kernel to use updated packages.
```

É criada a classe ADRS para lidar com os endereços, como demonstrado na secção 4.2 do documento. Algumas funções mencionadas na secção 4.1 utilizam esta classe como input, pois necessitam de um endereço de 32 bytes.

Esta classe trata de:

- WOTS_HASH - muda para este estado quando é preciso endereçar hashes em WOTS+ (Winternitz One-Time Signature)
- WOTS_PK - muda para este estado quando se comprimem as chaves públicas WOTS+
- TREE - muda para este estado quando são computadas as hashes numa árvore XMSS
- FORS_TREE - muda para este estado quando são computadas as hashes na árvore FORS

- FORS_ROOTS - muda para este estado quando são comprimidas as raízes k da árvore FORS
- WOTS_PRF - muda para este estado quando é gerado um valor secreto para as chaves WOTS+
- FORS_PRF - muda para este estado quando é gerado um valor secreto para as chaves FORS

Quanto às funções utilizadas:

- **init**(self, a=32): Inicializa o objeto com um array de bytes de tamanho 32
- copy(self): Retorna uma cópia do objeto ADRS
- set_layer_address(self, x): Define o endereço da camada
- set_tree_address(self, x): Define o endereço da árvore
- set_key_pair_address(self, x): Define o endereço do par de chaves
- get_key_pair_address(self): Obtém o endereço do par de chaves
- set_tree_height(self, x): Define a altura da árvore FORS
- set_chain_address(self, x): Define o endereço da cadeia WOTS+
- set_tree_index(self, x): Define o índice da árvore FORS
- get_tree_index(self): Obtém o índice da árvore FORS
- set_hash_address(self, x): Define o endereço de hash do WOTS+
- set_type_and_clear(self, t): Define o tipo de endereço e limpa os últimos 12 bytes do endereço
- adrs(self): Retorna o endereço como uma sequência de bytes
- adrsc(self): Retorna um endereço comprimido, utilizado com SHA-2

[ ]:
```python
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[20:32]
```

De seguida, é criada a classe SLH_DSA para lidar com todo o código, incluindo os algoritmos necessários.

Esta classe divide-se em várias componentes que, ao longo do paper, são necessárias para implementar os algoritmos do FIPS-205, como: set_random(self, rbg), shake256(self, x, l), shake_h_msg(self, r, pk_seed, pk_root, m), shake_prf(self, pk_seed, sk_seed, adrs), shake_prf_msg(self, sk_prf, opt_rand, m), shake_f(self, pk_seed, adrs, m1), sha256(self, x, n=32), sha512(self, x, n=64), ente outros.

Aqui estão os algoritmos do FIPS-205:

- to_int(self, s, n) - Converte uma string de bytes num inteiro
- to_byte(self, x, n) - Converte um inteiro numa string de bytes de comprimento n
- base_2b(self, s, b, out_len) - Converte uma string de bytes para uma representação em base 2**b
- chain(self, x, i, s, pk_seed, adrs) - Função de encadeamento usada na geração de chaves públicas WOTS+
- wots_pkgen(self, sk_seed, pk_seed, adrs) - Gera uma chave pública WOTS+
- wots_sign(self, m, sk_seed, pk_seed, adrs) - Gera uma assinatura WOTS+ para uma mensagem de n bytes
- wots_pk_from_sig(self, sig, m, pk_seed, adrs) - Recupera uma chave pública WOTS+ a

partir de uma assinatura e uma mensagem
- xmss_node(self, sk_seed, i, z, pk_seed, adrs) - Calcula o nó raiz de uma subárvore Merkle de chaves públicas WOTS+
- xmss_sign(self, m, sk_seed, idx, pk_seed, adrs) - Gera uma assinatura XMSS para uma mensagem
- xmss_pk_from_sig(self, idx, sig_xmss, m, pk_seed, adrs) - Recupera uma chave pública XMSS a partir de uma assinatura e uma mensagem
- ht_sign(self, m, sk_seed, pk_seed, i_tree, i_leaf) - Gera uma assinatura de hypertree para uma mensagem
- ht_verify(self, m, sig_ht, pk_seed, i_tree, i_leaf, pk_root) - Verifica uma assinatura de hypertree
- fors_sk_gen(self, sk_seed, pk_seed, adrs, idx) - Gera uma chave privada FORS
- fors_node(self, sk_seed, i, z, pk_seed, adrs) - Calcula o nó raiz de uma subárvore Merkle de valores públicos FORS
- fors_sign(self, md, sk_seed, pk_seed, adrs) - Gera uma assinatura FORS
- fors_pk_from_sig(self, sig_fors, md, pk_seed, adrs) - Recupera uma chave pública FORS a partir de uma assinatura
- keygen(self) - Gera um par de chaves SLH-DSA (chave pública e chave privada)
- split_digest(self, digest) - Auxilia na divisão de um digest em várias partes necessárias para a assinatura
- slh_sign(self, m, sk, randomize=True) - Gera uma assinatura SLH-DSA para uma mensagem
- slh_verify(self, m, sig, pk) - Verifica uma assinatura SLH-DSA para uma mensagem dada uma chave pública
- sign(self, m, sk) - Gera uma assinatura SLH-DSA para uma mensagem
- open(self, sm, pk) - Verifica uma assinatura SLH-DSA e retorna a mensagem original se a assinatura for válida

[ ]:
```python
class SLH_DSA:

    def __init__(self, hashname='SHAKE', paramid='f', n=16, h=66, d=22, hp=3,␣
↪a=6, k=33, lg_w=4, m=34, rbg=None):
        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}'

        # hash functions
        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

        # Equações da página 16
        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)) // self.lg_w + 1
        self.len = self.len1 + self.len2

        # Definição dos tamanhos dos parâmetros
        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) *␣
↪self.n

    # Random BIt Generator
    def set_random(self, rbg):
        self.rbg = rbg

    # Secção 10.1 - SLH-DSA Using SHAKE
    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)

    # Funções necessárias para a utilização de SHA-2

    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]

    # Mask Generation Function
    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]

    # Secção 10.2 - SLH-DSA Using SHA2 for Security Category 1

    def sha256_h_msg(self, r, pk_seed, pk_root, m):
```

```
        return self.mgf_sha256( r + pk_seed + self.sha256(r + pk_seed + pk_root␣
↪+ m), self.m)

    def sha256_prf(self, pk_seed, sk_seed, adrs):
        return self.sha256(pk_seed + bytes(64 - self.n) + adrs.adrsc() +␣
↪sk_seed, self.n)

    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,␣
↪self.n)

    # Secção 10.3 - SLH-DSA Using SHA2 for Security Categories 3 and 5

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

    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,␣
↪self.n)

    # Algoritmos a partir da secção 4.4

    def to_int(self, s, n):
        t = 0
        for i in range(n):
            t = (t << 8) + int(s[i])
        return t

    def to_byte(self, x, n):
        t = x
        s = bytearray(n)
        for i in range(n):
            s[n - 1 - i] = t & 0xFF
            t >>= 8
        return s

    def base_2b(self, s, b, out_len):
        i = 0 # in
        c = 0 # bits
        t = 0 # total
```

```
        v = [] # baseb
        m = (1 << b) - 1 # mask
        for j in range(out_len):
            while c < b:
                t = (t << 8) + int(s[i])
                i += 1
                c += 8
            c -= b
            v += [ (t >> c) & m ]
        return v

    def chain(self, x, i, s, pk_seed, adrs):
        if i + s >= self.w:
            return None
        t = x
        for j in range(i, i + s):
            adrs.set_hash_address(j)
            t = self.h_f(pk_seed, adrs, t)
        return t

    def wots_pkgen(self, sk_seed, pk_seed, adrs):
        sk_adrs = adrs.copy()
        sk_adrs.set_type_and_clear(ADRS.WOTS_PRF)
        sk_adrs.set_key_pair_address(adrs.get_key_pair_address())
        tmp = b''
        for i in range(self.len):
            sk_adrs.set_chain_address(i)
            sk = self.prf(pk_seed, sk_seed, sk_adrs)
            adrs.set_chain_address(i)
            tmp += self.chain(sk, 0, self.w - 1, pk_seed, adrs)
        wotspk_adrs = adrs.copy()
        wotspk_adrs.set_type_and_clear(ADRS.WOTS_PK)
        wotspk_adrs.set_key_pair_address(adrs.get_key_pair_address())
        pk = self.h_t(pk_seed, wotspk_adrs, tmp)
        return pk

    def wots_sign(self, m, sk_seed, pk_seed, 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) //␣
↪8), self.lg_w, self.len2)
        sk_adrs = adrs.copy()
        sk_adrs.set_type_and_clear(ADRS.WOTS_PRF)
        sk_adrs.set_key_pair_address(adrs.get_key_pair_address())
```

```
        sig = b''
        for i in range(self.len):
            sk_adrs.set_chain_address(i)
            sk = self.prf(pk_seed, sk_seed, sk_adrs)
            adrs.set_chain_address(i)
            sig += self.chain(sk, 0, msg[i], pk_seed, adrs)
        return sig

    def wots_pk_from_sig(self, sig, m, pk_seed, 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) //␣
↪8), self.lg_w, self.len2)
        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 - 1 -␣
↪msg[i], pk_seed, adrs)
        wots_pk_adrs = adrs.copy()
        wots_pk_adrs.set_type_and_clear(ADRS.WOTS_PK)
        wots_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())
        pk_sig = self.h_t(pk_seed, wots_pk_adrs, tmp)
        return  pk_sig

    def xmss_node(self, sk_seed, i, z, pk_seed, 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(sk_seed, pk_seed, adrs)
        else:
            lnode = self.xmss_node(sk_seed, 2 * i, z - 1, pk_seed, adrs)
            rnode = self.xmss_node(sk_seed, 2 * i + 1, z - 1, pk_seed, adrs)
            adrs.set_type_and_clear(ADRS.TREE)
            adrs.set_tree_height(z)
            adrs.set_tree_index(i)
            node = self.h_h(pk_seed, adrs, lnode + rnode)
        return node

    def xmss_sign(self, m, sk_seed, idx, pk_seed, adrs):
        auth = b''
        for j in range(self.hp):
            k = (idx >> j) ^ 1
```

```
            auth += self.xmss_node(sk_seed, k, j, pk_seed, adrs)
        adrs.set_type_and_clear(ADRS.WOTS_HASH)
        adrs.set_key_pair_address(idx)
        sig = self.wots_sign(m, sk_seed, pk_seed, adrs)
        sig_xmss = sig + auth
        return sig_xmss

    def xmss_pk_from_sig(self, idx, sig_xmss, m, pk_seed, adrs):
        adrs.set_type_and_clear(ADRS.WOTS_HASH)
        adrs.set_key_pair_address(idx)
        sig = sig_xmss[0:self.len*self.n]
        auth = sig_xmss[self.len*self.n:]
        node_0 = self.wots_pk_from_sig(sig, m, pk_seed, 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(pk_seed, adrs, node_0 + auth_k)
            else:
                adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                node_1  = self.h_h(pk_seed, adrs, auth_k + node_0)
            node_0 = node_1
        return node_0

    def ht_sign(self, m, sk_seed, pk_seed, i_tree, i_leaf):
        adrs = ADRS()
        adrs.set_tree_address(i_tree)
        sig_tmp = self.xmss_sign(m, sk_seed, i_leaf, pk_seed, adrs)
        sig_ht = sig_tmp
        root = self.xmss_pk_from_sig(i_leaf, sig_tmp, m, pk_seed, 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 = self.xmss_sign(root, sk_seed, i_leaf, pk_seed, adrs)
            sig_ht += sig_tmp
            if j < self.d - 1:
                root = self.xmss_pk_from_sig(i_leaf, sig_tmp, root, pk_seed,␣
↪adrs)
        return sig_ht
```

```
    def ht_verify(self, m, sig_ht, pk_seed, i_tree, i_leaf, pk_root):
        adrs = ADRS()
        adrs.set_tree_address(i_tree)
        sig_tmp = sig_ht[0:(self.hp + self.len)*self.n]
        node = self.xmss_pk_from_sig(i_leaf, sig_tmp, m, pk_seed, 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 = sig_ht[j*(self.hp + self.len)*self.n: (j+1)*(self.hp + self.len)*self.n]
            node = self.xmss_pk_from_sig(i_leaf, sig_tmp, node, pk_seed, adrs)
        return node == pk_root

    def fors_sk_gen(self, sk_seed, pk_seed, 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(pk_seed, sk_seed, sk_adrs)

    def fors_node(self, sk_seed, i, z, pk_seed, adrs):
        if z > self.a or i >= (self.k << (self.a - z)):
            return None
        if z == 0:
            sk = self.fors_sk_gen(sk_seed, pk_seed, adrs, i)
            adrs.set_tree_height(0)
            adrs.set_tree_index(i)
            node = self.h_f(pk_seed, adrs, sk)
        else:
            lnode = self.fors_node(sk_seed, 2 * i, z - 1, pk_seed, adrs)
            rnode = self.fors_node(sk_seed, 2 * i + 1, z - 1, pk_seed, adrs)
            adrs.set_tree_height(z)
            adrs.set_tree_index(i)
            node = self.h_h(pk_seed, adrs, lnode + rnode)
        return node

    def fors_sign(self, md, sk_seed, pk_seed, 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(sk_seed, pk_seed, adrs, (i << self.a) + indices[i])
            for j in range(self.a):
```

```
            s = (indices[i] >> j) ^ 1
            sig_fors += self.fors_node(sk_seed, (i << (self.a - j)) + s, j,␣
↪pk_seed, adrs)
        return sig_fors

    def fors_pk_from_sig(self, sig_fors, md, pk_seed, adrs):
        def get_sk(sig_fors, i):
            return sig_fors[i*(self.a+1)*self.n:(i*(self.a+1)+1)*self.n]

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

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

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

            auth = get_auth(sig_fors, 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(pk_seed, adrs, node_0 + auth_j)
                else:
                    adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                    node_1 = self.h_h(pk_seed, 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(pk_seed, fors_pk_adrs, root)
        return pk

    def keygen(self):

        # O comportamento é diferente se forem realizadas três chamadas␣
↪distintas para o RBG. O código de referência faz uma chamada e divide-a.
        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 (pk, sk)

    # Necessário para o slh_sign e slh_verify
    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 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_root)

    # Testes
    def sign(self, m, sk):
        sig = self.slh_sign(m, sk)
        return sig + m

    def open(self, sm, pk):
        if len(sm) < self.sig_sz:
            return None
        sig = sm[0:self.sig_sz]
        m = sm[self.sig_sz:]
        if self.slh_verify(m, sig, pk):
            return m
        return None
```

De seguida, são adicionados os12 parâmetros de SLH-DSA que podem ser utilizados. Na secção 10 do FIPS-205, existe uma tabela que representa o conteúdo abaixo. Um conjunto de parâmetros consiste em parâmetros para WOTS+ (n e lg w), XMSS e a árvore hiperbólica SLH-DSA (h e d), e FORS (k e a), bem como instâncias para as funções H_msg, PRF, PRF_msg, F, H e T_l.

Os conjuntos de parâmetros com n = 16 são considerados na categoria de segurança 1, os conjuntos de parâmetros com n = 24 são considerados na categoria de segurança 3 e os conjuntos de parâmetros com n = 32 são considerados na categoria de segurança 5.

O “s” no final do algoritmo representa “small simple” e o “f” representa “fast simple”. Apenas as

instâncias “simple” são aprovadas, de acordo com o documento (página 2). “s” cria assinaturas relativamente pequenas e “f” cria assinaturas de forma rápida (página 38).

[ ]:
```
SLH_DSA_SHA2_128s = SLH_DSA(hashname='SHA2', paramid='s', n=16, h=63, d=7,␣
 ↪hp=9, a=12, k=14, lg_w=4, m=30)
SLH_DSA_SHAKE_128s = SLH_DSA(hashname='SHAKE', paramid='s', n=16, h=63, d=7,␣
 ↪hp=9, a=12, k=14, lg_w=4, m=30)
SLH_DSA_SHA2_128f = SLH_DSA(hashname='SHA2', paramid='f', n=16, h=66, d=22,␣
 ↪hp=3, a=6, k=33, lg_w=4, m=34)
SLH_DSA_SHAKE_128f = SLH_DSA(hashname='SHAKE', paramid='f', n=16, h=66, d=22,␣
 ↪hp=3, a=6, k=33, lg_w=4, m=34)

SLH_DSA_SHA2_192s = SLH_DSA(hashname='SHA2', paramid='s', n=24, h=63, d=7,␣
 ↪hp=9, a=14, k=17, lg_w=4, m=39)
SLH_DSA_SHAKE_192s = SLH_DSA(hashname='SHAKE', paramid='s', n=24, h=63, d=7,␣
 ↪hp=9, a=14, k=17, lg_w=4, m=39)
SLH_DSA_SHA2_192f = SLH_DSA(hashname='SHA2', paramid='f', n=24, h=66, d=22,␣
 ↪hp=3, a=8, k=33, lg_w=4, m=42)
SLH_DSA_SHAKE_192f = SLH_DSA(hashname='SHAKE', paramid='f', n=24, h=66, d=22,␣
 ↪hp=3, a=8, k=33, lg_w=4, m=42)

SLH_DSA_SHA2_256s = SLH_DSA(hashname='SHA2', paramid='s', n=32, h=64, d=8,␣
 ↪hp=8, a=14, k=22, lg_w=4, m=47)
SLH_DSA_SHAKE_256s = SLH_DSA(hashname='SHAKE', paramid='s', n=32, h=64, d=8,␣
 ↪hp=8, a=14, k=22, lg_w=4, m=47)
SLH_DSA_SHA2_256f = SLH_DSA(hashname='SHA2', paramid='f', n=32, h=68, d=17,␣
 ↪hp=4, a=9, k=35, lg_w=4, m=49)
SLH_DSA_SHAKE_256f = SLH_DSA(hashname='SHAKE', paramid='f', n=32, h=68, d=17,␣
 ↪hp=4, a=9, k=35, lg_w=4, m=49)

# Ordem de acordo com os ficheiros

SLH_DSA_ALL = [ SLH_DSA_SHA2_128f,  SLH_DSA_SHA2_128s,
                SLH_DSA_SHA2_192f,  SLH_DSA_SHA2_192s,
                SLH_DSA_SHA2_256f,  SLH_DSA_SHA2_256s,
                SLH_DSA_SHAKE_128f, SLH_DSA_SHAKE_128s,
                SLH_DSA_SHAKE_192f, SLH_DSA_SHAKE_192s,
                SLH_DSA_SHAKE_256f, SLH_DSA_SHAKE_256s, ]
```

Passando agora para o teste do algoritmo:

Pensou-se em utilizar o sage.crypto.block_cipher.present para realizar processos de cifragem e decifragem de dados. No entanto, o processo ficou mais moroso quando este foi implementado, pelo que não se utilizou essa técnica. Esse processo funciona da seguinte maneira:

[ ]:
```
#Código exemplo
```

```
# from sage.crypto.block_cipher.present import PRESENT
# present = PRESENT()
# present.encrypt(plaintext=0, key=0).hex()
# present.decrypt(ciphertext=0x2844b365c06992a3, key=0)

# Como ficaria a implementação com o PRESENT

# from sage.crypto.block_cipher.present import PRESENT

# def get_bytes(self, num_bytes):
#     tmp = b''
#     present = PRESENT()
#     while len(tmp) < num_bytes:
#         self.__increment_ctr()
#         # O método encrypt do PRESENT aceita apenas um bloco de 64 bits (8␣
 ↪bytes), então dividimos em blocos menores
#         block_size = 8
#         for i in range(0, num_bytes, block_size):
#             block = present.encrypt(plaintext=int.from_bytes(self.ctr,␣
 ↪byteorder='big'), key=int.from_bytes(self.key, byteorder='big'))  #␣
 ↪Criptografa o bloco atual
#             tmp += block.to_bytes(block_size, byteorder='big')
#             # Incrementa o contador apenas para o próximo bloco
#             self.__increment_ctr()
#     return tmp[:num_bytes]
```

Ainda com o SageMath, era suposto utilizar:

- sage.misc.verbose, pois este imprime o valor de cputime(t), em que t é um parâmetro
- ascii_to_bin do sage.crypto.utils para realizar o encode no teste

No entanto, o SageMath mostrou interferência com outras bibliotecas do Python que vimos ser necessárias para o processo, como a Integer, pelo que optámos por não utilizar a biblioteca sage.

Assim sendo, o processo de teste ficou da seguinte maneira:

[ ]:
```
from sage.all import *
```

[ ]:
```
from Crypto.Cipher import AES
from Crypto.Hash import SHA256
import time
from tabulate import tabulate

# AES-256 CTR extrai saídas 'falsas' de DRBG que são compatíveis com␣
 ↪randombytes() nos testes KAT
class KAT_DRBG:

    def __init__(self, seed):
```

```
        self.seed_length = 48
        assert len(seed) == self.seed_length
        self.key = b'\x00'*32
        self.ctr = b'\x00'*16
        update = self.get_bytes(self.seed_length)
        update = bytes(a^b for a, b in zip(update, seed))
        self.key = update[:32]
        self.ctr = update[32:]

    def __increment_ctr(self):
        x = int(int.from_bytes(self.ctr, 'big') + 1)
        self.ctr = x.to_bytes(16, byteorder='big')

    def get_bytes(self, num_bytes):
        tmp = b''
        cipher = AES.new(self.key, AES.MODE_ECB)
        while len(tmp) < num_bytes:
            self.__increment_ctr()
            tmp += cipher.encrypt(self.ctr)
        return tmp[:num_bytes]

    def random_bytes(self, num_bytes):
        output_bytes = self.get_bytes(num_bytes)
        update = self.get_bytes(48)
        self.key = update[:32]
        self.ctr = update[32:]
        return output_bytes

# Test bench

def test_rsp(iut, katnum=100):
    """Generate NIST-style KAT response strings and perform benchmarks."""
    fail = 0
    drbg = KAT_DRBG(bytes([i for i in range(48)]))
    kat = f'# {iut.algname}\n\n'
    results = []

    for count in range(katnum):
        print(f'# {count}/{katnum} {iut.stdname}', flush=True)
        kat += f'count = {count}\n'
        seed = drbg.random_bytes(48)
        iut.set_random(KAT_DRBG(seed).random_bytes)
        kat += f'seed = {seed.hex().upper()}\n'
        mlen = 33 * (count + 1)
        kat += f'mlen = {mlen}\n'
        msg = drbg.random_bytes(mlen)
        kat += f'msg = {msg.hex().upper()}\n'
```

```
        # Keygen do benchmark
        start_time = time.time()
        (pk, sk) = iut.keygen()
        keygen_time = time.time() - start_time
        kat += f'pk = {pk.hex().upper()}\n'
        kat += f'sk = {sk.hex().upper()}\n'

        # Assina o benchmark
        start_time = time.time()
        sm = iut.sign(msg, sk)
        sign_time = time.time() - start_time
        kat += f'smlen = {len(sm)}\n'
        kat += f'sm = {sm.hex().upper()}\n'

        # Verifica o benchmark
        start_time = time.time()
        m2 = iut.open(sm, pk)
        verify_time = time.time() - start_time
        if m2 is None or m2 != msg:
            fail += 1
            kat += f'(verify error)\n'
        kat += '\n'

        # Envia os resultados para a tabela
        results.append([
            count,
            keygen_time,
            sign_time,
            verify_time,
            len(pk),
            len(sk),
            len(sm)
        ])

    return kat, results

if __name__ == "__main__":
    katnum = 1
    headers = ["Count", "Keygen Time (s)", "Sign Time (s)", "Verify Time (s)",␣
 ↪"Public Key Size (bytes)", "Private Key Size (bytes)", "Signature Size␣
 ↪(bytes)"]
    with open('benchmark_results.txt', 'w') as f:
        for iut in SLH_DSA_ALL:
            kat, results = test_rsp(iut, katnum=katnum)
            md = SHA256.new(kat.encode('ASCII')).hexdigest()
            f.write(f'{md} {iut.stdname} ({katnum})\n')
```

```
            f.write(tabulate(results, headers=headers, tablefmt="grid"))
            f.write('\n\n')
        print("Results written to benchmark_results.txt")
```

```
# 0/1 SLH-DSA-SHA2-128f
# 0/1 SLH-DSA-SHA2-128s
# 0/1 SLH-DSA-SHA2-192f
# 0/1 SLH-DSA-SHA2-192s
# 0/1 SLH-DSA-SHA2-256f
# 0/1 SLH-DSA-SHA2-256s
# 0/1 SLH-DSA-SHAKE-128f
# 0/1 SLH-DSA-SHAKE-128s
# 0/1 SLH-DSA-SHAKE-192f
# 0/1 SLH-DSA-SHAKE-192s
# 0/1 SLH-DSA-SHAKE-256f
# 0/1 SLH-DSA-SHAKE-256s
Results written to benchmark_results.txt
```

Os resultados apenas podem ser visualizados quando todo o benchmark terminar, pois só nessa altura os dados são enviados para o ficheiro .txt.

Este teste mostra apenas as hashes para 1, 10 e 100 primeiras respostas, caso contrário o output seria muito grande, consumindo vários recursos da máquina.

[ ]:

[ ]: