Estruturas Criptográficas - Criptografia e Segurança da Informação

Grupo 03

(PG54177) Ricardo Alves Oliveira

(PG54236) Simão Oliveira Alvim Barroso

TP4 - Exercício 2

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

A nossa resolução baseou-se principalmente no FIPS 205. Também tiramos algumas inspirações de recursos que encontramos na internet como o paper de submissão ao concurso pós-quantico da NIST.

Temos portanto uma lista de algoritmos para implementar, assim como fizemos nos exercicios antes deste trabalho:

List of Algorithms

Imports necessários

In [ ]:
```
import random
import hashlib
import os
```

O SPHINCS+ é um esquema de assinatura digital baseado em *hashes* criptográficos que é seguro contra ataques de computadores quânticos. Destaca-se que é um esquema *stateless*, ou seja, não necessita de registar informações após cada assinatura.

Parâmetros do esquema SPHINCS+

In [ ]:
```
n = 32 # The security parameter
w = 256
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 = 12
d = 3
h_prime = h // d
k = 8
a = 4
t = 2^a
```

base_w

In [ ]:
```
def base_w(X, w, out_len):
    i_in = 0
    i_out = 0
    ui_total = 0
    i_bits = 0
    basew = [None] * out_len

    for consumed in range(out_len):
        if (i_bits == 0):
            ui_total = X[i_in]
            i_in += 1
            i_bits += 8

        i_bits -= log(w, 2)
        basew[i_out] = (ui_total >> i_bits) & (w - 1)
        i_out += 1

    return basew
```

ADRS - adress

In [ ]:
```
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 = int(self.layer).to_bytes(4, byteorder='big')
        adrs += int(self.tree_address).to_bytes(12, byteorder='big')
        adrs += int(self.type).to_bytes(4, byteorder='big')
        adrs += int(self.word_1).to_bytes(4, byteorder='big')
        adrs += int(self.word_2).to_bytes(4, byteorder='big')
        adrs += int(self.word_3).to_bytes(4, byteorder='big')
        return adrs

    def reset_words(self):
        self.word_1 = 0
        self.word_2 = 0
        self.word_3 = 0

    def set_type(self, val):
        self.type = val
        self.word_2 = 0
        self.word_3 = 0
        self.word_1 = 0

    def set_layer_address(self, val):
        self.layer = val

    def set_tree_address(self, val):
        self.tree_address = val

    def set_key_pair_address(self, val):
        self.word_1 = val

    def get_key_pair_address(self):
        return self.word_1

    def set_chain_address(self, val):
        self.word_2 = val

    def set_hash_address(self, val):
        self.word_3 = val

    def set_tree_height(self, val):
        self.word_2 = val

    def get_tree_height(self):
        return self.word_2
```

```python
    def set_tree_index(self, val):
        self.word_3 = val

    def get_tree_index(self):
        return self.word_3
```

Funções auxiliares

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

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

    pre_hashed = m.digest()
    hashed = pre_hashed[:digest_size]

    return hashed

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


def hash_msg(r, public_seed, public_root, value, digest_size=n):
    m = hashlib.sha256()

    m.update(str(r).encode('ASCII'))
    m.update(public_seed)
    m.update(public_root)
    m.update(value)

    pre_hashed = m.digest()
    hashed     = pre_hashed[:digest_size]
    i = 0
    while len(hashed) < digest_size:
        i += 1
        m = hashlib.sha256()

        m.update(str(r).encode('ASCII'))
        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):
    random.seed(int.from_bytes(secret_seed + opt + hash_msg(b'0', b'0', b
    return int(random.randint(0, 256 ^ n)).to_bytes(n, byteorder='big')

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
```

WOTS+ Function chain

In [ ]:
```
# Input: Input string X, start index i, number of steps s, public seed PK
# 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
```

Função *wots_pkGen*

In [ ]:
```
# 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):
    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())
        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)
    return pk
```

Função *wots_sign*

In [ ]:
```
# Input: Message M, secret seed SK.seed, public seed PK.seed, address ADR
# 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
    if (len_2 * math.floor(math.log(w, 2))) % 8 != 0:
        csum = csum << (8 - (len_2 * math.floor(math.log(w, 2))) % 8 )
    len2_bytes = math.ceil((len_2 * math.floor(math.log(w, 2))) / 8)
    msg += base_w(int(csum).to_bytes(len2_bytes, byteorder='big'), w, len

    sig = []
    for i in range(0, len_0):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk = prf(secret_seed, adrs.copy())
        sig += [chain(sk, 0, msg[i], public_seed, adrs.copy())]

    return sig
```

Função *wots_pkFromSig*

In [ ]:
```
def wots_pk_from_sig(sig, m, public_seed, adrs: ADRS):
    csum = 0
    wots_pk_adrs = adrs.copy()

    # convert message to base w
    msg = base_w(m, w, len_1)

    # compute checksum
    for i in range(0, len_1):
        csum += w - 1 - msg[i]

    # convert csum to base w
    if (len_2 * math.floor(math.log(w, 2))) % 8 != 0:
        padding = (len_2 * math.floor(math.log(w, 2))) % 8
    else:
        padding = 8
    csum = csum << (8 - padding)
    msg += base_w(int(csum).to_bytes(math.ceil((len_2 * math.floor(math.l

    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.co

    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)
    return pk_sig
```

Função *xms_node*

In [ ]:
```
# Input: Secret seed SK.seed, start index s, target node height z, public
# Output: n-byte root node - top node on Stack
```

```
def node(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']
                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']
```

Função xmss_sign

In [ ]:
```
# Input: n-byte message M, secret seed SK.seed, index idx, public seed PK
# Output: XMSS signature SIG_XMSS = (sig || AUTH)
def xmss_sign(m, secret_seed, idx, public_seed, adrs):
    # build authentication path
    auth = []
    for j in range(0, h_prime):
        ki = math.floor(idx // 2^j)
        if ki % 2 == 1:
            ki -= 1
        else:
            ki += 1
        auth += [node(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
```

Função *xmss_pkFromSig*

In [ ]:
```
# Input: index idx, XMSS signature SIG_XMSS = (sig || AUTH), n-byte messa
# 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)

    node0 = wots_pk_from_sig(sig, m, public_seed, adrs.copy())
    node1 = 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)
            node1 = hash(public_seed, adrs.copy(), node0 + auth[i], n)
        else:
            adrs.set_tree_index( (adrs.get_tree_index() - 1) // 2)
            node1 = hash(public_seed, adrs.copy(), auth[i] + node0, n)
        node0 = node1

    return node0
```

Função *ht_sign*

In [ ]:

```
# Input: Message M, private seed SK.seed, public seed PK.seed, tree index
# Output: HT signature SIG_HT
def ht_sign(m, secret_seed, public_seed, idx_tree, idx_leaf):
    # init
    adrs = ADRS()

    # sign
    adrs.set_layer_address(0)
    adrs.set_tree_address(idx_tree)
    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, adr
        sig_ht = sig_ht + sig_tmp
        if j < d - 1:
            root = xmss_pk_from_sig(idx_leaf, sig_tmp, root, public_seed,

    return sig_ht
```

Função *ht_verify*

In [ ]:

```
# Input: Message M, signature SIG_HT, public seed PK.seed, tree index idx
# 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, adr
    if node == public_key_ht:
        return True
    else:
        return False
```

Função *fors_SKgen*

In [ ]:

```
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())
    return sk
```

Função *fors_node*

In [ ]:

```
def fors_node(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())
        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']
                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']
```

Função *fors_sign*

In [ ]:

```
# Input: Bit string M, secret seed SK.seed, address ADRS, public seed PK.
# Output: FORS signature SIG_FORS
def fors_sign(m, secret_seed, public_seed, adrs):
    # compute signature elements
    m_int = int.from_bytes(m, 'big')
```

```
    sig_fors = []
    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())]

        # compute auth path
        auth = []
        for j in range(0, a):
            s = math.floor(idx // 2 ^ j)
            if s % 2 == 1:
                s -= 1
            else:
                s += 1
            auth += [fors_node(secret_seed, i * t + s * 2^j, j, public_se
        sig_fors += auth

    return sig_fors
```

Função *fors_pkFromSig*

In [ ]:

```
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)
        node_1 = 0

        # compute root from lead and AUTH
        auth = sigs[i][1]
        adrs.set_tree_index(i * t + idx)

        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],
            else:
                adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
                node_1 = hash(public_seed, adrs.copy(), auth[j] + node_0,

            node_0 = node_1
```

```
        root += node_0

    fors_pk_adrs = adrs.copy() # copy address to create FTS public key ad
    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
```

Função *slh_keygen*

In [ ]:
```
def slh_key_gen():
    secret_seed = os.urandom(n)
    secret_prf = os.urandom(n)
    public_seed = os.urandom(n)

    adrs = ADRS()
    adrs.set_layer_address(d - 1)
    adrs.set_tree_address(0)

    public_root = node(secret_seed, 0, h_prime, public_seed, adrs.copy())


    return [secret_seed, secret_prf, public_seed, public_root], [public_s
```

Função *slh_sign*

In [ ]:
```
RANDOMIZE = True


def slh_sign(m, secret_key):
    # Init
    adrs = ADRS()
    secret_seed = secret_key[0]
    secret_prf = secret_key[1]
    public_seed = secret_key[2]
    public_root = secret_key[3]

    # Generate randomizer
    opt = bytes(n)
    if RANDOMIZE:
        opt = os.urandom(n)
    r = prf_msg(secret_prf, opt, m)
    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_
    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 = int(md_int).to_bytes(math.ceil(k * a / 8), 'big')
    idx_tree = int.from_bytes(tmp_idx_tree, 'big') >> (len(tmp_idx_tree)
    idx_leaf = int.from_bytes(tmp_idx_leaf, 'big') >> (len(tmp_idx_leaf)

    # 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_lea
    sig += [sig_ht]

    return sig
```

Função *slh_verify*

In [ ]:
```
# Input: Message M, signature SIG, public key PK
# Output: Boolean
def slh_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_
    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 = int(md_int).to_bytes(math.ceil(k * a / 8), 'big')
    idx_tree = int.from_bytes(tmp_idx_tree, 'big') >> (len(tmp_idx_tree)
    idx_leaf = int.from_bytes(tmp_idx_leaf, 'big') >> (len(tmp_idx_leaf)

    # 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, pu
```

Exemplo de teste

In [ ]:
```
# Generate key pair
sk, pk = slh_key_gen()

print("Private key:\n", sk)
print("Public key:\n", pk)

m = b'Mensagem teste!!!!!! :)'
print("\nMessage to be signed:\n", m)

s = slh_sign(m, sk)

print("Verificado?\n", slh_verify(m, s, pk))
```

```
Private key:
 [b'\xe1\xf5\xa3H\x01\x84\x97\xec\x15\xba\xe9L\r\xf2\xe0}\x7f\t\xd7\xc2E\x
b1\x1e\x8b{\xfb\xa7\xaau~\xbe\xea', b'\x02\xb0\x94CX"\xce=\x06\xbc\x9f\xf2
3\x81no\x83\x03\xf5\x0e\xfa\x87\xbb\x19\xc1\x02k\xf1\x88B)\xa5', b'\x8e\x9
6i+tC\x17\x8ad\xaa\x00\xfdu\x11\x11\x89\x91pV\x9b\xf1\xa2qg\x88\x16^3@\xca
\xc6\xc9', b'\x96\x17\xc8\xb9\x88\x17\xce\xd9\xce\x15\xf6\xc6\xaae=\x96\x8
8\xe5]\xfcb\x03a8\xf8L|`h4o$']
Public key:
 [b'\x8e\x96i+tC\x17\x8ad\xaa\x00\xfdu\x11\x11\x89\x91pV\x9b\xf1\xa2qg\x88
\x16^3@\xca\xc6\xc9', b'\x96\x17\xc8\xb9\x88\x17\xce\xd9\xce\x15\xf6\xc6\x
aae=\x96\x88\xe5]\xfcb\x03a8\xf8L|`h4o$']

Message to be signed:
 b'Mensagem teste!!!!!! :)'
Verificado?
 True
```