g13-tp4-ex2

May 28, 2024

1 Estruturas criptograficas: TP4 problema 2

1.1 Sphincs+

Este é um algoritmo de assinatura digital pós-quântico que nos permite perceber se aconteceu uma alteração não autorizada, ou seja, o remetente poderá utilizar a assinatura digital para provar, que uma determinada informação não foi modificada e que a mesma veio de um determinado emissor.

WOTS (Winternitz One-Time Signature) É um esquema de assinatura digital que é usado apenas uma vez.

XMSS (Extended Merkle Signature Scheme) É uma extensão do WOTS que cria várias assinaturas usando uma única chave privada, usando uma estrutura de árvore de Merkle.

Fors (Few-Time Signature Scheme) É um esquema de assinatura baseado em árvores de Merkle.

ADRS A classe ADRS é utilizada para guardar endereços no contexto do esquema de assinatura SPHINCS (Sphincs - Signature Scheme) e possui vários métodos para manobrar os diferentes campos de cada endereço.

Adicionalmente, a classe possui algumas funções de hash e geradores pseudo aleatórios.

[3]:
```
import os
import math
import random
import hashlib

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_bin = int(self.layer).to_bytes(4, byteorder='big')
        adrs_bin += self.tree_address.to_bytes(12, byteorder='big')
        adrs_bin += self.type.to_bytes(4, byteorder='big')
        adrs_bin += self.word_1.to_bytes(4, byteorder='big')
        adrs_bin += self.word_2.to_bytes(4, byteorder='big')
        adrs_bin += self.word_3.to_bytes(4, byteorder='big')
        return adrs_bin

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

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

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

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

    def get_key_pair_address(self):
        return self.word_1

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

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

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

    def get_tree_height(self):
        return self.word_2

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

    def get_tree_index(self):
        return self.word_3

# FUNÇÕES DE HASH TWEAKABLES

# Calcula um hash com base em uma seed, um endereço ADRS, e um valor de entrada.
def hash_(seed, adrs: ADRS, value, digest_size):
    hasher = hashlib.sha256()
    hasher.update(seed)
    hasher.update(adrs.to_bin())
    hasher.update(value)
    hashed_value = hasher.digest()[:digest_size]
    return hashed_value

# Gera uma chave pseudorrandômica com base numa seed secreta e um endereço ADRS
def prf(secret_seed, adrs, digest_size):
    # Pseudorandom key generation
    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')

# Comprimir uma mensagem a ser assinada usando a função de hash SHA256
def hash_msg(r, public_seed, public_root, message, digest_size):
    # Comprime a mensagem a ser assinada
    hasher = hashlib.sha256()
    hasher.update(r)
    hasher.update(public_seed)
    hasher.update(public_root)
    hasher.update(message)
    hashed_msg = hasher.digest()[:digest_size]

    i = 0
    while len(hashed_msg) < digest_size:
        i += 1
        hasher = hashlib.sha256()
        hasher.update(r)
        hasher.update(public_seed)
```

```
        hasher.update(public_root)
        hasher.update(message)
        hasher.update(bytes([i]))
        hashed_msg += hasher.digest()[:digest_size - len(hashed_msg)]

    return hashed_msg

# Gera aleatoriedade para a compressão da mensagem.
def prf_msg(secret_seed, opt, message, digest_size):
    # Gera aleatoriedade para a compressão da mensagem
    random.seed(int.from_bytes(secret_seed + opt + hash_msg(b'0', b'0', b'0',␣
 ↪message, digest_size * 2), "big"))
    return random.randint(0, 256 ** digest_size - 1).to_bytes(digest_size,␣
 ↪byteorder='big')

# Converte uma string em um array de inteiros baseado em um determinado valor␣
 ↪de w (base)
# Input: len_X-byte string X, int w, tamanho do output out_len
# Output: out_len int array basew
def base_w(input_bytes, base, output_len):
    input_idx = 0
    output_idx = 0
    total = 0
    bits = 0
    basew = list()

    for _ in range(output_len):
        if bits == 0:
            total = input_bytes[input_idx]
            input_idx += 1
            bits += 8
        bits -= math.floor(math.log(base, 2))
        basew.append((total >> bits) % base)
        output_idx += 1

    return basew
```

2 FORS few-time signature scheme

Este código implementa a classe Fors (Few-Time Signature Scheme).

Um esquema de assinatura criptográfica baseado em árvores de Merkle chamadas de FORS (Few-time Signature Scheme).

[4]:
```
class Fors:
    def __init__(self):
        self._n = 16  # tamanho em bytes do nó da árvore
```

```
        self._k = 10  # número de árvores FORS
        self._a = 15
        self._t = 2 ** self._a

    # Recebe uma assinatura sig e retorna uma lista com as autenticações correspondentes
    def auths_from_sig_fors(self, assinatura):
        assinaturas = []
        for indice in range(self._k):
            assinaturas.append([])
            assinaturas[indice].append(assinatura[(self._a + 1) * indice])
            assinaturas[indice].append(assinatura[((self._a + 1) * indice + 1):((self._a + 1) * (indice + 1))])

        return assinaturas

    # Gera a chave secreta (secret key) para uma árvore FORS específica,
    # com base em uma seed secreta, um endereço ADRS e um índice
    def fors_sk_gen(self, semente_secreta, endereco: ADRS, indice):
        endereco.set_tree_height(0)
        endereco.set_tree_index(indice)
        chave_secreta = prf(semente_secreta, endereco.copy(), self._n)

        return chave_secreta

    # Calcula o nó raiz de uma árvore FORS, com base em uma seed secreta,
    # um índice inicial, uma altura alvo, uma seed pública e um endereço ADRS

    # 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, semente_secreta, indice_inicial, altura_alvo, semente_publica, endereco):
        if indice_inicial % (1 << altura_alvo) != 0:
            return -1

        pilha = []

        for indice in range(2 ** altura_alvo):
            endereco.set_tree_height(0)
            endereco.set_tree_index(indice_inicial + indice)
            chave_secreta = prf(semente_secreta, endereco.copy(), self._n)
            no = hash_(semente_publica, endereco.copy(), chave_secreta, self._n)

            endereco.set_tree_height(1)
            endereco.set_tree_index(indice_inicial + indice)
            if len(pilha) > 0:
```

```
                while pilha[len(pilha) - 1]['height'] == endereco.
↪get_tree_height():
                    endereco.set_tree_index((endereco.get_tree_index() - 1) //␣
↪2)
                    no = hash_(semente_publica, endereco.copy(), pilha.
↪pop()['node'] + no, self._n)

                    endereco.set_tree_height(endereco.get_tree_height() + 1)

                    if len(pilha) <= 0:
                        break
            pilha.append({'node': no, 'height': endereco.get_tree_height()})

        return pilha.pop()['node']

    # Gera a chave pública (public key) para o esquema FORS, com base em uma␣
↪seed
    # secreta, uma seed pública e um endereço ADRS

    # Input: Secret seed SK.seed, public seed PK.seed, address ADRS
    # Output: FORS public key PK
    def fors_pk_gen(self, semente_secreta, semente_publica, endereco: ADRS):
        endereco_fors_pk = endereco.copy()

        raiz = bytes()
        for indice in range(0, self._k):
            raiz += self.fors_treehash(semente_secreta, indice * self._t, self.
↪_a, semente_publica, endereco)

        endereco_fors_pk.set_type(ADRS.FORS_ROOTS)
        endereco_fors_pk.set_key_pair_address(endereco.get_key_pair_address())
        chave_publica = hash_(semente_publica, endereco_fors_pk, raiz, self._n)
        return chave_publica

    # Assina uma mensagem usando o esquema FORS, com base em uma mensagem, uma␣
↪seed
    # secreta, uma seed pública e um endereço ADRS

    # Input: Bit string M, secret seed SK.seed, address ADRS, public seed PK.
↪seed
    # Output: FORS signature SIG_FORS
    def fors_sign(self, mensagem, semente_secreta, semente_publica, endereco):
        mensagem_inteira = int.from_bytes(mensagem, 'big')
        assinatura_fors = []

        for indice in range(self._k):
```

```
            indice_ajustado = (mensagem_inteira >> (self._k - 1 - indice) *␣
↪self._a) % self._t

            endereco.set_tree_height(0)
            endereco.set_tree_index(indice * self._t + indice_ajustado)
            assinatura_fors += [prf(semente_secreta, endereco.copy(), self._n)]

            autenticacao = []

            for j in range(self._a):
                indice_inicial = math.floor(indice_ajustado // 2 ** j)
                if indice_inicial % 2 == 1:  # XORING indice_inicial / 2**j␣
↪with 1
                    indice_inicial -= 1
                else:
                    indice_inicial += 1

                autenticacao += [self.fors_treehash(semente_secreta, indice *␣
↪self._t + indice_inicial * 2 ** j, j, semente_publica, endereco.copy())]

            assinatura_fors += autenticacao

        return assinatura_fors

    # Verifica a assinatura e recupera a chave pública correspondente, com base
    # em uma assinatura sig, uma mensagem, uma seed pública e um endereço ADRS

    # 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, assinatura_fors, mensagem, semente_publica,␣
↪endereco: ADRS):
        mensagem_inteira = int.from_bytes(mensagem, 'big')

        assinaturas = self.auths_from_sig_fors(assinatura_fors)
        raiz = bytes()

        for indice in range(self._k):
            indice_ajustado = (mensagem_inteira >> (self._k - 1 - indice) *␣
↪self._a) % self._t

            chave_secreta = assinaturas[indice][0]
            endereco.set_tree_height(0)
            endereco.set_tree_index(indice * self._t + indice_ajustado)
            no_0 = hash_(semente_publica, endereco.copy(), chave_secreta, self.
↪_n)
```

```
            no_1 = 0

            autenticacao = assinaturas[indice][1]
            endereco.set_tree_index(indice * self._t + indice_ajustado)

            for j in range(self._a):
                endereco.set_tree_height(j + 1)

                if math.floor(indice_ajustado / 2 ** j) % 2 == 0:
                    endereco.set_tree_index(endereco.get_tree_index() // 2)
                    no_1 = hash_(semente_publica, endereco.copy(), no_0 +␣
↪autenticacao[j], self._n)
                else:
                    endereco.set_tree_index((endereco.get_tree_index() - 1) //␣
↪2)
                    no_1 = hash_(semente_publica, endereco.copy(),␣
↪autenticacao[j] + no_0, self._n)

                no_0 = no_1

            raiz += no_0

        endereco_fors_pk = endereco.copy()
        endereco_fors_pk.set_type(ADRS.FORS_ROOTS)
        endereco_fors_pk.set_key_pair_address(endereco.get_key_pair_address())

        chave_publica = hash_(semente_publica, endereco_fors_pk, raiz, self._n)
        return chave_publica
```

3 WOTS

O Sphincs+ utiliza um One-Time Signature (OTS), onde cada par de chaves pode ser utilizado para assinar uma única mensagem.

[5]:
```
class Wots:
    def __init__(self):
        self._n = 16  # Parametro de segurança
        self._w = 16  # Parametro de Winternitz (4, 16 ou 256)
        self._h = 64  # Altura da Hypertree
        self._d = 8   # Camadas da Hypertree
        self._a = 15  # Numero de folhas de cada arvore no FORS

        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 # n-bit values in WOTS+ sk, pk,␣
↪and signature.

    # Retorna o valor de F iterado s vezes em x
    # 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, entrada, inicio, passos, semente_publica, endereco: ADRS):
        if passos == 0:
            return bytes(entrada)

        if (inicio + passos) > (self._w - 1):
            return -1

        temp = self.chain(entrada, inicio, passos - 1, semente_publica,␣
↪endereco)

        endereco.set_hash_address(inicio + passos - 1)
        temp = hash_(semente_publica, endereco, temp, self._n)

        return temp

    # Gera a chave privada (secret key) para o esquema WOTS+,
    # com base em uma seed secreta e um endereço ADRS.
    # Input: secret seed SK.seed, address ADRS
    # Output: WOTS+ private key sk
    def wots_sk_gen(self, semente_secreta, endereco: ADRS):
        chave_privada = []
        for i in range(self._len_0):
            endereco.set_chain_address(i)
            endereco.set_hash_address(0)
            chave_privada.append(prf(semente_secreta, endereco.copy(), self._n))
        return chave_privada

    # Gera a chave pública (public key) para o esquema WOTS+, com base em uma␣
↪seed
    # secreta, uma seed pública e um endereço ADRS
    # Input: secret seed SK.seed, address ADRS, public seed PK.seed
    # Output: WOTS+ public key pk
    def wots_pk_gen(self, semente_secreta, semente_publica, endereco: ADRS):
        endereco_pk = endereco.copy()
        temp = bytes()
        for i in range(self._len_0):
            endereco.set_chain_address(i)
            endereco.set_hash_address(0)
            chave_privada = prf(semente_secreta, endereco.copy(), self._n)
```

```
            temp += bytes(self.chain(chave_privada, 0, self._w - 1,␣
↪semente_publica, endereco.copy()))

        endereco_pk.set_type(ADRS.WOTS_PK)
        endereco_pk.set_key_pair_address(endereco.get_key_pair_address())

        chave_publica = hash_(semente_publica, endereco_pk, temp, self._n)
        return chave_publica

    # Assina uma mensagem usando o esquema WOTS+, com base em uma mensagem, uma␣
↪seed
    # secreta, uma seed pública e um endereço ADRS
    # Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
    # Output: WOTS+ signature sig
    def wots_sign(self, mensagem, semente_secreta, semente_publica, endereco):
        soma_verificacao = 0

        msg = base_w(mensagem, self._w, self._len_1)

        for i in range(self._len_1):
            soma_verificacao += self._w - 1 - msg[i]

        preenchimento = (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
        soma_verificacao = soma_verificacao << (8 - preenchimento)
        soma_verificacaob = soma_verificacao.to_bytes(math.ceil((self._len_2 *␣
↪math.floor(math.log(self._w, 2))) / 8), byteorder='big')
        soma_verificacaow = base_w(soma_verificacaob, self._w, self._len_2)
        msg += soma_verificacaow

        assinatura = []
        for i in range(self._len_0):
            endereco.set_chain_address(i)
            endereco.set_hash_address(0)
            chave_privada = prf(semente_secreta, endereco.copy(), self._n)
            assinatura += [self.chain(chave_privada, 0, msg[i],␣
↪semente_publica, endereco.copy())]

        return assinatura

    # Verifica a assinatura e recupera a chave pública correspondente, com base
    # em uma assinatura sig, uma mensagem, uma seed pública e um endereço ADRS
    def wots_pk_from_sig(self, assinatura, mensagem, semente_publica, endereco:␣
↪ADRS):
        soma_verificacao = 0
```

```
        endereco_pk = endereco.copy()

        msg = base_w(mensagem, self._w, self._len_1)

        for i in range(0, self._len_1):
            soma_verificacao += self._w - 1 - msg[i]

        preenchimento = (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
        soma_verificacao = soma_verificacao << (8 - preenchimento)
        soma_verificacaob = soma_verificacao.to_bytes(math.ceil((self._len_2 *␣
↪math.floor(math.log(self._w, 2))) / 8), byteorder='big')
        soma_verificacaow = base_w(soma_verificacaob, self._w, self._len_2)
        msg += soma_verificacaow

        temp = bytes()
        for i in range(self._len_0):
            endereco.set_chain_address(i)
            temp += self.chain(assinatura[i], msg[i], self._w - 1 - msg[i],␣
↪semente_publica, endereco.copy())

        endereco_pk.set_type(ADRS.WOTS_PK)
        endereco_pk.set_key_pair_address(endereco.get_key_pair_address())
        chave_publica_assinatura = hash_(semente_publica, endereco_pk, temp,␣
↪self._n)
        return chave_publica_assinatura
```

4 XMSS - Extended Merkle Signature Scheme

A classe Xmss implementa o esquema XMSS.

Também inclui métodos para converter entre diferentes formatos de assinatura (sig_wots_from_sig_xmss, auth_from_sig_xmss, sigs_xmss_from_sig_hypertree).

[6]:
```
import math

class Xmss:

    def __init__(self):

        self._n = 16
        self._w = 16
        self._h = 64
        self._d = 8
        self.wots = Wots()
```

```
        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


    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_hypertree(self, sig):
        sigs = [sig[i * (self._h_prime + self._len_0):(i + 1) * (self._h_prime␣
↪+ self._len_0)] for i in range(self._d)]
        return sigs

    # 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(2 ** z):
            adrs.set_type(ADRS.WOTS_HASH)
            adrs.set_key_pair_address(s + i)
            node = self.wots.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']

    # Gera a chave pública para o esquema HYPERTREE XMSS

    # 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

    # Gera uma assinatura para uma determinada mensagem usando o esquema␣
↪HYPERTREE XMSS.

    # 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(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.wots_sign(m, secret_seed, public_seed, adrs.copy())
        sig_xmss = sig + auth
        return sig_xmss

    # Verifica a autenticidade de uma assinatura para uma determinada mensagem␣
↪usando o esquema HYPERTREE 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.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(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
    # Por questão de eficiência, é utilizado uma HYPERTREE (árvore de árvores)␣
↪descrita na documentação
    # ================================================

    # Input: Private seed SK.seed, public seed PK.seed
    # Output: Hypertree public key PK_HT
    def hypertree_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: Mensagem M, private seed SK.seed, public seed PK.seed, tree index␣
↪idx_tree, leaf index idx_leaf
    # Output: Assinatura HT SIG_HYPERTREE
    def hypertree_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_hypertree = 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_hypertree = sig_hypertree + sig_tmp

            if j < self._d - 1:
                root = self.xmss_pk_from_sig(idx_leaf, sig_tmp, root,␣
↪public_seed, adrs.copy())

        return sig_hypertree

    # Input: Mensagem M, assinatura SIG_HYPERTREE, public seed PK.seed, tree␣
↪index idx_tree, leaf index idx_leaf, HT public key PK_HT
    # Output: Boolean
    def hypertree_verify(self, m, sig_hypertree, public_seed, idx_tree,␣
↪idx_leaf, public_key_hypertree):
        adrs = ADRS()

        sigs_xmss = self.sigs_xmss_from_sig_hypertree(sig_hypertree)
        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_hypertree:
            return True
        else:
            return False
```

5 Sphincs+

Esta é a classe principal do exercício que implementa o algoritmo Sphincs+. Abaixo, são descritas todas as etapas das diferentes funções

5.0.1 KeyGen

- Gerar uma public e uma secret seed
- Gerar uma seed para aleatoriedade
- Gerar hypertree public key
- Chave privada é a junção de todas as anteriores
- Chave publica é a junção da public seed com a hypertree public key

5.0.2 Sign

- Parse da chave
- Geramos a randomness r com o segredo prf, uma string de bytes pseudoaleatória e a mensagem
- Hash da mensagem juntamente com a randomness r
- Conversão para inteiros para obter os endereços
- Armazenamos os endereços
- Geramos a assinatura do FORS e adicionamos esta assinatura à signature
- Calculamos a chave pública FORS a partir da assinatura FORS gerada
- Assinamos da chave pública FORS utilizando a Hypertree e adicionamos à signature
- Retornamos a signature (r, a assinatura FORS e a assinatura Hypertree)

5.0.3 Verify

- Parse da assinatura
- Extraímos os componentes da chave pública
- Digest da mensagem (este será dividido em três partes)
  - msg_digest (para a assinatura FORS)
  - idx_tree_digest (para selecionar a árvore XMSS)
  - idx_leaf_digest (para selecionar a chave WOTS+ e a chave FORS correspondente dentro dessa árvore)
- Convertemo-los em inteiros para obter os índices
- Configuramos os endereços
- Obtemos private key do FORS através da assinatura
- Verificamos a private key do FORS com a Hypertree e avaliamos se a assinatura é valida

[7]:
```
from math import *

class Sphincs:

    def __init__(self):

        #Parametros

        self._n = 16 #  Parametro de segurança
        self._w = 16 #  Parametro de Winternitz (4, 16 ou 256)
        self._h = 64 #  Altura da Hypertree
        self._d = 8  #  Camadas da Hypertree
        self._k = 10 #  Numero de arvores no FORS (Forest of Random Subsets)
        self._a = 15 #  Numero de folhas de cada arvore no FORS

        self._len_1 = ceil(8 * self._n / log(self._w, 2))
        self._len_2 = floor(log(self._len_1 * (self._w - 1), 2) / log(self._w,␣
↪2)) + 1
        self._len_0 = self._len_1 + self._len_2 # n-bit values in WOTS+ sk, pk,␣
↪and signature.

        self._h_prime = self._h // self._d
        self._t = 2 ** self._a

        # XMSS e FORS
        self.xmss = Xmss()
        self.fors = Fors()


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


    # Implementação SPHINCS+

    # Gerar um par de chaves para o Sphincs+ signatures

    def keygen(self):

        # Geração dos seeds
        s_seed = os.urandom(self._n) # Para gerar chaves paraos outros␣
↪algoritmos
        s_prf = os.urandom(self._n) # para gerar randomness
        p_seed = os.urandom(self._n)

        # hypertree public key
```

```
        p_root = self.xmss.hypertree_pk_gen(s_seed, p_seed)

        return [s_seed, s_prf, p_seed, p_root], [p_seed, p_root]


    # Assinar uma mensagem com o algoritmo Sphincs
    def sign(self, m, secret_key):

        adrs = ADRS()
        adrs.set_layer_address(0)

        #Obter seeds
        s_seed = secret_key[0]
        s_prf = secret_key[1]
        p_seed = secret_key[2]
        p_root = secret_key[3]

        # Gerar a randomness r com o segredo prf, uma string de bytes␣
↪pseudoaleatória e a mensagem
        rand = os.urandom(self._n)
        r = prf_msg(s_prf, rand, m, self._n)
        sig = [r]


        # Hash da mensagem juntamente com a randomness r
        digest = hash_msg(r, p_seed, p_root, m, self.size_md + self.
↪size_idx_tree + self.size_idx_leaf)
        tmp_md = digest[:self.size_md]
        tmp_idx_tree = digest[self.size_md:(self.size_md + self.size_idx_tree)]
        tmp_idx_leaf = digest[(self.size_md + self.size_idx_tree):len(digest)]

        # Conversão para inteiros para obter os endereços
        md_int = int.from_bytes(tmp_md, 'big') >> (len(tmp_md) * 8 - self._k *␣
↪self._a)
        md = md_int.to_bytes(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))

        # Armazena os endereços
        adrs.set_tree_address(idx_tree)
        adrs.set_type(ADRS.FORS_TREE)
        adrs.set_key_pair_address(idx_leaf)

        # Assinatura do FORS e adicionamos esta assinatura à signature
        sig_fors = self.fors.fors_sign(md, s_seed, p_seed, adrs.copy())
```

```
        sig += [sig_fors]

        # Calculamos a chave pública FORS a partir da assinatura FORS gerada
        pk_fors = self.fors.fors_pk_from_sig(sig_fors, md, p_seed, adrs.copy())

        # Assinatura da chave pública FORS utilizando a Hypertree e adicionamos␣
↪à signature
        adrs.set_type(ADRS.TREE)
        sig_hypertree = self.xmss.hypertree_sign(pk_fors, s_seed, p_seed,␣
↪idx_tree, idx_leaf)
        sig += [sig_hypertree]

        return sig

    # Verificar a assinatura
    def verify(self, m, sig, public_key):

        #parse da assinatura
        # Obtemos r, a assinatura FORS e a assinatura Hypertree
        adrs = ADRS()
        r = sig[0]
        sig_fors = sig[1]
        sig_hypertree = sig[2]

        # Extraímos os componentes da chave pública
        p_seed = public_key[0]
        p_root = public_key[1]

        # Digest da mensagem
        digest = hash_msg(r, p_seed, p_root, m, self.size_md + self.
↪size_idx_tree + self.size_idx_leaf)
        # Este hash será dividido em três partes:
        # msg_digest (para a assinatura FORS)
        # idx_tree_digest (para selecionar a árvore XMSS)
        # idx_leaf_digest (para selecionar a chave WOTS+ e a chave FORS␣
↪correspondente dentro dessa árvore).
        tmp_md = digest[:self.size_md]
        tmp_idx_tree = digest[self.size_md:(self.size_md + self.size_idx_tree)]
        tmp_idx_leaf = digest[(self.size_md + self.size_idx_tree):len(digest)]

        # Convertemos as partes do digest em inteiros e bytes
        md_int = int.from_bytes(tmp_md, 'big') >> (len(tmp_md) * 8 - self._k *␣
↪self._a)
        md = md_int.to_bytes(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))

        # Configurar os endereços
        adrs.set_layer_address(0)
        adrs.set_tree_address(idx_tree)
        adrs.set_type(ADRS.FORS_TREE)
        adrs.set_key_pair_address(idx_leaf)

        # Obter PK do FORS através da assinatura
        pk_fors = self.fors.fors_pk_from_sig(sig_fors, md, p_seed, adrs)

        adrs.set_type(ADRS.TREE)
        # Verifica a PK do FORS com a Hypertree
        return self.xmss.hypertree_verify(pk_fors, sig_hypertree, p_seed,␣
↪idx_tree, idx_leaf, p_root)
```

5.1 Teste

Assiantura válida

[8]:
```
sphincs = Sphincs()
sk, pk = sphincs.keygen()
print("Secret key: ", sk)
print("\nPublic key: ", pk)

m = "Estruturas  criptográficas 2023/2024".encode()
print("\nMensagem: ", m.decode())

signature = sphincs.sign(m, sk)

if sphincs.verify(m, signature, pk):
    print("Assinatura válida")
else:
    print("Assinatura inválida")
```

```
Secret key:  [b'Q+\xe3\xad\xca\x85\xe8\x15\xf7\xc87\xa7\xa6U\xc6\xdb',
b'\x0b\x81\x19\xb3\x82\xb8\xf0\x90#c\x1a\xdc\xaf\xd4\xf7f',
b'|q\xef\xf2\x91\x99\x1dr\xd4`\xcd\xcd\xb2\x001\x17',
b'\xe52\xa6\xa4\xbeJo[\xe1(\xe1|\x86[\xa1c']

Public key:  [b'|q\xef\xf2\x91\x99\x1dr\xd4`\xcd\xcd\xb2\x001\x17',
b'\xe52\xa6\xa4\xbeJo[\xe1(\xe1|\x86[\xa1c']

Mensagem:  Estruturas  criptográficas 2023/2024
Assinatura válida
```

5.2 Teste

Assiantura inválida

[9]:
```
if sphincs.verify("Engenharia de Segurança 2023/2024".encode(), signature, pk):
    print("Assinatura válida")
else:
    print("Assinatura inválida")
```

```
Assinatura inválida
```

[0]:
```
```