TP4 EC

June 24, 2020

1 qTESLA

O qTelsa é uma familia de esquemas de assinaturas criptográficas pos-quanticas baseado no R-LWE.
Existem duas implementações do qtesla, o qTesla-p-I e o qTesla-p-III,
sendo que a principal diferença entre os dois são os paramentros utilizados,
como também a tabela CDT gerada para o gaussian sampling

1.0.1 Parametros

CDT_v2 Primeiro temos a tabela pre-computacionada CDT, que neste caso foi copiada da implementação
de referência que podemos encotrar na submissão da nist

Os parametros utilizados fazem referência ao qTesla-p-I, onde n = 1024

lambdaVal = 95
sigma = 8.5
PARAM_N = 1024
PARAM_K = 4
K_bits = 256
SEEDBYTES = 32
PARAM_N_LOG = 10
PARAM_Q = 343576577
qLog = 29
PARAM_H = 25
PARAM_B = (2**19)-1
PARAM_D = 22 Le = 554
Ls = 554 rateXOF = 168
bgen_a = 108
CHUNK_SIZE = 512
Bit_precise = 64
CDT_COLS = 78
CDT_ROWS = 2

[203]:
```
# tabela pre-computacionada CDT
CDT_v2 = [
    0x00000000, 0x00000000,
    0x0601F22A, 0x280663D4,
    0x11F09FFA, 0x162FE23D,
```

```
    0x1DA089E9, 0x437226E8,
    0x28EAB25D, 0x04C51FE2,
    0x33AC2F26, 0x14FDBA70,
    0x3DC767DC, 0x4565C960,
    0x4724FC62, 0x3342C78A,
    0x4FB448F4, 0x5229D06D,
    0x576B8599, 0x7423407F,
    0x5E4786DA, 0x3210BAF7,
    0x644B2C92, 0x431B3947,
    0x697E90CE, 0x77C362C4,
    0x6DEE0B96, 0x2798C9CE,
    0x71A92144, 0x5765FCE4,
    0x74C16FD5, 0x1E2A0990,
    0x7749AC92, 0x0DF36EEB,
    0x7954BFA4, 0x28079289,
    0x7AF5067A, 0x2EDC2050,
    0x7C3BC17C, 0x123D5E7B,
    0x7D38AD76, 0x2A9381D9,
    0x7DF9C5DF, 0x0E868CA7,
    0x7E8B2ABA, 0x18E5C811,
    0x7EF7237C, 0x00908272,
    0x7F4637C5, 0x6DBA5126,
    0x7F7F5707, 0x4A52EDEB,
    0x7FA808CC, 0x23290599,
    0x7FC4A083, 0x69BDF2D5,
    0x7FD870CA, 0x42275558,
    0x7FE5FB5D, 0x3EF82C1B,
    0x7FEF1BFA, 0x6C03A362,
    0x7FF52D4E, 0x316C2C8C,
    0x7FF927BA, 0x12AE54AF,
    0x7FFBBA43, 0x749CC0E2,
    0x7FFD5E3D, 0x4524AD91,
    0x7FFE6664, 0x535785B5,
    0x7FFF0A41, 0x0B291681,
    0x7FFF6E81, 0x132C3D6F,
    0x7FFFAAFE, 0x4DBC6BED,
    0x7FFFCEFD, 0x7A1E2D14,
    0x7FFFE41E, 0x4C6EC115,
    0x7FFFF059, 0x319503C8,
    0x7FFFF754, 0x5DDD0D40,
    0x7FFFFB43, 0x0B9E9823,
    0x7FFFFD71, 0x76B81AE1,
    0x7FFFFEA3, 0x7E66A1EC,
    0x7FFFFF49, 0x26F6E191,
    0x7FFFFFA1, 0x2FA31694,
    0x7FFFFFCF, 0x5247BEC9,
    0x7FFFFFE7, 0x4F4127C7,
```

```
    0x7FFFFFF3, 0x6FAA69FD,
    0x7FFFFFFA, 0x0630D073,
    0x7FFFFFFD, 0x0F2957BB,
    0x7FFFFFFE, 0x4FD29432,
    0x7FFFFFFF, 0x2CFAD60D,
    0x7FFFFFFF, 0x5967A930,
    0x7FFFFFFF, 0x6E4C9DFF,
    0x7FFFFFFF, 0x77FDCCC8,
    0x7FFFFFFF, 0x7C6CE89E,
    0x7FFFFFFF, 0x7E6D116F,
    0x7FFFFFFF, 0x7F50FA31,
    0x7FFFFFFF, 0x7FB50089,
    0x7FFFFFFF, 0x7FE04C2D,
    0x7FFFFFFF, 0x7FF2C7C1,
    0x7FFFFFFF, 0x7FFA8FE3,
    0x7FFFFFFF, 0x7FFDCB1B,
    0x7FFFFFFF, 0x7FFF1DE2,
    0x7FFFFFFF, 0x7FFFA6B7,
    0x7FFFFFFF, 0x7FFFDD39,
    0x7FFFFFFF, 0x7FFFF2A3,
    0x7FFFFFFF, 0x7FFFFAEF,
    0x7FFFFFFF, 0x7FFFFE1B,
    0x7FFFFFFF, 0x7FFFFF4D,
    0x7FFFFFFF, 0x7FFFFFBF,
    0x7FFFFFFF, 0x7FFFFFE9,
    0x7FFFFFFF, 0x7FFFFFF8,
    0x7FFFFFFF, 0x7FFFFFFD,
    0x7FFFFFFF, 0x7FFFFFFF
    ]
```

[204]:
```
#Parametros para qTesla-I
lambdaVal = 95
sigma = 8.5
PARAM_N = 1024
PARAM_K = 4
K_bits = 256
SEEDBYTES = 32
PARAM_N_LOG = 10
PARAM_Q = 343576577
qLog = 29
PARAM_H = 25
PARAM_B = (2**19)-1
PARAM_D = 22
Le = 554
Ls = 554
rateXOF = 168
bgen_a = 108
```

```
CHUNK_SIZE = 512
Bit_precise = 64
CDT_COLS = 78
CDT_ROWS = 2

#imports
import os
from CompactFIPS202 import SHAKE128, Keccak
from cshake import cSHAKE128
import pickle
```

1.0.2 Funções Auxiliares

poly_multi() - multiplica dois polinomios

poly_add() - soma dois polinomios

poly_subtract() - subtrai dois polinomios

poly_mod() - realiza o modulo de um polinomio

[205]:
```
def poly_multi(lista, listb):
    return [(a * b)  for a,b in zip(lista,listb)]

#add polynomial
def poly_add(lista,listb):
    return [(a + b)   for a,b in zip(lista,listb)]

#subtract polynomial
def poly_subtract(lista,listb):
    return [(a - b) for a,b in zip(lista,listb)]

def poly_mod(lista,module):
    return [(a % module)   for a in lista]
```

1.0.3 Funções

PRF1(pre-seed) -> (seedss,seeds...,seedsek,seedsa) função pseudo-aleatoria que utiliza SHAKE128 para gerar N seeds de tamanho 256//8 (k_bits)

PRF2(rand + G + msg) -> val(bytes) Semelhante a PRF1, que recebe um valor aleatorio mais uma hash G e
uma mensagem de 320 bits e gera uma hash de SEEDBYTES *(PARAM_K+3) bytes de tamanho

Check_ES(es, bound) -> True, False Função que verifica se os maiores h valores obtidos por um determinado polinomio
são menores que o limite estipulado(neste caso ambos os limites são 554)

gen_a(seed_a) -> polinomios a... Recebe uma seeda, que depois é expandida para rateXOF*bgena (168*108) com o uso do cSHAKE128
O processo produz polinomios com coeficientes aleatorios,
sendo que a implementação assume que estes já se encotram no domino NTT

gauss_sampler(seed, nonce) -> sequencia z de n gaussian samples Recebe uma seed um nonce incremental, sendo que faz o output de um polinomio(erro ou secreto), sendo que polinomio R O algoritmo implementa distribuição gaussiana discreta, baseada na tecnica da CDT (cumulative distribution table)
Sendo que é uma usada uma tabela pre-computacionada, de precisão -bit(neste caso serão 64 bits)

Enc(seed) -> pos_list,sign_list Eesta função encodifica uma hash _c como um polinomio c, sendo este polinomio representado por uma lista que contem as posições, e outra os coeficientes não-zero

spare_poly_multi(g,pos_list,sign_list) -> polinomio f multiplicação de polinomios

ySampler(seedrand,nonceD) -> polinomio y faz sampling de um polinomio y aleatorio, recebendo com seed um valor aleatorio e um counter

Hhash(v,Gm,Gt) -> hash Devolve a Hash do polinomio v com a Hash da msg e a hash de t. Usa SHAKE128

testreject(z) -> True, False A função testreject verifica se z R[B−S]

test_correct(v) -> True, False A função testcorrect verifica se ‖[wi]L‖∞<2^(d−1) −E and ‖wi‖∞ < q/2 −E

[199]:
```
#Função pseudo-aleatoria que recebe uma seed como input e
#devolve uma lista de seeds de tamanho 256/8
def PRF1(pre_seed):
    digest = SHAKE128(pre_seed, SEEDBYTES *(PARAM_K+3))
    seeds = []
    for i in range(PARAM_K+3):
        seeds.append(digest[i*SEEDBYTES:SEEDBYTES + SEEDBYTES * i])
    return seeds

#Função pseudo-aleatoria que recebe uma seed (que será um valor aleatorio + uma␣
 ↪hash G e uma mensagem de 320 bits)
#como input e devolve uma hash de tamanho SEEDBYTES *(PARAM_K+3)
def PRF2(seed):
    return SHAKE128(seed, SEEDBYTES *(PARAM_K+3))

#verifica se a soma dos  h maiores valores de um polinomio e
#verifica se são menores que os seus bounds correspondentes
#neste caso ambos são 554
def check_ES(es,bound):
```

```
    sum = 0
    lse = list(es)
    lse.sort(reverse=True)
    for i in range(0, PARAM_H):
        sum += lse[i]
    if sum > bound:
        print(sum)
        return 1
    return 0

#gera os polinomios a1,a...,ak, que recebe como input uma seed
# e mapeia-a para K polinomios ai
def gen_a(seed_a):

        #S = 0
        S = int(1)
        #b = log(q,2)/8, sendo que q ~= 2**29, logo qLog = 29
        b = (qLog+7)//8

        #pos = 0
        pos = 0
        #b' = bgen_a = 108
        b1 = bgen_a
        #colunas serão n = 1024
        cols = PARAM_N
        #rows serão k = 4
        rows = PARAM_K
        #iniciar a
        a = [[0 for i in range(cols)] for j in range(rows+1)]
        #tamanho de c
        size = (rateXOF * b1)//b
        #hash cshake
        digest = cSHAKE128(seed_a, rateXOF*b1, S.to_bytes((S.bit_length() + 7) /
↪/ 8, 'big'))
        c = []
        #c1,c2,c3,cT =
        for j in range(size):
            c.append(digest[j * b:b * j + b])

        i = 0
        while i < (PARAM_K*PARAM_N):
            if pos > size - 1:
                S +=1
                pos = 0
                b1 = 1
                digest = cSHAKE128(seed_a, rateXOF*b1, int(S).to_bytes((int(S).
↪bit_length() + 7) // 8, sys.byteorder))
```

```
                sizenew = (rateXOF * b1)//(b)
                for w in range(round(sizenew)):
                    c[w] = (digest[w * b:(b * i) + b])
            res = int.from_bytes(c[pos],"big") % 2**qLog
            if  res < PARAM_Q:
                row = abs(i//PARAM_N) +1
                col = i - PARAM_N* abs(i//PARAM_N)
                a[row][col] = res
                i+=1

            pos +=1
        a.pop(0)
        return a

#recebe uma seed (seeds,seed...,seedek) e um counter e devolve
#um polinomio
def gauss_sampler(seed, nonce):
    _nonce = nonce<<8
    z = [None] * (PARAM_N+CHUNK_SIZE)
    i = 0
    while i <= PARAM_N:
        digest = cSHAKE128(seed,CHUNK_SIZE *(Bit_precise//8),int(_nonce).
 ↪to_bytes((int(_nonce).bit_length() + 7) // 8, 'big'))
        r = [None] * CHUNK_SIZE
        for m in range(CHUNK_SIZE):
            r[m] = digest[m * Bit_precise//8:Bit_precise//8 + (Bit_precise//
 ↪8*m)]
        _nonce += 1
        for j in range(CHUNK_SIZE):
            z[j+i] = 0
            sign = int.from_bytes(r[j],"big")//(2**(Bit_precise-1))
            val = int.from_bytes(r[j],"big")


            for k in reversed(range(CDT_COLS)):
                if val >= CDT_v2[k]:
                    z[j+i] += 1
            if sign == 1:
                z[j+i] = -z[j+i]
        i = i + CHUNK_SIZE
    return z


def Enc(seed):
    D = 1
    cnt = 0
```

```
    digest = cSHAKE128(seed,rateXOF ,int(D).to_bytes((int(D).bit_length() + 7) /
↪/ 8, 'big'))
    r = [None] * rateXOF
    c = [0] * PARAM_N
    pos_list = [None] * PARAM_H
    sign_list = [None] * PARAM_H
    for u in range(rateXOF):
        r[u] = digest[u * 1:(u*1) + 1]
    i = 0
    while i < PARAM_H:
        if cnt > (rateXOF-3):
            D += 1
            cnt = 0
            for p in range(rateXOF):
                r[p] = digest[p * 1:(p*1) + 1]
        pos = (int.from_bytes(r[cnt],"big") * 2**8 + int.
↪from_bytes(r[cnt+1],"big")) % PARAM_N
        if c[pos] == 0:
            if int.from_bytes(r[cnt+2],"big") % 2 == 1:
                c[pos] = -1
            else:
                c[pos] = 1
            pos_list[i] = pos
            sign_list[i] = c[pos]
            i+=1
        cnt += 3
    return pos_list,sign_list
print(Enc(b'w'))

def spare_poly_multi(g,pos_list,sign_list):
    f = [0] * PARAM_N
    for i in range(PARAM_H):
        pos = pos_list[i]
        for j in range(pos):
            f[j] = f[j] - sign_list[i] * g[j+PARAM_N-pos]
        for j in range(pos, PARAM_N):
            f[j] = f[j] - sign_list[i] * g[j-pos]
    return f

def ySampler(seedrand,nonceD):
    b = int(log(PARAM_B,2) + 1)//8
    pos = 0
    _n = PARAM_N
    _nonceD = nonceD << 8
    digest = cSHAKE128(seedrand,b * _n, int(_nonceD).to_bytes((int(_nonceD).
↪bit_length() + 7) // 8, 'big'))
    c = []
```

```
    y = [None] * _n
    for w in range(_n):
        c.append(digest[w * b:w*b + b])
    i = 0
    while i < _n:
        if pos > _n:
            _nonceD += 1
            pos = 0
            _n = rateXOF//b
            digest = cSHAKE128(seedrand,rateXOF, bytes(_nonceD))
            for r in range(_n):
                c[r] = digest[r * b:r*b + b]

        y[i] = int.from_bytes(c[pos],"big") % 2**(int(log(PARAM_B,2) + 1))
        y[i] -= PARAM_B
        if y[i] != PARAM_B + 1:
            i+=1
        pos +=1
    return y

def Hhash(v,Gm,Gt):
    w = [None] * (PARAM_N * PARAM_K + 80)
    for i in range(PARAM_K):
        for j in range(PARAM_N):
            val = v[i][j] % 2**PARAM_D
            if val  > 2**(PARAM_D-1):
                val = val - 2**PARAM_D
            w[(i-1)*PARAM_N +j] = (v[i][j] -  val)/2**PARAM_D
    for p in range(40):
        w[PARAM_K*PARAM_N+p] = Gm[p:p+1]
        w[PARAM_K*PARAM_N+40+p] = Gt[p:p+1]
    _c = SHAKE128(pickle.dumps(w),K_bits//8)
    return _c

def testreject(z):
    valid = int(0)
    for i in range(PARAM_N):
        valid |= int(PARAM_B-Ls) - int(abs(z[i]))
    return valid >> 31

def test_correct(v):
    val = 0
    for i in range(PARAM_N):
        if v[i] > PARAM_Q/2:
            val =v[i] - PARAM_Q
        else:
            val = v[i]
```

```
        if abs(val) < PARAM_Q/2 - Le:
            t0 = 0
        else:
            t0 = 1
        if abs(val) < (1<< (PARAM_D -1))-Le:
            t1 = 0
        else:
            t1 = 1

        if (t0|t1) == 1:
            return 1
    return 0
```

keygen() -> sk,pk É gerado uma pre-seed de valor aleatorio de tamanho K_bits/8
São gerados a[0]…a[k] polinomios aleatoriamente sobre Rq usando a seed[a] gerada pela função PRF1
Depois um polinomio s é obtido através da distribuição gaussiana
Este polinomio s tem de corresponder a Check(s) < bound de s (554),
sendo check uma função que verifica se o h maiores numeros sumados do polinomio são menores que o limite especificado
Um processo semelhante é usado para obter e[0]…e[k]
A chave publica corresponderá a t[i] = a[i]s+e[i] mod q e à seed[a]
A chave privada corresponderá aos polinomios s, e, à seed[y] e à seed[a] e à hash g, obtida atráves de t[i]

Sign(msg, sk) -> signature É primeiro escolhido um polinomio y aleatoriamente, tal que y R[B] e inicializa-se os counters
São gerados a[0]…a[k] polinomios aleatoriamente sobre Rq usando a seed[a] obtida da chave privada
Para ser usado como seed, é gerado um valor aleatorio usando a função PFR2,
que recebe como input o valor de seed[y] , uma string aleatoria r, e a Hash da mensagem.
Computaciona-se _c, tal que _c é a hash do polinomio v e a tag opcional g
Para que a assinatura gerada (z = y + sc, _c) seja devolvida,
necessita de passar no teste de segurança,
que verifica que a assinatura não deixa vazar informaçoes sobre o segredo s.
A função testreject verifica se z R[B−S] e recomeça o processo de assinatura caso a condição falhe, sendo que o counter incrementa em 1
A função test_correct verifica se ‖[wi]L‖∞<2^(d−1) −E and ‖wi‖∞ < q/2 −E e recomeça o processo de assinatura caso a condição falhe, sendo que o counter incrementa em 1

Verify(msg,sig,pk) -> True, False O algoritmo de verificação recebe a mensagem, a assinatura e a chave publica
Computaciona-se pos_list, sign_list = Enc(sig_c)
Gera-se a[0]…a[k] com gen_a(pk_seed[a])
Faz a Hash de w, a mensagem e t de tal forma a verificar se correponde a sig_c

[200]:
```
#print(test_correct(random.sample(range(44555,62222),PARAM_N)))

def keygen():
    preseed = os.urandom(K_bits//8)
    seeds = PRF1(preseed)
    a = gen_a(seeds[-2])
    s = None
    counters =1
    countere =1
    while True:
        s = gauss_sampler(seeds[0],counters)
        counters +=1
        if check_ES(s,Ls) == 0:
            break
    e = [None] * PARAM_K
    t = [None] * PARAM_K
    for i in range(PARAM_K):
        while True:
            e[i] = gauss_sampler(seeds[i],countere)
            if check_ES(e[i],Le) == 0:
                break
        #print(s[i])

        t[i] = poly_mod(poly_add(poly_multi(a[i],s), e[i]), int(PARAM_Q))
    #não deve funcionar, mas tambem não chego aqui ainda
    g = SHAKE128(pickle.dumps(t),40)
    sk = (s,e,seeds[-2],seeds[-1],g)
    pk = (t,seeds[-2])
    return(sk, pk)



def Sign(msg, sk):
    s,e,seeda,seedy,g = sk
    r = os.urandom(K_bits//8)
    rand = PRF2(seedy+r+SHAKE128(msg,40))
    step1 = False
    step2 = False
    counter = 1
    while step1 == False or step2 == False:
        y = ySampler(rand,counter)
        a = gen_a(seeda)
        v = [None] * PARAM_K
        for i in range(PARAM_K):
            v[i]= poly_mod(poly_multi(a[i],y), PARAM_Q)
        _c = Hhash(pickle.dumps(v),SHAKE128(msg,40),g)
        pos_list,sign_list = Enc(_c)
```

```
        sc = spare_poly_multi(s,pos_list,sign_list)
        z = [None] * PARAM_N
        for j in range(PARAM_N):
            z[j] = y[j] + sc[j]
        if testreject(z) !=0:
            counter += 1
            continue
        else:
            step1 = True
        w = [None] * PARAM_K
        for k in range(PARAM_K):
            w[k] =␣
↪poly_subtract(v[k],poly_mod(spare_poly_multi(e[k],pos_list,sign_list),PARAM_Q))
            if test_correct(w[k]) != 0:
                counter += 1
                continue
            else:
                step2 = True
    return z,_c

def Verify(msg,sig,pk):
    z,_c = sig
    t,seeda = pk
    pos_list, sign_list = Enc(_c)
    a = gen_a(seeda)
    w = [None] * PARAM_K
    for k in range(PARAM_K):
        w[k] =␣
↪poly_subtract(poly_multi(a[k],z),poly_mod(spare_poly_multi(t[k],pos_list,sign_list),PARAM_Q

    if testreject(z) != 0 or _c != Hhash(w,SHAKE128(msg,40),SHAKE128(pickle.
↪dumps(t),40)):
        return False
    return True
```

[202]:
```
sk, pk = keygen()
sig = Sign(b'wdwdwdwd',sk)
Verify(b'wdwdwdwd',sig,pk)
```

```
1950
1950
1950
1950
1950
1950
```

```
1950
1950
1950
```

```
      ␣
↪---------------------------------------------------------------------------
```

```
      KeyboardInterrupt                         Traceback (most recent call␣
↪last)

      <ipython-input-202-42153b358afa> in <module>
----> 1 sk, pk = keygen()
      2 sig = Sign(b'wdwdwdwd',sk)
      3 Verify(b'wdwdwdwd',sig,pk)


      <ipython-input-196-95e446de5c78> in keygen()
      114     countere =Integer(1)
      115     while True:
--> 116         s = gauss_sampler(seeds[Integer(0)],counters)
      117         counters +=Integer(1)
      118         if check_ES(s,Ls) == Integer(0):


      <ipython-input-196-95e446de5c78> in gauss_sampler(seed, nonce)
       99             for k in reversed(range(CDT_COLS)):
      100                 if val >= CDT_v2[k]:
--> 101                     z[j+i] += Integer(1)
      102             if sign == Integer(1):
      103                 z[j+i] = -z[j+i]


      src/cysignals/signals.pyx in cysignals.signals.python_check_interrupt()


      KeyboardInterrupt:
```

2 SPHINCS+

Esta implementação do Esquema SPHINCS+ teve como referência e base os seguintes documentos.

- [1] https://sphincs.org/data/sphincs+-round2-specification.pdf
- [2] https://www.ietf.org/rfc/rfc8391.html
- [3] https://github.com/GaloisInc/cryptol-specs/blob/master/Primitive/Asymmetric/Signature/SphincsPlus

3 INTRODUÇÃO

Inicialmente, o Esquema SPHINCS+ é uma das propostas de esquema HBS(hash based signatures), ou melhor, esquemas de assinatura digital que apenas usam funçoes de hash. Este esquema traz consigo enumeros benefícios, sendo um deles, que a sua segurança depende de um conjunto limitado de assunções. Essencialmente, nos esquemas HBS existe um limite superir ao numero de mensagens que uma chave privada pode assinar, caso se verifique que este limite tenha sido ultrapassado a chave é revelada. Além disto, o SPHINCS pode ser visto como uma variante do esquema XMSS, que nao usam estado. A geraçao dos pares de chaves publica e privada é um "pouco" diferente dos esquemas de chave publica, pois este produz um numero muito grande de pares de chaves com numero suficiente para todo o tempo de vida da instancia. De modo a armazenar e gerir esses pares de chaves é então utilizado estruturas de dados nomeadas como Árvores de Merkle.

Então em relação ás Estrutura de dados usadas por este esquema, para definir uma arvore de Merkle(MSS) é necessario começar por uma OTS(one-time signature), neste caso será o WOTS+. Basicamente é um esquema de assinaturas em que cada par de chaves apenas pode ser usado para assinar uma mensagem e por consequente um unica assinatura.

No caso do XMS, os elementos de chaves e de assinatura colocam-se ao nivel do WOTS e do XMS. Ainda no XMS, existe um par de chaves(publica e privada) que podem ser usadas para produzir assinaturas de varias mensagens.

Com isto, é possivel criar uma ligaçao entre o XMS e de OTS com a Arvore de Merkle(MSS).

4 Índice

Portanto, nesta implementação seguimos a organização presente no próprio documento do SPHINCS+ que é a seguinte:

- Implementação do Hash Function Address Scheme - CLASSE ADRS;
- WOTS+ - o OTS usado no SPHINCS+;
- XMSS - o MTS usado no SPHINCS+;
- FORS e STF
- SPHINCS+

Os esquemas descritos neste documento randomizam cada chamada de função hash. Isso significa que, além do digest da mensagem inicial, uma chave diferente e uma máscara de bit diferente são usadas para cada chamada de função de hash. Esses valores são gerados pseudo-aleatoriamente usando uma função pseudo-aleatória que usa uma chave SEED e um endereço ADRS de 32 bytes como entrada e gera um valor n-byte, em que n é o parâmetro de segurança.

[1]:
```
#!/usr/bin/env python3
import os
import random
#from bytes_utils import xor, chunkbytes,ints_from_4bytes, ints_to_4bytes
#from math import ceil, log
import math
import hashlib
```

5 BYTES UTILS

[2]:
```python
def xor(b1, b2):
    """Expects two bytes objects of equal length, returns their XOR"""
    assert len(b1) == len(b2)
    return bytes([x ^ y for x, y in zip(b1, b2)])


def chunkbytes(a, n):
    return [a[i:i+n] for i in range(0, len(a), n)]


def ints_from_4bytes(a):
    for chunk in chunkbytes(a, 4):
        yield int.from_bytes(chunk, byteorder='little')


def ints_to_4bytes(x):
    for v in x:
        yield int.to_bytes(v, length=4, byteorder='little')
```

6 Strings of Base w Numbers [RFC8391]

Uma sequência de bytes pode ser considerada como uma sequência de números base w, ou seja, números inteiros no conjunto {0,. . . , w-1}.

A correspondência é definida pela função base_w (X, w, out_len) da seguinte maneira. Seja X uma string len_X-byte e w seja um elemento do conjunto {4, 16, 256}, então base_w (X, w, out_len) gera uma matriz de inteiros out_len entre 0 e w - 1. O comprimento out_len necessita de ser menor ou igual a 8 ∗ len_X / log (w).

[3]:
```python
#Adaptado de [1] na Pag 8
# Input: len_X-byte string X, int w, output length out_len
# Output: out_len int array basew
# Computing the base-w representation
def base_w(x, w, out_len):
    vin = 0
    vout = 0
    total = 0
    bits = 0
    basew = []

    for consumed in range(0, out_len):
        if bits == 0:
            total = x[vin]
            vin += 1
            bits += 8
```

```
        bits -= math.floor(math.log(w, 2))
        basew.append((total >> bits) % w)
        vout += 1

    return basew
```

6.1 Classe ADRS

Começamos por implementar a estrutura ADRS(Hash Function Address Scheme)[Página 11] e [Página] Existem cinco tipos diferentes de endereços para os diferentes casos no SPHINCS+: - 1º tipo é usado para os hashes nos esquemas WOTS+; - 2º Tipo é usado para compactação da chave pública WOTS +; - 3º Tipo é usado para hashes na construção principal da árvore Merkle - 4º Tipo é usado para as hashes na árvore Merkle no FORS - 5º Tipo é usado para a compressão das raízes das árvores do FORS.

Os esquemas nas próximas duas seções usam dois tipos de funções de hash parametrizadas pelo parâmetro de segurança n. Para as construções da árvore de hash, é usada uma função de hash que mapeia uma chave de n bytes e entradas de 2 n bytes para saídas de n bytes. Para randomizar essa função, são necessários 3n bytes - n bytes para a chave e 2n bytes para uma máscara de bit. Para as construções do esquema OTS, é utilizada uma função hash que mapeia chaves de n bytes e entradas de n bytes para saídas de n bytes. Para randomizar essa função, são necessários 2n bytes - n bytes para a chave en bytes para uma máscara de bit. Consequentemente, são necessários três endereços para a primeira função e dois endereços para a segunda.

[6]:
```
#Retirado e adaptado de [1]
#Classe responsavel pelos endereços ADRS, sendo estes valores de 32 bytes
#que segue a estrutura acima. Possui ainda metodos definidos para manipular o␣
 ↪endereço.
class ADRS:

    #TIPOS de endereços
    #Endereço usado para as hashs no esquema WOTS+
    WOTS_HASH = 0
    #Tipo é usado para compactação da chave pública WOTS +
    WOTS_PK = 1
    #Tipo é usado para hashes na construção principal da árvore Merkle
    TREE = 2
    #Tipo é usado para as hashes na árvore Merkle no FORS
    FORS_TREE = 3
    #Tipo é usado para a compressão das raízes das árvores do FORS.
    FORS_ROOTS = 4

    #Construtor da classe ADRS
    def __init__(self):
        #TIPO DE ENDEREÇO
        self.type = 0
        #LAYER - ALTURA DA ARVORE
        self.layer = 0
```

```
        #ENDEREÇO TREE
        self.tree_address = 0
        #Palavras para as quais a função pode mudar dependendo do ADRS.type
        #Somente o endereço da árvore (ou seja, o índice de uma subárvore␣
↪específica na árvore principal)
        #é muito longo para caber numa única palavra: para isso, reservamos␣
↪três palavras.
        self.word_1 = 0
        self.word_2 = 0
        self.word_3 = 0

    #Metodo para definir o tipo através do valor
    def set_type(self, val):
        self.type = val
        self.word_2 = 0
        self.word_3 = 0
        self.word_1 = 0

    #Metodo Copy - Atribuir Valores
    def copy(self):
        #Acompanhar contexto atual
        adrs = ADRS()
        adrs.layer = self.layer
        adrs.tree_address = self.tree_address
        adrs.type = self.type
        #Atzar ADRRESS
        adrs.word_1 = self.word_1
        adrs.word_2 = self.word_2
        adrs.word_3 = self.word_3
        return adrs

    def int_to_basew(self, x, base):
        for _ in range(self.l1):
            yield x % base
            x //= base

    #Converter para uma cadeia de bytes big-endian
    def to_bin(self):
        #https://stackoverflow.com/questions/846038/
↪convert-a-python-int-into-a-big-endian-string-of-bytes
        adrs = self.layer.to_bytes(4, byteorder='big')
        #Sem especificar tamanho
        #https://docs.python.org/3/library/stdtypes.html#int.to_bytes
        adrs = adrs + self.tree_address.to_bytes(12, byteorder='big')
        adrs = adrs + self.type.to_bytes(4, byteorder='big')
        #AS 3 WORDS
        adrs = adrs + self.word_1.to_bytes(4, byteorder='big')
```

```
        adrs = adrs + self.word_2.to_bytes(4, byteorder='big')
        adrs = adrs + self.word_3.to_bytes(4, byteorder='big')
        return adrs

#

#1-------------------------------------
    #Definir o Endereço Par de Chaves
    def set_key_pair_address(self, val):
        self.word_1 = val
    def get_key_pair_address(self):
        return self.word_1
#-------------------------------------
    #Definir os Endereços da LAYER e TREE
    def set_layer_address(self, val):
        self.layer = val
    def set_tree_address(self, val):
        self.tree_address = val
#2-------------------------------------
    #Definir Chain Address e altura da árvore
    def set_chain_address(self, val):
        self.word_2 = val
    def set_tree_height(self, val):
        self.word_2 = val
    def get_tree_height(self):
        return self.word_2
#3-------------------------------------
    #Definir Endereço HASH e o index da árvore
    def set_hash_address(self, val):
        self.word_3 = val
    def set_tree_index(self, val):
        self.word_3 = val
    def get_tree_index(self):
        return self.word_3
```

Essencialmente, todas as funções precisam acompanhar o contexto atual, atualizando os endereços após cada chamada de hash.

Um endereço está estruturado da seguinte maneira: - Ele começa sempre com uma LAYER ADDRESS de uma palavra(WORD) nos bits mais significativos - Depois, seguido por um TREE ADDRESS de três palavras. Esses endereços descrevem a posiçao de uma arvore dentro da hiperarvore.

A LAYER ADDRESS descreve a altura de uma árvore dentro da hiperarvore apartir da altura zero para arvores na camada inferior.

O TREE ADDRESS descreve a posição de uma árvore dentro de uma camada de uma árvore múltipla começando com o índice zero da árvore mais à esquerda. A próxima palavra define o tipo de endereço.

Portanto, o tipo de endereço é definido como 0 para um hash WOTS+ ADDRESS. 1 para a compactaçao da chave pública WOTS+, 2 para Endereço de arvore de hash, 3 para endereço de FORS e 4 para compactaçao das raizes da arvore do FORS.

7 Member Functions (Functions set, get)

Estas funções como designadas no documento do SPHINCS+ assumem-se para simplificar as descriçoes do algoritmo.

Se uma estrutura de dados complexa, como uma chave pública PK, contiver uma variável X, PK.getX () retornará o valor de X para essa chave pública. Por conseguinte, PK.setX (Y) define a variável X em PK para o valor mantido por Y.

[7]:
```
# Baseado numa outra implementação SPHINCS 256
# Pagina

##Função Responsavel por calcular
def cal_var():
    _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_prime = _h // _d
    _t = 2 ** _a

#FUNÇOES AUXILIARES, APENAS PARA SIMPLIFICAR IMPLEMENTAÇÃO
def set_hypertree_height( val):
    _h = val
    cal_var()
def set_h( val):
    _h = val
    cal_var()
def get_hypertree_height():
    return _h
def set_hypertree_layers( val):
    _d = val
    cal_var()
def set_d( val):
    _d = val
    cal_var()
def get_hypertree_layers():
    return _d
def set_fors_trees_number( val):
    _k = val
    cal_var()
def set_k( val):
    _k = val
    cal_var()
def get_fors_trees_number():
```

```
    return _k
def set_fors_trees_height( val):
    _a = val
    cal_var()
def set_a( val):
    _a = val
    cal_var()
def get_fors_trees_height():
    return _a

# Valor de Segurança
def set_security( val):
    _n = val
    cal_var()
# Valor de Winternitz que pertence ao conjunto {4,16,256}
def setWinternitz( val):
    if val == 4 or val == 16 or val == 256:
        _w = val
    cal_var()
def get_winternitz():
    return _w
```

8 Cryptographic (Hash) Function Families

8.1 Tweakable Hash Functions (Functions T_l, F, H)

Uma função de hash tweakable usa uma seed pública $PK.seed$ e informações de contexto na forma de um endereço ADRS, além da mensagem. Solicita cada par de chaves e posição na estrutura em árvore virtual do SPHINCS +, independentes um do outro.

[8]:
```
# FUNÇÕES TWEAKABLE HASH
def hash(seed, adrs, value, digest_size):
    m = hashlib.sha256()
    m.update(seed)
    m.update(adrs.to_bin())
    m.update(value)
    hashed = m.digest()[:digest_size]
    return hashed
#Para compactar a mensagem a ser assinada,
#o SPHINCS + usa uma função de hash com chave adicional Hmsg
#que pode processar mensagens de tamanho arbitrário:
def hash_msg(r, public_seed, public_root, value, digest_size):
    #Hmsg : B^n * B^n * B^n * B^* -> B^m
    m = hashlib.sha256()
    m.update(r)
    m.update(public_seed)
    m.update(public_root)
```

```
    m.update(value)
    hashed = m.digest()[:digest_size]

    i = 0
    while len(hashed) < digest_size:
        i += 1
        m = hashlib.sha256()
        m.update(r)
        m.update(public_seed)
        m.update(public_root)
        m.update(value)
        m.update(bytes([i]))
        hashed += m.digest()[:digest_size - len(hashed)]

    return hashed

#O SPHINCS + utiliza uma função pseudo-aleatória PRF para geração de chave␣
 ↪pseudo-aleatória
def prf(secret_seed, adrs, digest_size):
     #PRF:B^n * B^32 -> B^n
    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')
#Além disso, o SPHINCS + usa uma função pseudo-aleatória PRFmsg
#para gerar aleatoriedade para a compactação de mensagens:
def prf_msg(secret_seed, opt, m, digest_size):
    #PRFmsg:B^n * B^n * B^* -> B^n
    random.seed(int.from_bytes(secret_seed + opt + hash_msg(b'0', b'0', b'0',␣
 ↪m, digest_size * 2), "big"))
    return random.randint(0, 256 ** digest_size - 1).to_bytes(digest_size,␣
 ↪byteorder='big')
```

9 WOTS+ One-Time Signatures

WOTS + é um esquema OTS; enquanto uma chave privada pode ser usada para assinar qualquer mensagem, cada chave privada NÃO DEVE ser usada para assinar mais de uma única mensagem. Em particular, se uma chave privada for usada para assinar duas mensagens diferentes, o esquema se tornará inseguro.

Pressupõem-se que o esquema é usado como uma sub-rotina dentro de um esquema de ordem superior e não é suficiente para uma implementação autônoma do WOTS+. - 1º Parametros - 2º Função de encadeamento - 3º Geração e Assinatura de Chaves - 4º Calcular uma chave pública WOTS+ a partir de uma assinatura WOTS+.

9.1 WOTS+ PARAMETROS

WOTS+ utiliza os seguintes parametros que recebem valores positivos: - n: Corresponde ao parametro de segurança, é o tamanho da mensagem e o comprimento de uma chave privada,chave pública ou elemento de assinatura em bytes. - w: Corresponde ao parametro Winternitz, que é um elemento do conjunto {4, 16, 256}.

Estes parametros sao usados para computar valores len, len1 e len2 - *len*: Corresponde ao número de elementos de cadeia de n-byte-string numa chave privada WOTS +, chave pública e assinatura. É calculado como len = len1 + len2, com - len1 = [(8n)/log(w)] - len2 = [(log (len1(w − 1)))/log(w)]+1

Além disto, o parametro n é o mesmo que o n do SPHINCS+. Determina o comprimento de entrada e saida da função de hash ajustável usada para o Wots+. Também determina o tamanho das mensagens que podem ser processadas pelo algoritmo de assinatura WOTS+

O parâmetro w pode ser escolhido no conjunto {4, 16, 256}. Um valor maior de w resulta em assinaturas mais curtas, mas em operações mais lentas; não afeta a segurança.

9.2 WOTS+ Chaining Function (Function chain)

A função Chaining calcula uma iteração de F numa entrada de n bytes, usando um endereço hash ADRS de WOTS+ e uma seed pública *PK.seed*. O endereço ADRS DEVE ter as sete primeiras palavras de 32 bits definidas para codificar o endereço dessa cadeia. Em cada iteração, o endereço é atualizado para codificar a posição atual na cadeia antes que o ADRS seja usado para processar a entrada por F. A função encadeamento recebe como entrada uma string de n bytes X; um índice inicial i, uma série de etapas s, além de ADRS e PK.seed. A função encadeamento retorna como saída o valor obtido pela iteração F por s vezes na entrada X

[9]:
```
'''
#Pagina 14
chain(X, i, s, PK.seed, ADRS) {
    if ( s == 0 ) {
        return X;
    }
    if ( (i + s) > (w - 1) ) {
        return NULL;
    }

byte[n] tmp = chain(X, i, s - 1, PK.seed, ADRS);

ADRS.setHashAddress(i + s - 1);
tmp = F(PK.seed, ADRS, tmp);
return tmp;
}
'''
def chain( x, i, s, public_seed, 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
```

9.3 WOTS+ Private Key (Function wots_SKgen)

A chave privada do WOTS +, denotada por sk (s para segredo), é uma matriz de comprimento de strings de n bytes. Esta chave privada NÃO DEVE ser usada para assinar mais de uma mensagem. Essa chave privada é usada apenas implicitamente. Cada sequência de n-bytes na chave privada WOTS + é derivada de uma semente secreta *SK.seed* que faz parte da chave secreta SPHINCS+ e de um endereço ADRS WOTS+ usando pseudorandom function PRF. A mesma semente secreta é usada para gerar todos os valores de chave secreta no SPHINCS+. O endereço usado para gerar a sequência de enésimo n-byte de sk DEVE codificar a posição da enésima cadeia de hash dessa instância do WOTS+ dentro da estrutura SPHINCS+.

[10]:
```
'''
#Input: secret seed SK.seed, address ADRS
#Output: WOTS+ private key sk
wots_SKgen(SK.seed, ADRS) {
    for ( i = 0; i < len; i++ ) {
        ADRS.setChainAddress(i);
        ADRS.setHashAddress(0);
        sk[i] = PRF(SK.seed, ADRS);
    }
    return sk;
}
'''
# Input: secret seed SK.seed, address ADRS
# Output: WOTS+ private key sk
def wots_sk_gen( secret_seed, adrs):
    sk = []
    for i in range(0, _len_0):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk.append(prf(secret_seed, adrs.copy(), _n))
    return sk
```

9.4 WOTS+ Public Key Generation (Function wots_PKgen)

Um par de chaves WOTS+ define uma estrutura virtual que consiste em cadeias de hash len de comprimento w. Cada um dos len stings de n-bytes na chave privada define o nó inicial de uma cadeia de hash. A chave pública é o hash ajustável dos nós finais dessas cadeias de hash. Para calcular as cadeias de hash, a função de encadeamento(chain) é usada. Um endereço hash WOTS + ADRS e uma semente PK.seed devem ser fornecidos pelo algoritmo de chamada, bem como uma semente secreta *SK.seed*. O endereço ADRS DEVE codificar o endereço do par de

chaves WOTS+ dentro da estrutura SPHINCS+. Portanto, um algoritmo WOTS + NÃO DEVE manipular nenhuma parte do ADRS além das últimas três palavras de 32 bits.

[11]:
```
def wots_pk_gen( secret_seed, public_seed, adrs):
    wots_pk_adrs = adrs.copy()# copiar endereço para criar o endereço da chave␣
 ↪pública do OTS
    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(), _n)
        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, _n)
    return pk
```

9.5 WOTS+ Signature Generation (Function wots_sign)

Uma assinatura WOTS+ é uma matriz de comprimento *len* de strings de n-bytes. A assinatura WOTS+ é gerada através do mapeamento de uma mensagem M para números inteiros entre 0 e $w-1$. Para esse fim, a mensagem é transformada em len1 $base-w$ usando a função $base_w$. Em seguida, um CheckSum sobre M é computado e anexado à mensagem transformada como números len2 $base-w$ usando a função $base_w$.

O CheckSum pode atingir um valor inteiro máximo de len1 · (w - 1) e, portanto, depende dos parâmetros n e w.

Cada um dos inteiros base-w é usado para selecionar um nó de uma cadeia de hash diferente. A assinatura é formada concatenando os nós selecionados. Um endereço hash WOTS + ADRS, uma semente pública PK.seed e uma semente secreta SK.seed devem ser fornecidas pelo algoritmo de chamada.

Com isto, um algoritmo WOTS + NÃO DEVE manipular nenhuma parte do ADRS além das últimas três palavras de 32 bits.O $PK.seed$ usado aqui é uma informação pública também disponível para um verificador, enquanto a semente secreta $SK.seed$ é uma informação privada.

9.5.1 Formato:

```
--------------------
|                   |
|     sig_ots[0]    | n-bytes
|                   |
|-------------------|
|        ...        |
|-------------------|
|                   |
|   sig_ots[len-1]  | n-bytes
```

```
|                   |
|-------------------|
```

Formatos de dados da assinatura

[12]:
```python
# Input: Message M, secret seed SK.seed, public seed PK.seed, address ADRS
# Output: WOTS+ signature sig
def wots_sign( m, secret_seed, public_seed, adrs):
    csum = 0
    # converte messagem para base-w
    msg = base_w(m, _w, _len_1)
    # Calcula o checksum
    for i in range(0, _len_1):
        #checksum pode atingir um valor maximo inteiro de len1 (w-1)
        csum += _w - 1 - msg[i]
    # converter Checksum para base-w
    padding = (_len_2 * math.floor(math.log(_w, 2))) % 8 if (_len_2 * math.
 ↪floor(math.log(_w, 2))) % 8 != 0 else 8
    csum = csum << (8 - padding)
    csumb = csum.to_bytes(math.ceil((_len_2 * math.floor(math.log(_w, 2))) /␣
 ↪8), byteorder='big')
    csumw = base_w(csumb, _w, _len_2)
    msg += csumw
    sig = []
    for i in range(0, _len_0):
        adrs.set_chain_address(i)
        adrs.set_hash_address(0)
        sk = prf(secret_seed, adrs.copy(), _n)
        sig += [chain(sk, 0, msg[i], public_seed, adrs.copy())]

    return sig
```

9.6 WOTS+ Compute Public Key from Signature (Function wots_pkFromSig)

O SPHINCS+ usa a verificação implícita de assinatura para o WOTS+. Para verificar uma assinatura *sig* WOTS+ numa mensagem M, o verificador calcula um valor de chave pública WOTS+ a partir da assinatura. Isso pode ser feito "completando" os cálculos da cadeia(chain) a partir dos valores da assinatura, usando os valores $base - w$ do hash da mensagem e a seu *checksum*.

O resultado de $wots_pkFromSig$ precisa ser verificado. Numa versão autônoma, isso seria feito por comparação simples. Como aqui é pa ser no SPHINCS+, o valor de saída é verificado usando-o para calcular uma chave pública SPHINCS+.

Um endereço de hash WOTS+ ADRS e uma seed pública $PK.seed$ são fornecidas pelo algoritmo de chamada. O endereço codificará o endereço do par de chaves WOTS+ na estrutura SPHINCS+. Portanto, um algoritmo WOTS+ NÃO DEVE manipular nenhuma parte do ADRS além das últimas três palavras de 32 bits. É de destacar que o PK.seed usado aqui são informações públicas também disponíveis para um verificador.

[13]:
```python
#Input: Message M, WOTS+ signature sig, address ADRS, public seed PK.seed
#Output: WOTS+ public key pk_sig derived from sig
def wots_pk_from_sig( sig, m, public_seed, adrs):
    csum = 0
    wots_pk_adrs = adrs.copy()

    # converte messagem para base-w
    msg = base_w(m, _w, _len_1)

    # calcula checksum
    for i in range(0, _len_1):
        csum += _w - 1 - msg[i]

    # converte checksum to base w
    padding = (_len_2 * math.floor(math.log(_w, 2))) % 8 if (_len_2 * math.
↪floor(math.log(_w, 2))) % 8 != 0 else 8
    csum = csum << (8 - padding)
    csumb = csum.to_bytes(math.ceil((_len_2 * math.floor(math.log(_w, 2))) /␣
↪8), byteorder='big')
    csumw = base_w(csumb, _w, _len_2)
    msg += csumw

    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.copy())

    wots_pk_adrs.set_type(ADRS.WOTS_PK)
    wots_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())
    pk_sig = hash(public_seed, wots_pk_adrs, tmp, _n)
    return pk_sig
```

10 The SPHINCS+ Hypertree [p18]

10.1 XMSS

Aqui iremos abordar a Hypertree do SPHINCS+, como funciona, e a sua respetiva implementação. (Baseada nos algoritomos fornecidos pelo Documento[1]) Então para construir a Hypertree, o WOTS + é combinado com uma árvore binária de hash, obtendo uma versão fixa do comprimento da entrada do eXtended Merkle Signature Scheme (XMSS).

- 1º XMSS - Parametros,
- 2º Chave Privada,
- 3º Tree HASH,
- 4º Geração Chave publica,
- 5º Assinatura
- 6º Calcular Chave publica através da Assinatura

XMSS é um método para assinar um número potencialmente grande, mas fixo de mensagens. É baseado no esquema de assinatura Merkle. Autentica Chaves publicas WOTS+ usando altura de arvore binaria.

Portanto, um par de chaves XMSS para altura h' pode ser usado para assinar 2^h' mensagens diferentes. Cada nó na árvore binária é um valor de n-bytes, que é o hash ajustável da concatenação dos seus dois nós filhos. As folhas são as chaves públicas do WOTS+. ### Chave Publica XMSS A chave pública XMSS é o nó raiz da árvore. ### Chave Privada XMSS No SPHINCS+, a chave secreta XMSS é a unica seed secreta usada para gerar todas as chaves secretas WOTS+.

10.2 XMSS Parametros

XMSS tem os seguintes parametros: - h': a altura (número de níveis - 1) da árvore. - n: o comprimento em bytes das mensagens e de cada nó. - w: o parâmetro Winternitz, conforme definido para WOTS + na seção anterior

10.3 TreeHash (Function treehash)

Para o cálculo dos nós de n-bytes internos de uma árvore Merkle, a sub-rotina treehash (Algoritmo 7) aceita uma seed secreta $SK.seed$, uma seed pública $PK.seed$, um número inteiro sem sinal s (o índice inicial), um número inteiro sem sinal z (a altura do nó de destino) e um endereço ADRS que codifica o endereço da árvore que o contém. Para a altura de um nó dentro de uma árvore, a contagem começa com as folhas na altura zero. O algoritmo treehash retorna o nó raiz de uma árvore de altura z com a folha mais à esquerda sendo o WOTS+ pk no índice s.

É NECESSÁRIO que s, quer dizer que a folha no índice s é uma folha mais à esquerda de uma subárvore de altura z. Caso contrário, o algoritmo falhará, pois computaria nós inexistentes. O algoritmo treehash descrito aqui usa uma pilha contendo até (z −1) nós, com as funções usuais da pilha push () e pop (). Além disso, assumimos que a altura de um nó (um número inteiro não assinado) é armazenado ao lado do valor de um nó (uma sequência de n bytes) na pilha.

[14]:
```
# 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( secret_seed, s, z, public_seed, 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'] +␣
↪node, _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']
```

10.4 XMSS Public Key Generation (Function xmss_PKgen)

Como já referimos na introdução do XMSS, a chave pública XMSS PK é a raiz da árvore de hash binária. Esta raiz é calculada usando treehash, a função acima. A geração de chave pública usa uma semente secreta $SK.seed$, uma semente pública $PK.seed$ e um endereço ADRS.

[15]:
```
'''
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: XMSS public key PK
xmss_PKgen(SK.seed, PK.seed, ADRS) {
pk = treehash(SK.seed, 0, h', PK.seed, ADRS)
return pk;
'''
def xmss_pk_gen( secret_seed, public_key, adrs):
    pk = treehash(secret_seed, 0, _h_prime, public_key, adrs.copy())
    return pk
```

10.5 XMSS Signature Generation (Function xmss_sign)

Para calcular a assinatura XMSS de uma mensagem M no contexto do SPHINCS + são necessarios os seguintes parametros:

- a semente secreta $SK.seed$,
- a semente pública $PK.seed$,
- o índice idx do par de chaves WOTS + a ser usado
- endereço ADRS da instância XMSS. Primeiro, uma assinatura WOTS+ da hash da mensagem é calculada usando a instância WOTS+ no índice idx. Em seguida, o caminho da autenticação é calculado. Os valores do nó do caminho de autenticação podem ser calculados de qualquer forma.

[16]:
```
# 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( m, secret_seed, idx, public_seed, adrs):
```

```
    auth = []
    # //build authentication path
    for j in range(0, _h_prime):
        ki = math.floor(idx // 2 ** j)
        if ki % 2 == 1:  # XORING idx/ 2**j with 1
            ki -= 1
        else:
            ki += 1

        auth += [treehash(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
```

10.6 XMSS Compute Public Key from Signature (Function xmss_pkFromSig)

Como já referido, o SPHINCS+ utiliza verificação implícita de assinatura de assinaturas XMSS. Uma assinatura XMSS é usada para calcular uma chave pública XMSS candidata, ou seja, a raiz da árvore. Isso é usado em cálculos adicionais (assinatura da árvore acima) e verificado implicitamente pelo resultado desse cálculo. Portanto, esta especificação não contém um método xmss_verify, mas o método xmss_pkFromSig.

O método xmss_pkFromSig recebe:

- uma mensagem de n-bytes M,
- uma assinatura XMSS SIGXMSS,
- um índice de assinatura idx,
- uma semente pública $PK.seed$,
- um endereço ADRS.

Primeiro, wots_pkFromSig é usado para calcular uma chave pública WOTS + candidata. Por sua vez, é usado junto com o caminho de autenticação para calcular um nó raiz que é retornado.

[17]:
```
# 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( 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)
```

```
    node_0 = wots_pk_from_sig(sig, m, public_seed, adrs.copy())
    node_1 = 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)
            node_1 = hash(public_seed, adrs.copy(), node_0 + auth[i], _n)
        else:
            adrs.set_tree_index((adrs.get_tree_index() - 1) // 2)
            node_1 = hash(public_seed, adrs.copy(), auth[i] + node_0, _n)

        node_0 = node_1

    return node_0
```

10.7 HT: The Hypertee

O SPHINCS + hypertree HT é uma variante do XMSSMT. É essencialmente uma árvore de certificação de instâncias XMSS. Um HT é uma árvore de várias camadas de árvores XMSS. As árvores nas camadas superior e intermediária são usadas para assinar as chaves públicas, ou seja, os nós raiz, das árvores XMSS na respectiva camada seguinte abaixo. As árvores na camada mais baixa são usadas para assinar as atuais mensagens, que são chaves públicas do FORS no SPHINCS+, que será o proximo esquema a ser abordado neste trabalho. Ainda todas as árvores XMSS no HT têm a mesma altura.

10.7.1 HT Parametros

Além dos parametros do XMSS, o HT tambem precisa da altura da hiperárvore h, o número de camadas de árvore d, especificado como um valor inteiro que divide h sem o resto. A mesma altura da árvore $h' = h/d$ e o mesmo parâmetro Winternitz w são usados para todas as camadas da árvore.

10.7.2 HT Key Generation (Function ht_PKgen)

A chave privada HT é a semente secreta $SK.seed$ usada para gerar todas as chaves privadas WOTS+ dentro da estrutura virtual estendida pelo HT. A chave pública HT é a chave pública (nó raiz) da única árvore XMSS na camada superior. A geração de chave pública assume como entrada uma semente privada e uma pública.

[18]:
```
'''
# Input: Private seed SK.seed, public seed PK.seed
# Output: HT public key PK_HT
ht_PKgen(SK.seed, PK.seed){
```

```
    ADRS = toByte(0, 32);
    ADRS.setLayerAddress(d-1);
    ADRS.setTreeAddress(0);
    root = xmss_PKgen(SK.seed, PK.seed, ADRS);
return root;
'''
# Input: Private seed SK.seed, public seed PK.seed
# Output: HT public key PK_HT
def ht_pk_gen(secret_seed, public_seed):
    adrs = ADRS()
    adrs.set_layer_address(_d - 1)
    adrs.set_tree_address(0)
    root = xmss_pk_gen(secret_seed, public_seed, adrs.copy())
    return root
```

10.7.3 HT Signature Generation (Function ht_sign)

Uma assinatura HT SIGHT é uma cadeia de bytes de comprimento $(h + d = len) * n$, que consiste em d assinatura XMSS (de (h/d+len)∗n bytes cada).

Para calcular uma assinatura HT SIGHT de uma mensagem M usando, ht_sign é preciso usar a função xmss_sign conforme definido na implementação acima, a do XMSS. O algoritmo ht_sign usa como entrada: - 1º uma mensagem M, - 2º uma semente particular SK.seed, - 3º uma semente pública PK.seed - 4º um índice idx.

O índice identifica a folha da hiperárvore a ser usada para assinar a mensagem. A assinatura HT consiste numa pilha(stack) de assinaturas XMSS usando as árvores XMSS no caminho da folha com o índice idx para a árvore superior. É ainda usada a função xmss_pkFromSig para determinar a nó raiz da instancia XMSS

[19]:
```
# Input: Message M, private seed SK.seed, public seed PK.seed, tree index␣
 ↪idx_tree, leaf index idx_leaf
# Output: HT signature SIG_HT
def ht_sign( m, secret_seed, public_seed, idx_tree, idx_leaf):
    # init
    adrs = ADRS()
    adrs.set_layer_address(0)
    adrs.set_tree_address(idx_tree)

    # sign
    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, adrs.
↪copy())
        sig_ht = sig_ht + sig_tmp

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

    return sig_ht
```

10.7.4 HT Signature Verification (Function ht_verify)

Podemos resumir a verificação da assinatura HT como d chama $xmss_pkFromSig$ e uma comparação com um determinado valor. A verificação da assinatura HT leva: - 1º uma mensagem M, - 2º uma assinatura SIGHT, - 3º uma semente pública $PK.seed$, - 4º um índice idx (dividido num índice de árvore e um índice folha, como acima) - 5º uma chave pública $HTPKHT$

[20]:
```
# Input: Message M, signature SIG_HT, public seed PK.seed, tree index idx_tree,
#leaf index idx_leaf, HT public key PK_HT.
# Output: Boolean
def ht_verify( 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, adrs)

    if node == public_key_ht:
```

```
        return True
    else:
        return False
```

10.8 FORS: Forest Of Random Subsets

O SPHINCS + hypertree HT não é usado para assinar as mensagens reais, mas as chaves públicas das instâncias do FORS que, por sua vez, são usadas para assinar os digest(resumos) de mensagens.

Primeiramente, FORS é uma melhoria do HORST, que por sua vez é uma variante do HORS. Por segurança, é referido no documento em estudo que é essencial que a entrada no FORS seja a saída de uma função de hash. O FORS usa os parâmetros k e $t = 2a$ (os parâmetros de exemplo são $t = 215$, $k = 10$). FORS assina cadeias de comprimento ka bits.

- A chave privada consiste em kt cadeias de nbytes aleatórias agrupadas em k conjuntos, cada uma contendo cadeias de t n-bytes. Os valores da chave privada são gerados pseudoaleatoriamente a partir da semente particular principal SK.seed na chave privada SPHINCS +.No SPHINCS +, os valores da chave privada FORS são gerados temporariamente apenas como resultado intermediário ao calcular a chave pública ou uma assinatura.
- A chave pública FORS é um único valor de hash de n bytes. É calculado como o hash ajustável dos nós raiz das k árvores binárias de hash.
- Uma assinatura numa sequencia M consiste em k valores de chave privada - um por conjunto de elementos de chave privada - e os caminhos de autenticação associados.

10.8.1 FORS Parametros

FORS usa os parâmetros n, k e t; todos eles recebem valores inteiros positivos. - n: o parâmetro de segurança; é o comprimento de uma chave privada, chave pública ou elemento de assinatura em bytes - k: o número de conjuntos de chaves privadas, árvores e índices calculados a partir da sequência de entrada. - t: o número de elementos por conjunto de chaves privadas, número de folhas por árvore de hash e limite superior nos valores do índice.(Deve ser potencia de 2)

10.8.2 FORS Private Key (Function fors_SKgen)

Uma chave privada FORS é a única semente privada SK.seed contida na chave privada SPHINCS. É usado para gerar os valores da chave privada kt n-byte usando PRF com um endereço.

[21]:
```
'''
#Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
#Output: FORS private key sk
fors_SKgen(SK.seed, ADRS, idx) {
ADRS.setTreeHeight(0);
ADRS.setTreeIndex(idx);
sk = PRF(SK.seed, ADRS);
return sk;
}
'''
```

```
# Input: secret seed SK.seed, address ADRS, secret key index idx = it+j
# Output: FORS private key sk
def fors_sk_gen( secret_seed, adrs, idx):
    adrs.set_tree_height(0)
    adrs.set_tree_index(idx)
    sk = prf(secret_seed, adrs.copy(), _n)
    return sk
```

10.8.3 FORS TreeHash (Function fors_treehash)

Antes de abordar propriamente a chave pública FORS é preciso para o cálculo dos nós de n bytes, nas árvores de hash FORS, é utilizada a sub-rotina fors_treehash. O algoritmo fors_treehash aceita: - uma semente secreta SK.seed, - uma semente pública PK.seed, - um número inteiro não assinado s (o índice inicial), - um número inteiro não assinado z (a altura do nó de destino) - um endereço ADRS que codifica o endereço da chave FORS par. Como no *treehash*, o algoritmo fors_treehash devolve o nó raiz de uma árvore de altura z com a folha mais à esquerda sendo o hash do elemento da chave privada no índice s. Além disto, s abrange todos os elementos da chave privada kt e ainda É NECESSÁRIO que s

[22]:
```
# 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( 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(), _n)
        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'] +␣
↪node, _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']
```

10.9 FORS Public Key (Function fors_PKgen)

A chave pública FORS nunca é gerada sozinha. É só gerado junto com uma assinatura. O algoritmo fors_PKgen utiliza - uma semente privada SK.seed, - uma semente pública PK.seed - um endereço FORS ADRS. E devolve a Chave publica PK

[23]:
```
# Input: Secret seed SK.seed, public seed PK.seed, address ADRS
# Output: FORS public key PK
def fors_pk_gen( secret_seed, public_seed, adrs):

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

    root = bytes()
    for i in range(0, _k):
        root += fors_treehash(secret_seed, i * _t, _a, public_seed, adrs)

    fors_pk_adrs.set_type(ADRS.FORS_ROOTS)
    fors_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())
    pk = hash(public_seed, fors_pk_adrs, root, _n)
    return pk
```

10.10 FORS Signature Generation (Function fors_sign)

Uma assinatura FORS é uma matriz de comprimento $k(logt + 1)$ de sequencias de caracteres de n bytes. Ele contém k valores de chave privada, n bytes cada e seus caminhos de autenticação associados, registram valores de log t n bytes cada. O algoritmo fors_sign usa: - uma sequência de bits (k log t) M, - uma semente particular SK.seed, - uma semente pública PK.seed - um endereço ADRS. E Devolve assinatura Fors

[24]:
```
# Input: Bit string M, secret seed SK.seed, address ADRS, public seed PK.seed
# Output: FORS signature SIG_FORS
def fors_sign( m, secret_seed, public_seed, adrs):
    m_int = int.from_bytes(m, 'big')
    sig_fors = []

    # compute signature elements
    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(), _n)]

        auth = []
```

```
        # compute auth path
        for j in range(0, _a):
            s = math.floor(idx // 2 ** j)
            if s % 2 == 1:  # XORING idx/ 2**j with 1
                s -= 1
            else:
                s += 1

            auth += [fors_treehash(secret_seed, i * _t + s * 2 ** j, j,␣
↪public_seed, adrs.copy())]

        sig_fors += auth

    return sig_fors
```

10.11 FORS Compute Public Key from Signature (Function fors_pkFromSig)

Uma assinatura FORS é usada para calcular uma chave pública FORS candidata. Essa chave pública é usada em outros cálculos (mensagem para a assinatura da árvore XMSS acima) e verificada implicitamente pelo resultado desse cálculo.

O método fors_pkFromSig utiliza: - uma cadeia de bits t de log k M, - uma assinatura FORS SIGFORS, - uma semente pública PK.seed, - um endereço ADRS. Devolve Chave Publica FORS

[25]:
```
# 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( sig_fors, m, public_seed, 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, _n)
        node_1 = 0
```

```
        # compute root from leaf and AUTH
        auth = sigs[i][1]
        adrs.set_tree_index(i * _t + idx)  # Really Useful?

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

            node_0 = node_1

        root += node_0

    # copy address to create FTS public key address
    fors_pk_adrs = adrs.copy()
    fors_pk_adrs.set_type(ADRS.FORS_ROOTS)
    fors_pk_adrs.set_key_pair_address(adrs.get_key_pair_address())

    pk = hash(public_seed, fors_pk_adrs, root, _n)
    return pk
```

10.12 SPHINCS+

Agora que temos todos os elementos implementados, o SPHINCS+ irá apenas "manipular" todos esses metodos e classes desenvolvida, acrescentando apenas compactação aleatória de mensagens e geração de índice verificável.

10.12.1 PARAMETROS SPHINCS+

O SPHINCS+ possui os seguintes parâmetros: - n: o parâmetro de segurança em bytes. - w: o parâmetro Winternitz - h: a altura da hiperárvore - d: o número de camadas na hiperárvore - k: o número de árvores no FORS - t: o número de folhas de uma árvore FORS

Lembrando ainda que, m: o comprimento do resumo da mensagem em bytes. É calculado como: $m = b(klogt + 7)/8c + b(h-h/d + 7)/8c + b(h/d + 7)/8c$.

E as formulas para calcular os len, len1 e len2, sendo len = len1 + len2.

[32]:
```
#Parametros SPHINCS+
"""
    n -- length of hash in WOTS / HORST (in bits)
    m -- length of message hash (in bits)
    h -- height of the hyper-tree
    d -- layers of the hyper-tree
```

```
    w -- Winternitz parameter used for WOTS signature
    tau -- layers in the HORST tree (2^tau is no. of secret-key elements)
    k -- number of revealed secret-key elements per HORST signature
"""
#Parametro de Segurança em Bytes
_n = 16
#Altura da hyper-Arvore
_h = 64
#Numero de Camadas na Hyper-Tree
_d = 8
#Numero de elementos chaves privadas por assinaturas FORS
_k = 10
# Número de folhas da árvore
_t = 2 ** 15
#Valor de Winternitz, usado no WOTS
_w = 16
_a = 15
#número de elementos de cadeia de caracteres de n-bytes
#numa chave privada WOTS +, chave pública e assinatura.
# Valor de len_1
_len_1 = math.ceil(8 * _n / math.log(_w, 2))
# Valor de len_2
_len_2 = math.floor(math.log(_len_1 * (_w - 1), 2) / math.log(_w, 2)) + 1
# Valor de len_0, sendo a soma dos dois anteriores
_len_0 = _len_1 + _len_2
_h_prime = _h // _d
```

[33]:
```
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
```

10.12.2 SPHINCS+ Key Generation (Function spx_keygen) [P31]

A chave privada SPHINCS+ contém dois elementos.

- 1º a semente secreta de n-bytes $SK.seed$, usada para gerar todos os elementos de chave privada WOTS+ e FORS.
- 2º Uma chave PRF de n-bytes $SK.prf$ que é usada para gerar deterministicamente um valor random para o hash da mensagem. A chave pública SPHINCS+ também contém dois elementos.
- 1º a chave pública HT, ou seja, a raiz da árvore na camada superior.
- 2º um valor de semente pública de n-bytes $PK.seed$, que é amostrado uniformemente aleatoriamente.

Como o spx_sign não obtém a chave pública, mas precisa acessar o PK.seed (e possivelmente o PK.root para atenuar o ataque a falhas), a chave secreta SPHINCS+ contém uma cópia da chave pública.

[34]:
```
'''
# Input: (none)
# Output: SPHINCS+ key pair (SK,PK)
spx_keygen( ){
    SK.seed = sec_rand(n);
    SK.prf = sec_rand(n);
    PK.seed = sec_rand(n);
    PK.root = ht_PKgen(SK.seed, PK.seed);
    return ( (SK.seed, SK.prf, PK.seed, PK.root), (PK.seed, PK.root) );
}
'''
#Gerar Par de Chaves para Assinaturas SPHINCS
def generate_key_pair():
    sk, pk = spx_keygen()
    sk0, pk0 = bytes(), bytes()
    for i in sk:
        sk0 += i
    for i in pk:
        pk0 += i
    #Devolve chave privada e a chave publica
    return sk0, pk0

# Input: (none)
# Output: SPHINCS+ key pair (SK,PK)
def spx_keygen():
    #Seed secreta de n-bytes, usada para gerar todos
    #os elementos de chave privada WOTS+ e FORS.
    secret_seed = os.urandom(_n)
    #chave PRF - Usada gerar deterministicamente um valor random para o hash da␣
 ↪mensagem
```

```
    secret_prf = os.urandom(_n)
    #Chave Publica que possui 2 elementos (sk.seed e pk.seed)
    public_seed = os.urandom(_n)
    public_root = ht_pk_gen(secret_seed, public_seed)
    #DevolveS PHINCS+ key pair (SK,PK)
    return [secret_seed, secret_prf, public_seed, public_root], [public_seed,␣
↪public_root]
```

10.12.3 SPHINCS+ Signature Generation (Function spx_sign)

A geração de uma assinatura SPHINCS+ consiste em quatro etapas.

- Primeiro, um valor aleatório R é gerado pseudoaleatoriamente.
- Em seguida, isso é usado para calcular um resumo de mensagens de byte que é dividido em ab (k log t + 7) / resumo parcial de 8 bytes de mensagem tmp_md, ab (h - h / d + 7) / índice de árvore de 8 bytes tmp_idx_tree, e ab (h / d + 7) / índice foliar de 8c bytes tmp_idx_leaf.
- Em seguida, os valores reais md, idx_tree e idx_leaf são calculados extraindo o número necessário de bits.
- O digest de mensagem parcial md é então assinado com o par de chaves FORS idx_leaf-th da árvore XMSS idx_tree-th na camada HT mais baixa.
- A chave pública do par de chaves FORS é então assinada usando HT.
- Ao calcular R, o PRF usa uma sequência de n-bytes opt, que é inicializada com zero, mas pode ser substituída aleatoriamente se a variável global RANDOMIZE estiver configurada.

[35]:
```
def sign(m, sk):
    """
    Sign a message with sphincs algorithm
    :param m: Message to be signed
    :param sk: Secret Key
    :return: Signature of m with sk
    """
    sk_tab = []

    for i in range(0, 4):
        sk_tab.append(sk[(i * _n):((i + 1) * _n)])

    sig_tab = spx_sign(m, sk_tab)

    sig = sig_tab[0]  # R
    for i in sig_tab[1]:  # SIG FORS
        sig += i
    for i in sig_tab[2]:  # SIG Hypertree
        sig += i

    return sig

# Input: Message M, private key SK = (SK.seed, SK.prf, PK.seed, PK.root)
# Output: SPHINCS+ signature SIG
```

```
def spx_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 True:
        opt = os.urandom(_n)
    r = prf_msg(secret_prf, opt, m, _n)
    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_tree +␣
↪size_idx_leaf)
    tmp_md = digest[:size_md]
    tmp_idx_tree = digest[size_md:(size_md + size_idx_tree)]
    tmp_idx_leaf = digest[(size_md + size_idx_tree):len(digest)]

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

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

    # 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_leaf)
    sig += [sig_ht]

    return sig
```

10.12.4 SPHINCS+ Signature Verification (Function spx_verify)

A verificação de assinatura SPHINCS+ pode ser vista como recalculando o digest e o índice da mensagem, calculando uma chave pública FORS candidata e verificando a assinatura HT nessa chave pública. Podemos também afirmar que a verificação da assinatura HT falhará se a chave pública FORS não corresponder à real. A verificação da assinatura SPHINCS+ leva:

- uma mensagem M,
- uma assinatura SIG
- uma chave pública SPHINCS+ PK.

[36]:
```
def verify(m,sig,pk):
    """
    Check integrity of signature
    :param m: Message signed
    :param sig: Signature of m
    :param pk: Public Key
    :return: Boolean True if signature correct
    """
    pk_tab = []

    for i in range(0, 2):
        pk_tab.append(pk[(i * _n):((i + 1) * _n)])

    sig_tab = []

    sig_tab += [sig[:_n]]  # R

    sig_tab += [[]]  # SIG FORS
    for i in range(_n,
                   _n + _k * (_a + 1) * _n,
                   _n):
        sig_tab[1].append(sig[i:(i + _n)])

    sig_tab += [[]]  # SIG Hypertree
    for i in range(_n + _k * (_a + 1) * _n,
                   _n + _k * (_a + 1) * _n + (_h + _d * _len_0) * _n,
                   _n):
        sig_tab[2].append(sig[i:(i + _n)])

    return spx_verify(m, sig_tab, pk_tab)
```

```
# Input: Message M, signature SIG, public key PK
# Output: Boolean
def spx_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_tree +␣
↪size_idx_leaf)
    tmp_md = digest[:size_md]
    tmp_idx_tree = digest[size_md:(size_md + size_idx_tree)]
    tmp_idx_leaf = digest[(size_md + size_idx_tree):len(digest)]

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

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

    # 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,␣
↪public_root)
```

[37]:
```
#Valor de Winternitz, Elemento de {4, 16, 256}.
setWinternitz(4)
#setWinternitz(16)
#setWinternitz(256)
#Gerar O par de Chaves, Chave Secreta e a Chave Publica
sk, pk = generate_key_pair()
#Mensagem
msg = b'Sao Joao Come a sardinha no pao'
#Gerar assinatura
signatur = sign(msg, sk)
#Verificar assinatura
ver_Sig = verify(msg, signatur, pk)
if ver_Sig == 1:
    print('Verificado')
else:
    print('Não verificado')
```

```
Verificado
```