Sphincs+

May 30, 2023

1 TP4

1.1 Grupo 17:

PG50315 - David Alexandre Ferreira Duarte

PG51247 - João Rafael Cerqueira Monteiro

1.2 Exercício 1.

Neste trabalho pretende-se implementar em Sagemath de algumas dos candidatos a "standartização" ao concurso NIST Post-Quantum Cryptography na categoria de esquemas de assinatura digital. Ver também a directoria com a documentação. Construa

2. Um protótipo Sagemath do algoritmo Sphincs+

1.3 Sphincs+

O algoritmo SPHINCS+ (SPHINCS Plus) é um esquema de assinatura digital pós-quântico. Ele foi desenvolvido como uma solução para resistir a ataques de computadores quânticos, que poderiam quebrar muitos dos algoritmos de criptografia atualmente usados.

O SPHINCS+ utiliza uma construção conhecida como árvore hash Merkle, que é um tipo de estrutura de dados de árvore onde os dados são verificados através de hashes criptográficos. Ele emprega um conjunto de funções hash criptográficas e algoritmos de autenticação para gerar uma assinatura digital.

O WOTS (Winternitz One-Time Signature) é um esquema de assinatura digital que é usado apenas uma vez. O XMSS (Extended Merkle Signature Scheme) é uma extensão do WOTS que permite criar várias assinaturas usando uma única chave privada. Ele usa uma estrutura de árvore de Merkle para fornecer segurança e eficiência na verificação das assinaturas.

Fors (Few-Time Signature Scheme), que é um esquema de assinatura criptográfica baseado em árvores de Merkle chamadas de FORS (Few-time Signature Scheme).

[6]:
```
from ADRS import *
from xmss import Xmss
from fors import Fors
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
      :return: secret key e public key
  """
  def keygen(self):

      # Geração dos seeds
      secret_seed = os.urandom(self._n) # Para gerar sk do WOTS
      secret_prf = os.urandom(self._n)
      public_seed = os.urandom(self._n)

      public_root = self.xmss.hypertree_pk_gen(secret_seed, public_seed)
```

```
        return [secret_seed, secret_prf, public_seed, public_root],␣
↪[public_seed, public_root]

    """
        Assinar uma mensagem com o algoritmo Sphincs
        :param m: Mensagem a ser assinada
        :param sk: Secret Key
        :return: Assinatura da mensagem com a Secret key
    """
    def sign(self, m, secret_key):
        #  Assinatura FORS do hash da mensagem, assinatura WOTS+ da pk do FORS␣
↪correspondente e
        # uma série de caminhos de autenticação, além das assinaturas WOTS+␣
↪para autenticar as pk's

        adrs = ADRS()

        #Obter seeds
        secret_seed = secret_key[0]
        secret_prf = secret_key[1]
        public_seed = secret_key[2]
        public_root = secret_key[3]

        # Gerar o r que será usado como nounce na geração da assinatura
        opt = os.urandom(self._n)
        r = prf_msg(secret_prf, opt, m, self._n)
        sig = [r]


        #hash da mensagem
        #comprime a mensagem a ser assinada usando o sha256
        digest = hash_msg(r, public_seed, public_root, m, self.size_md + self.
↪size_idx_tree + self.size_idx_leaf)
        # indices dos nós das árvores FORS
        tmp_md = digest[:self.size_md]
        # indice da arvore
        tmp_idx_tree = digest[self.size_md:(self.size_md + self.size_idx_tree)]
        # indice da folha
        tmp_idx_leaf = digest[(self.size_md + self.size_idx_tree):len(digest)]

        # conversão para inteiros
        # X >> Y  retorna X com bits deslocados para a direita em Y casas
        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_layer_address(0)
        adrs.set_tree_address(idx_tree)
        adrs.set_type(ADRS.FORS_TREE)
        adrs.set_key_pair_address(idx_leaf)

        # Assinatura do FORS
        sig_fors = self.fors.fors_sign(md, secret_seed, public_seed, adrs.
↪copy())
        sig += [sig_fors]

        pk_fors = self.fors.fors_pk_from_sig(sig_fors, md, public_seed, adrs.
↪copy())

        # Assinatura das PK do FORS utilizando a Hypertree
        adrs.set_type(ADRS.TREE)
        sig_hypertree = self.xmss.hypertree_sign(pk_fors, secret_seed,␣
↪public_seed, idx_tree, idx_leaf)
        sig += [sig_hypertree]

        return sig

    """
        Verificar a assinatura
        :param m: Mensagem
        :param sig: Assinatura
        :param pk: Public Key
        :return: Boolean True se assinatura correta
    """
    def verify(self, m, sig, public_key):
        #retirar as partes da assinatura
        adrs = ADRS()
        r = sig[0]
        sig_fors = sig[1]
        sig_hypertree = sig[2]

        public_seed = public_key[0]
        public_root = public_key[1]

        #hash da mensagem
        #comprime a mensagem a ser assinada
        digest = hash_msg(r, public_seed, public_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)]

        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_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, public_seed, adrs)

        adrs.set_type(ADRS.TREE)
        # Se Hypertree verifica a PK do FORS, retorna True
        return self.xmss.hypertree_verify(pk_fors, sig_hypertree, public_seed,␣
↪idx_tree, idx_leaf, public_root)
```

1.4 Teste

[7]:
```
sphincs = Sphincs()
sk, pk = sphincs.keygen()
print("sk: ", sk)
print("\npk: ", pk)

m = os.urandom(32)
print("\nMensagem: ", m)

signature = sphincs.sign(m, sk)

print("\nAssinatura correta? ", sphincs.verify(m, signature, pk))
```

```
sk:  [b'\xda;\x0c\x88\x90\xa55\xea\x93\x0fj\xf2&\r\xca\x0e',
b'\xc8\xabM\x00P\x87\xf4\x9e\xca]9D)@\xbb\xe4',
b'\x0f\x0f>\xb8\x9d\xb8\x9d\xf5\x1e\xeca_\xc1\x81\xe7\x15',
b'\xca&n\x8fq+\x9c:\xab\x8e\xc6}\xd8\xb2H\x01']

pk:  [b'\x0f\x0f>\xb8\x9d\xb8\x9d\xf5\x1e\xeca_\xc1\x81\xe7\x15',
b'\xca&n\x8fq+\x9c:\xab\x8e\xc6}\xd8\xb2H\x01']
```

```
Mensagem:  b'ER\xaa*}\x04y;\x87\xcdR\x9fm!\xe5\xa0\x94"\xc1\xa3\x83i:A\xebd
\xb1\xe8\x82\xf9\x83'

Assinatura correta?  True
```