TP3_Exercicio2

April 30, 2024

1 TP3 - Exercício 2

1.0.1 Autores

Afonso Ferreira - pg52669
Tiago Rodrigues - pg52705

1.0.2 Enunciado

2. Em Agosto de 2023 a NIST publicou um draf da norma FIPS203 para um Key Encapsulation Mechanism (KEM) derivado dos algoritmos KYBER.

   O preâmbulo do "draft" > A key-encapsulation mechanism (or KEM) is a set of algorithms that, under certain conditions, can be used by two parties to establish a shared secret key over a public channel. A shared secret key that is securely established using a KEM can then be used with symmetric-key cryptographic algorithms to perform basic tasks in secure communications, such as encryption and authentication. This standard specifes a key-encapsulation mechanism called ML-KEM. The security of ML-KEM is related to the computational diffculty of the so-called Module Learning with Errorsproblem. At present, ML-KEM is believed to be secure even against adversaries who possess a quantum computer

Neste trabalho pretende-se implementar em Sagemath um protótipo deste standard parametrizado de acordo com as variantes sugeridas na norma (512, 768 e 1024 bits de segurança)

Fizemos o *import* das bibliotecas necessárias

[ ]:
```
from typing import List, Tuple
import hashlib
import os
from functools import reduce
```

1.1 ShakeStream

Classe auxiliar que permite a leitura de dados a partir de uma função hash SHAKE128 em blocos, como se fosse um fluxo de dados.

[ ]:
```
class ShakeStream:
        def __init__(self, digestfn) -> None:
                self.digest = digestfn
                self.buf = self.digest(32)
                self.offset = 0
```

```
        def read(self, n: int) -> bytes:
                while self.offset + n > len(self.buf):
                        self.buf = self.digest(len(self.buf) * 2)
                res = self.buf[self.offset:self.offset + n]
                self.offset += n
                return res

if __name__ == "__main__":
        from hashlib import shake_128

        a = ShakeStream(shake_128(b"hello").digest)
        foo = a.read(17) + a.read(5) + a.read(57) + a.read(1432) + a.read(48)
        bar = shake_128(b"hello").digest(17 + 5 + 57 + 1432 + 48)
        assert(foo == bar)
```

1.2 Parâmetros

[ ]:
```
n = int(256)
q = int(3329)

DEFAULT_PARAMETERS = {
    "kyber_512" : {
        "k" : 2,
        "eta_1" : 3,
        "eta_2" : 2,
        "du" : 10,
        "dv" : 4,
    },
    "kyber_768" : {
        "k" : 3,
        "eta_1" : 2,
        "eta_2" : 2,
        "du" : 10,
        "dv" : 4,
    },
    "kyber_1024" : {
        "k" : 4,
        "eta_1" : 2,
        "eta_2" : 2,
        "du" : 11,
        "dv" : 5,
    }
}
```

1.3 Funções auxiliares

- `BitRev7` - Inverte a ordem dos bits de um número inteiro 'i' de 7-bits (entre 0 e 127).

- Funções `poly256` usadas para cálculos de polinómios

[ ]:
```python
def bitrev7(n: int) -> int:
        return int(f"{n:07b}"[::-1], 2)

def poly256_add(a: List[int], b: List[int]) -> List[int]:
        return [(x + y) % q for x, y in zip(a, b)]

def poly256_sub(a: List[int], b: List[int]) -> List[int]:
        return [(x - y) % q for x, y in zip(a, b)]

def poly256_slow_mul(self, a: List[int], b: List[int]) -> List[int]:
        c = [0] * 511
        # textbook multiplication, without carry
        for i in range(256):
                for j in range(256):
                        c[i+j] = (c[i+j] + a[j] * b[i]) % q

        # now for reduction mod X^256 + 1
        for i in range(255):
                c[i] = (c[i] - c[i+256]) % q
                # we could explicitly zero c[i+256] here, but there's no need...

        return c[:256]
```

2 NTT

Foi implementada um NTT diferente ao que foi implementado no trabalho prático anterior, visto que o NTT não era adequado para este trabalho.

Foi seguido, tal como no resto dos algoritmos, as indicações do FIPS203.

[ ]:
```python
ZETA = [pow(17, bitrev7(k), q) for k in range(128)] # used in ntt and ntt_inv
GAMMA = [pow(17, 2*bitrev7(k)+1, q) for k in range(128)] # used in ntt_mul


# by the way, this is O(n logn)
def ntt(f_in: List[int]) -> List[int]:
    f_out = f_in.copy()
    k = 1
    for log2len in range(7, 0, -1):
        length = 2**log2len
        for start in range(0, 256, 2 * length):
            zeta = ZETA[k]
            k += 1
            for j in range(start, start + length):
                t = (zeta * f_out[j + length]) % q
                f_out[j + length] = (f_out[j] - t) % q
```

```
                f_out[j] = (f_out[j] + t) % q
    return f_out


# so is this
def ntt_inv(f_in: List[int]) -> List[int]:
    f_out = f_in.copy()
    k = 127
    for log2len in range(1, 8):
        length = 2**log2len
        for start in range(0, 256, 2 * length):
            zeta = ZETA[k]
            k -= 1
            for j in range(start, start + length):
                t = f_out[j]
                f_out[j] = (t + f_out[j + length]) % q
                f_out[j + length] = (zeta * (f_out[j + length] - t)) % q

    for i in range(256):
        f_out[i] = (f_out[i] * 3303) % q  # 3303 == pow(128, -1, q)

    return f_out

ntt_add = poly256_add  # it's just elementwise addition

# and this is just O(n)
def ntt_mul(a: List[int], b: List[int]) -> List[int]:
    c = []
    for i in range(128):
        a0, a1 = a[2 * i: 2 * i + 2]
        b0, b1 = b[2 * i: 2 * i + 2]
        c.append((a0 * b0 + a1 * b1 * GAMMA[i]) % q)
        c.append((a0 * b1 + a1 * b0) % q)
    return c
```

3 Implementação da classe Kyber

Foram implementadas funções conforme está descrito no FIPS203.

- `PRF` - Pseudorandom function: gera sequências de números que parecem aleatórias
- `XOF` - Cálculo que gera saídas de tamanho arbitrário a partir de uma entrada

3.0.1 Funções de Hash

- `H`, `J`, `G`

3.0.2 Algoritmos de conversão

- `BitsToBytes`, `BytesToBits`

3.0.3 Compression & Decompression

- `Compress` - Remove os bits menos importantes de um número.
- `Decompress` - Adiciona bits iguais a zero nos lugares menos importantes.

3.0.4 Encoding & Decoding

- `ByteEncode` - Serialização
- `ByteDecode` - Desserialização

3.0.5 Algoritmos de Sampling

- `SampleNTT` - Representação uniforme do NTT
- `SamplePolyCBD` - Seleciona os coeficientes de um polinômio aleatoriamente de acordo com a distribuição.

3.1 K-PKE

3.1.1 K-PKE Key Generation

- Definir parâmetros de segurança (tamanho do módulo, etc.) e gerar chaves públicas e privadas para o emissor e receptor.

3.1.2 K-PKE Encryption

- Combinar a mensagem pré-processada com a chave pública do receptor e o ruído aleatório, resultando em um texto cifrado

3.1.3 K-PKE Decryption

- Combinar o texto cifrado com a chave privada do receptor e remover o ruído aleatório, recuperando a mensagem original

3.2 ML-KEM

3.2.1 ML-KEM Key Generation

- Definir parâmetros de segurança (tamanho do módulo, etc.) e gerar chaves públicas e privadas para o emissor e receptor.
- Gerar chaves encapsuladas aleatórias e combinar com a chave pública do receptor, resultando em uma cápsula.

3.2.2 ML-KEM Encapsulation

- Combinar a mensagem pré-processada com a cápsula e o ruído aleatório, resultando em um texto cifrado.

3.2.3 ML-KEM Decapsulation

- Verificar a validade do texto cifrado e recuperar a chave encapsulada.
- Combinar o texto cifrado com a chave privada do receptor, a chave encapsulada e remover o ruído aleatório, recuperando a mensagem original.

[ ]:
```python
class Kyber:
    def __init__(self, parameter_set) :
        self.k = parameter_set["k"]
        self.eta_1 = parameter_set["eta_1"]
        self.eta_2 = parameter_set["eta_2"]
        self.du = parameter_set["du"]
        self.dv = parameter_set["dv"]

    # crypto functions

    def mlkem_prf(self, eta: int, data: bytes, b: int) -> bytes:
        return hashlib.shake_256(data + bytes([b])).digest(64 * eta)

    def mlkem_xof(self, data: bytes, i: int, j: int) -> ShakeStream:
        return ShakeStream(hashlib.shake_128(data + bytes([i, j])).digest)

    def mlkem_hash_H(self, data: bytes) -> bytes:
        return hashlib.sha3_256(data).digest()

    def mlkem_hash_J(self, data: bytes) -> bytes:
        return hashlib.shake_256(data).digest(32)

    def mlkem_hash_G(self, data: bytes) -> bytes:
        return hashlib.sha3_512(data).digest()


    # encode/decode logic
    def bits_to_bytes(self,bits: List[int]) -> bytes:
        assert(len(bits) % 8 == 0)
        return bytes(
            sum(bits[i + j] << j for j in range(8))
            for i in range(0, len(bits), 8)
        )

    def bytes_to_bits(self,data: bytes) -> List[int]:
        bits = []
        for word in data:
            for i in range(8):
                bits.append((word >> i) & 1)
        return bits
```

```python
    def byte_encode(self,d: int, f: List[int]) -> bytes:
        assert(len(f) == 256)
        bits = []
        for a in f:
            for i in range(d):
                bits.append((a >> i) & 1)
        return self.bits_to_bytes(bits)

    def byte_decode(self, d: int, data: bytes) -> List[int]:
        bits = self.bytes_to_bits(data)
        return [sum(bits[i * d + j] << j for j in range(d)) for i in range(256)]

    def compress(self, d: int, x: List[int]) -> List[int]:
        return [(((n * 2**d) + q // 2 ) // q) % (2**d) for n in x]

    def decompress(self, d: int, x: List[int]) -> List[int]:
        return [(((n * q) + 2**(d-1) ) // 2**d) % q for n in x]


    # sampling

    def sample_ntt(self, xof: ShakeStream):
        res = []
        while len(res) < 256:
            a, b, c = xof.read(3)
            d1 = ((b & 0xf) << 8) | a
            d2 = c << 4 | b >> 4
            if d1 < q:
                res.append(d1)
            if d2 < q and len(res) < 256:
                res.append(d2)
        return res


    def sample_poly_cbd(self,eta: int, data: bytes) -> List[int]:
        assert(len(data) == 64 * eta)
        bits = self.bytes_to_bits(data)
        f = []
        for i in range(256):
            x = sum(bits[2*i*eta+j] for j in range(eta))
            y = sum(bits[2*i*eta+eta+j] for j in range(eta))
            f.append((x - y) % q)
        return f


    # K-PKE
```

```
  def kpke_keygen(self, seed: bytes=None) -> Tuple[bytes, bytes]:
      d = os.urandom(32) if seed is None else seed
      ghash = self.mlkem_hash_G(d)
      rho, sigma = ghash[:32], ghash[32:]

      ahat = []
      for i in range(self.k):
          row = []
          for j in range(self.k):
              row.append(self.sample_ntt(self.mlkem_xof(rho, i, j)))
          ahat.append(row)

      shat = [
          ntt(self.sample_poly_cbd(self.eta_1, self.mlkem_prf(self.eta_1,␣
↪sigma, i)))
          for i in range(self.k)
      ]
      ehat = [
          ntt(self.sample_poly_cbd(self.eta_1, self.mlkem_prf(self.eta_1,␣
↪sigma, i+self.k)))
          for i in range(self.k)
      ]
      that = [ # t = a * s + e
          reduce(ntt_add, [
              ntt_mul(ahat[j][i], shat[j])
              for j in range(self.k)
          ] + [ehat[i]])
          for i in range(self.k)
      ]
      ek_pke = b"".join(self.byte_encode(12, s) for s in that) + rho
      dk_pke = b"".join(self.byte_encode(12, s) for s in shat)
      return ek_pke, dk_pke


  def kpke_encrypt(self, ek_pke: bytes, m: bytes, r: bytes) -> bytes:
      that = [self.byte_decode(12, ek_pke[i*128*self.k:(i+1)*128*self.k]) for␣
↪i in range(self.k)]
      rho = ek_pke[-32:]

      # this is identical to as in kpke_keygen
      ahat = []
      for i in range(self.k):
          row = []
          for j in range(self.k):
              row.append(self.sample_ntt(self.mlkem_xof(rho, i, j)))
          ahat.append(row)
```

```
        rhat = [
            ntt(self.sample_poly_cbd(self.eta_1, self.mlkem_prf(self.eta_1, r,␣
↪i)))
            for i in range(self.k)
        ]
        e1 = [
            self.sample_poly_cbd(self.eta_2, self.mlkem_prf(self.eta_2, r,␣
↪i+self.k))
            for i in range(self.k)
        ]
        e2 = self.sample_poly_cbd(self.eta_2, self.mlkem_prf(self.eta_2, r,␣
↪2*self.k))

        u = [ # u = ntt-1(AT*r)+e1
            poly256_add(ntt_inv(reduce(ntt_add, [
                ntt_mul(ahat[i][j], rhat[j]) # note that i,j are reversed here
                for j in range(self.k)
            ])), e1[i])
            for i in range(self.k)
        ]
        mu = self.decompress(1, self.byte_decode(1, m))
        v = poly256_add(ntt_inv(reduce(ntt_add, [
            ntt_mul(that[i], rhat[i])
            for i in range(self.k)
        ])), poly256_add(e2, mu))

        c1 = b"".join(self.byte_encode(self.du, self.compress(self.du, u[i]))␣
↪for i in range(self.k))
        c2 = self.byte_encode(self.dv, self.compress(self.dv, v))
        return c1 + c2


    def kpke_decrypt(self,dk_pke: bytes, c: bytes) -> bytes:
        c1 = c[:32*self.du*self.k]
        c2 = c[32*self.du*self.k:]
        u = [
            self.decompress(self.du, self.byte_decode(self.du, c1[i*32*self.du:
↪(i+1)*32*self.du]))
            for i in range(self.k)
        ]
        v = self.decompress(self.dv, self.byte_decode(self.dv, c2))
        shat = [self.byte_decode(12, dk_pke[i*384:(i+1)*384]) for i in␣
↪range(self.k)]
        # NOTE: the comment in FIPS203 seems wrong here?
```

```
        # it says "NTT-1 and NTT invoked k times", but I think NTT-1 is only␣
↪invoked once.
        w = poly256_sub(v, ntt_inv(reduce(ntt_add, [
            ntt_mul(shat[i], ntt(u[i]))
            for i in range(self.k)
        ])))
        m = self.byte_encode(1, self.compress(1, w))
        return m


    # KEM time

    def mlkem_keygen(self,seed1=None, seed2=None):
        z = os.urandom(32) if seed1 is None else seed1
        ek_pke, dk_pke = self.kpke_keygen(seed2)
        ek = ek_pke
        dk = dk_pke + ek + self.mlkem_hash_H(ek) + z
        return ek, dk


    def mlkem_encaps(self, ek: bytes, seed=None) -> Tuple[bytes, bytes]:
        # TODO !!!! input validation !!!!!!!
        m = os.urandom(32) if seed is None else seed
        ghash = self.mlkem_hash_G(m + self.mlkem_hash_H(ek))
        k = ghash[:32]
        r = ghash[32:]
        c = self.kpke_encrypt(ek, m, r)
        return k, c


    def mlkem_decaps(self, c: bytes, dk: bytes) -> bytes:
        # TODO !!!! input validation !!!!!!!
        dk_pke = dk[:384*self.k]
        ek_pke = dk[384*self.k : 768*self.k + 32]
        h = dk[768*self.k + 32 : 768*self.k + 64]
        z = dk[768*self.k + 64 : 768*self.k + 96]
        mdash = self.kpke_decrypt(dk_pke, c)
        ghash = self.mlkem_hash_G(mdash + h)
        kdash = ghash[:32]
        rdash = ghash[32:]
        # NOTE: J() has unnecessary second argument in the spec???
        kbar = self.mlkem_hash_J(z + c)
        cdash = self.kpke_encrypt(ek_pke, mdash, rdash)
        if cdash != c:
            # I suppose this branch ought to be constant-time, but that's␣
↪already out the window with this impl
```

```
            #print("did not match") # XXX: what does implicit reject mean? I␣
↪suppose it guarantees it fails in a not-attacker-controlled way?
            return kbar
        return kdash
```

3.3 Teste

Este excerto de código demonstra a utilização básica do Kyber para mecanismos de encapsulamento de chaves pós-quânticas (K-PKE e ML-KEM): - Gerar pares de chaves; - Cifrar/decifrar; - Encapsular/desencapsular chaves secretas.

[ ]:
```
kyber = Kyber(DEFAULT_PARAMETERS["kyber_768"])
ek_pke, dk_pke = kyber.kpke_keygen(b"SEED"*8)

msg = b"This is a demonstration message."
ct = kyber.kpke_encrypt(ek_pke, msg, b"RAND"*8)
pt = kyber.kpke_decrypt(dk_pke, ct)
print(pt)
assert(pt == msg)

ek, dk = kyber.mlkem_keygen()
k1, c = kyber.mlkem_encaps(ek)
print("encapsulated:", k1.hex())

k2 = kyber.mlkem_decaps(c, dk)
print("decapsulated:", k2.hex())

assert(k1 == k2)
```

```
b'This is a demonstration message.'
encapsulated: 82a18e1c51e93ebce7aabea9d01f0743c70e63c0b8c61c576dc96be803f353c9
decapsulated: 82a18e1c51e93ebce7aabea9d01f0743c70e63c0b8c61c576dc96be803f353c9
```