TP3 - Ex.2

Abril 25, 2024

Estruturas Criptográficas

PG53886, Ivo Miguel Alves Ribeiro

A95323, Henrique Ribeiro Fernandes

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)

Hash Functions

```
In [ ]: import hashlib
import secrets

def Random_32_bytes():
    return secrets.token_bytes(32)

def H(s):
    return hashlib.sha3_256(s).digest()

def J(s):
    return hashlib.shake_256(s).digest(32)

def G(s):
    return hashlib.sha3_512(s).digest()

def XOF(x, i, j):
    return (hashlib.shake_128(x + bytes([i, j])).digest(256*3))

def PRF(data, b, eta):
        return hashlib.shake_256(data + bytes([b])).digest(64 * eta)


def polyADD(a, b, Q):
        return [(x + y) % Q for x, y in zip(a, b)]

def polySUB(a, b, Q):
        return [(x - y) % Q for x, y in zip(a, b)]
```

Algoritmo 2 BitsToBytes e Algoritmo 3 BytesToBits

```
In [ ]: def BitsToBytes(bits):
        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 BytesToBits(data):
        bits = []
        for word in data:
                for i in range(8):
                        bits.append((word >> i) & 1)
        return bits
```

Algoritmo 5 ByteEncode e 6 ByteDecode

```
In [ ]: def ByteEncode(F, d):
        assert(len(F) == 256)
        bits = []
        for a in F:
                for i in range(d):
                        bits.append((a >> i) & 1)
        return BitsToBytes(bits)

def ByteDecode(B, d):
        bits = BytesToBits(B)
        return [sum(bits[i * d + j] << j for j in range(d)) for i in range(256)]

def Compress(x, d, Q):
        return [(((n * 2**d) + Q // 2 ) // Q) % (2**d) for n in x]

def Decompress(x, d, Q):
        return [(((n * Q) + 2**(d-1) ) // 2**d) % Q for n in x]
```

Algoritmo 6 SampleNTT

```
In [ ]: def SampleNTT(B, Q):
        res = []
        i = 0
        while len(res) < 256:
                a, b, c = B[i:i+3]
                i+=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
```

Algoritmo 7 SamplePolyCBC

```
In [ ]: def SamplePolyCBC(B, Q, eta):
        assert(len(B) == 64 * eta)
        bits = BytesToBits(B)
        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
```

Algoritmo 8 NTT

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

def NTT(f_in, Q):
        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 = pow(17, bitrev7(k), Q)
                        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
```

Algoritmo 9 NTT-1

```
In [ ]: def NTT_inv(f_in, Q):
        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 = pow(17, bitrev7(k), Q)
                        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

        return f_out
```

Algoritmo 11 NTT mult

```
In [ ]: def NTT_mult(a, b, Q):
        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 * pow(17, 2*bitrev7(i)+1, Q)) % Q)
                c.append((a0 * b1 + a1 * b0) % Q)
        return c
```

K-PKE Component Scheme

```
In [ ]: class K_PKE:
    def __init__(self, n, q, k, n1, n2, du, dv):
        self.n = n
        self.q = q
        self.k = k
        self.n1 = n1
        self.n2 = n2
        self.du = du
        self.dv = dv

    def keygen(self):
        d = Random_32_bytes()
        ghash = G(d)
        rho, sigma = ghash[:32], ghash[32:]
        a_hat = []
        for i in range(self.k):
            row = []
            for j in range(self.k):
                row.append(SampleNTT(XOF(rho, i, j), self.q))
            a_hat.append(row)

        s_hat = [NTT(SamplePolyCBC(PRF(sigma, i, self.n1), self.q, self.n1), self.q) for i in range(self.k)]
        e_hat = [NTT(SamplePolyCBC(PRF(sigma, i+self.k, self.n1), self.q, self.n1), self.q) for i in range(self.k)]

        t_hat = []
        for i in range(self.k):
            sum = e_hat[i]
            for j in range(self.k):
                sum = polyADD(NTT_mult(a_hat[j][i], s_hat[j], self.q), sum, self.q)
            t_hat.append(sum)

        ek_pke = b"".join(ByteEncode(s, 12) for s in t_hat) + rho
        dk_pke = b"".join(ByteEncode(s, 12) for s in s_hat)
        return ek_pke, dk_pke


    def PKE_Encrypt(self, ek_pke, msg, r):
        t_hat = [ByteDecode(ek_pke[i*384:(i+1)*384], 12) for i in range(self.k)]

        rho = ek_pke[-32:]

        a_hat = []
        for i in range(self.k):
            row = []
            for j in range(self.k):
                row.append(SampleNTT(XOF(rho, i, j), self.q))
            a_hat.append(row)

        r_hat = [NTT(SamplePolyCBC(PRF(r, i, self.n1), self.q, self.n1), self.q) for i in range(self.k)]
        e1 = [SamplePolyCBC(PRF(r, i+self.k, self.n2), self.q, self.n2) for i in range(self.k)]
        e2 = SamplePolyCBC(PRF(r, 2*self.k, self.n2), self.q, self.n2)

        u = []
        for i in range(self.k):
            sum = e1[i]
            for j in range(self.k):
                # transposta de A_hat por trocar o i com o j
                sum = polyADD(NTT_inv(NTT_mult(a_hat[i][j], r_hat[j], self.q), self.q),sum, self.q)
            u.append(sum)

        μ = Decompress(ByteDecode(msg, 1), 1, self.q)

        v = [0]*256
        for i in range(self.k):
            v = polyADD(NTT_inv(NTT_mult(t_hat[i], r_hat[i], self.q), self.q), v, self.q)
        v = polyADD(v, polyADD(e2, μ, self.q), self.q)

        c1 = b"".join(ByteEncode(Compress(u[i], self.du, self.q), self.du) for i in range(self.k))
        c2 = ByteEncode(Compress(v, self.dv, self.q), self.dv)
        return c1 + c2


    def PKE_Decrypt(self, dk_pke, cypher):
        c1 = cypher[:32*self.du*self.k]
        c2 = cypher[32*self.du*self.k:]
        u = [Decompress(ByteDecode(c1[i*32*self.du:(i+1)*32*self.du], self.du), self.du, self.q) for i in range(self.k)]
        v = Decompress(ByteDecode(c2, self.dv), self.dv, self.q)
        s_hat = [ByteDecode(dk_pke[i*384:(i+1)*384], 12) for i in range(self.k)]

        w = [0]*256
        for i in range(self.k):
            w = polyADD(NTT_inv(NTT_mult(s_hat[i], NTT(u[i], self.q), self.q), self.q), w, self.q)
        w = polySUB(v, w, self.q)

        m = ByteEncode(Compress(w, 1, self.q),1)
        return m
```

Test K_PKE

```
In [ ]: test1 = K_PKE(256, 3329, 2, 3, 2, 10, 4)
ek_PKE, dk_PKE = test1.keygen()
print("Encryption Key (ek_PKE):", ek_PKE)
print("Tamanho da Encryption Key: ", len(ek_PKE))
print("Decryption Key (dk_PKE):", dk_PKE)
print("Tamanho da Decryption Key: ", len(dk_PKE))

msg = b"Esta e uma mensagem de 32 bytes!"
print("Tamanho da mensagem: ",len(msg))
r = Random_32_bytes()
cyphertext = test1.PKE_Encrypt(ek_PKE, msg, r)
print("Cyphertext: ", cyphertext)

decrypt_msg = test1.PKE_Decrypt(dk_PKE, cyphertext)
print("Decrypt message: ", decrypt_msg)
assert(decrypt_msg == msg)
```

```
Encryption Key (ek_PKE): b'[illegible]'
Tamanho da Encryption Key:  800
Decryption Key (dk_PKE): b'[illegible]'
Tamanho da Decryption Key:  768
Tamanho da mensagem:  32
Cyphertext:  b'[illegible]'
Decrypt message:  b'Esta e uma mensagem de 32 bytes!'
```

ML-KEM Key-Encapsulation Mechanism

```
In [ ]: class KEM:
    def __init__(self, kem_type):
        self.n = 256
        self.q = 3329
        self.sharedkeysize = 32
        self.type = kem_type
        if kem_type == 512:
            self.k = 2
            self.n1 = 3
            self.n2 = 2
            self.du = 10
            self.dv = 4
            self.enckeysize = 800
            self.deckeysize = 1632
            self.ciphersize = 768
        if kem_type == 768:
            self.k = 3
            self.n1 = 2
            self.n2 = 2
            self.du = 10
            self.dv = 4
            self.enckeysize = 1184
            self.deckeysize = 2400
            self.ciphersize = 1088
        if kem_type == 1024:
            self.k = 4
            self.n1 = 2
            self.n2 = 2
            self.du = 11
            self.dv = 5
            self.enckeysize = 1568
            self.deckeysize = 3168
            self.ciphersize = 1568
        self.PKE = K_PKE(self.n, self.q, self.k, self.n1, self.n2, self.du, self.dv)

    def KeyGen(self):
        z = Random_32_bytes()
        ek_pke, dk_pke = self.PKE.keygen()
        ek = ek_pke
        dk = dk_pke + ek + H(ek) + z
        # Verificar parametros
        assert len(ek) == self.enckeysize
        assert len(dk) == self.deckeysize
        return ek, dk


    def Encaps(self, ek):
        msg = b"Shared secrect key com 32 bytes!"
        print("Chave partilhada: ", msg)
        print("Tamanho da mensagem: ",len(msg))
        ghash = G(msg + H(ek))
        k = ghash[:32]
        r = ghash[32:]
        c = self.PKE.PKE_Encrypt(ek, msg, r)
        # Verificar parametros
        assert len(k) == self.sharedkeysize
        assert len(c) == self.ciphersize
        return k, c

    def Decaps(self, c, dk):
        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.PKE.PKE_Decrypt(dk_pke, c)
        ghash = G(mdash + h)
        kdash = ghash[:32]
        rdash = ghash[32:]
        kbar = J(z + c)
        cdash = self.PKE.PKE_Encrypt(ek_pke, mdash, rdash)
        if cdash != c:
            return kbar
        assert len(kdash) == self.sharedkeysize
        return kdash
```

test ML-KEM-512

```
In [ ]: kem1 = KEM(512)
ek_KEM, dk_KEM = kem1.KeyGen()
print("Encryption Key (ek_KEM):", ek_KEM)
print("Tamanho da Encryption Key: ", len(ek_KEM))
print("Decryption Key (dk_KEM):", dk_KEM)
print("Tamanho da Decryption Key: ", len(dk_KEM))

shared_key1, cypherText = kem1.Encaps(ek_KEM)
print("Shared Key: {}\nTamanho da chave: {}".format(shared_key1, len(shared_key1)))
print("Cypher Text: {}\nTamanho da cifra: {}".format(cypherText, len(cypherText)))

shared_key2 = kem1.Decaps(cypherText, dk_KEM)
print("Shared Key (Decaps): {}".format(shared_key2))

if shared_key1 == shared_key2:
    print("ML-KEM Key Encapsulation Mechanism Valid!")
else:
    print("ML-KEM Key Encapsulation Mechanism Invalid!")
```

```
Encryption Key (ek_KEM): b'[illegible]'
Tamanho da Encryption Key:  800
Decryption Key (dk_KEM): b'[illegible]'
Tamanho da Decryption Key:  1632
Chave partilhada:  b'Shared secrect key com 32 bytes!'
Tamanho da mensagem:  32
Shared Key: b'[illegible]'
Tamanho da chave: 32
Cypher Text: b'[illegible]'
Tamanho da cifra: 768
Shared Key (Decaps): b'[illegible]'
ML-KEM Key Encapsulation Mechanism Valid!
```

test ML-KEM-768

```
In [ ]: kem2 = KEM(768)
ek_KEM, dk_KEM = kem2.KeyGen()
print("Encryption Key (ek_KEM):", ek_KEM)
print("Tamanho da Encryption Key: ", len(ek_KEM))
print("Decryption Key (dk_KEM):", dk_KEM)
print("Tamanho da Decryption Key: ", len(dk_KEM))

shared_key1, cypherText = kem2.Encaps(ek_KEM)
print("Shared Key: {}\nTamanho da chave: {}".format(shared_key1, len(shared_key1)))
print("Cypher Text: {}\nTamanho da cifra: {}".format(cypherText, len(cypherText)))

shared_key2 = kem2.Decaps(cypherText, dk_KEM)
print("Shared Key (Decaps): {}".format(shared_key2))

if shared_key1 == shared_key2:
    print("ML-KEM Key Encapsulation Mechanism Valid!")
else:
    print("ML-KEM Key Encapsulation Mechanism Invalid!")
```

```
Encryption Key (ek_KEM): b'[illegible]'
Tamanho da Encryption Key:  1184
Decryption Key (dk_KEM): b'[illegible]'
Tamanho da Decryption Key:  2400
Chave partilhada:  b'Shared secrect key com 32 bytes!'
Tamanho da mensagem:  32
Shared Key: b'[illegible]'
Tamanho da chave: 32
Cypher Text: b'[illegible]'
Tamanho da cifra: 1088
Shared Key (Decaps): b'[illegible]'
ML-KEM Key Encapsulation Mechanism Valid!
```

test ML-KEM-1024

```
In [ ]: kem3 = KEM(1024)
ek_KEM, dk_KEM = kem3.KeyGen()
print("Encryption Key (ek_KEM):", ek_KEM)
print("Tamanho da Encryption Key: ", len(ek_KEM))
print("Decryption Key (dk_KEM):", dk_KEM)
print("Tamanho da Decryption Key: ", len(dk_KEM))

shared_key1, cypherText = kem3.Encaps(ek_KEM)
print("Shared Key: {}\nTamanho da chave: {}".format(shared_key1, len(shared_key1)))
print("Cypher Text: {}\nTamanho da cifra: {}".format(cypherText, len(cypherText)))

shared_key2 = kem3.Decaps(cypherText, dk_KEM)
print("Shared Key (Decaps): {}".format(shared_key2))

if shared_key1 == shared_key2:
    print("ML-KEM Key Encapsulation Mechanism Valid!")
else:
    print("ML-KEM Key Encapsulation Mechanism Invalid!")
```

```
Encryption Key (ek_KEM): b'[illegible]'
Tamanho da Encryption Key:  1568
Decryption Key (dk_KEM): b'[illegible]'
Tamanho da Decryption Key:  3168
Chave partilhada:  b'Shared secrect key com 32 bytes!'
Tamanho da mensagem:  32
Shared Key: b'[illegible]'
Tamanho da chave: 32
Cypher Text: b'[illegible]'
Tamanho da cifra: 1568
Shared Key (Decaps): b'[illegible]'
ML-KEM Key Encapsulation Mechanism Valid!
```