Bike

May 10, 2021

1 Exercício 3 - grupo 6 - Ana Margarida Campos (A85166) , Nuno Pereira (PG42846)

Neste primeiro exercício foi proposta a implementação de duas versões, IND-CPA e IND-CCA, do protótipo **BIKE** que foi candidato ao concurso *NIST-PQC*, neste caso da terceira ronda. De seguida, é apresentada apenas uma das versões, sendo esta correspondente ao IND-CPA. Os resultados da resolução deste exercício acompanhado de uma explicação detalhada de cada função implementada. Este, foi desenvolvido tendo por base o documento *BIKE_Spec.2020.10.22.1.pdf*.

Para a resolução deste exercício recorremos a várias funções presentes no notebook *BitFlip-BIKE.ipynb* disponibilizado pelo docente. Para além destas, foi necessário definir outras funções auxiliares sendo estas:

- ***H***: função que recorre ao algoritmo de cifragem *CTR*;
- ***L*** e ***K_sha***: funções de *hash* que utilizam *SHA384*;
- ***bxor***: calcula a operação *XOR* entre duas sequências de *bytes*.

Como funções principais foram implementadas as seguintes funções com os seguintes propósitos:

- ***KeyGen***: gera as chaves pública e privada que vão ser necessárias no encapsulmanto e desincapsulamento. A chave pública corresponde a variável *h* e a chave privada é divida por duas partes sendo a primeira constituída por *h0*, *h1* e a segunda constituída por *sigma*;
- ***encaps***: função responsável pelo encapsulamento;
- ***decaps***: função reponsável pelo desencapsulamento.

[1]:
```
# imports necessários para a implementação
import random as rn
from cryptography.hazmat.primitives import hashes
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

[2]:
```
# constantes BIKE
r = 257
n = 2*r
t = 16
l = 256
```

```
K = GF(2)
um = K(1)
zero = K(0)

Vn  = VectorSpace(K,n)
Vr  = VectorSpace(K,r)
Vq  = VectorSpace(QQ,r)

Mr  = MatrixSpace(K,n,r)

R = PolynomialRing(K,name='w')
w = R.gen()
Rr = QuotientRing(R,R.ideal(w^r+1))

def mask(u,v):
    return u.pairwise_product(v)

def hamm(u):
    return sum([1 if a == um else 0 for a in u])

def geraBits(l):
    bits = ""
    for i in range(l):
        bits = bits + str(rn.randint(0,1))
    return bits

# função que recorre ao algoritmo de cifragem CTR
def H(key, m):
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CTR(iv))
    encryptor = cipher.encryptor()
    ct = encryptor.update(m) + encryptor.finalize()
    return ct

# função de hash que utiliza SHA384
def L(e0,e1):
    digest = hashes.Hash(hashes.SHA384())
    digest.update(e0)
    digest.update(e1)
    r = digest.finalize()
    return r

# função de hash que utiliza SHA384
def K_sha(m,C):
    c1, c2 = C
    digest = hashes.Hash(hashes.SHA384())
    digest.update(m)
```

```
    digest.update(c1)
    digest.update(c2)
    r = digest.finalize()
    return r

def rot(h):
    v = Vr() ; v[0] = h[-1]
    for i in range(r-1):
        v[i+1] = h[i]
    return v

def Rot(h):
    M = Matrix(K,r,r) ;M[0] = expand(h)
    for i in range(1,r):
        M[i] = rot(M[i-1])
    return M

def expand(f):
    fl = f.list(); ex = r - len(fl)
    return Vr(fl + [zero]*ex)

def expand2(code):
    (f0,f1) = code
    f = expand(f0).list() + expand(f1).list()
    return Vn(f)

def unexpand2(vec):
    u = vec.list()
    return (Rr(u[:r]),Rr(u[r:]))


# operação XOR entre duas sequências de bytes
def bxor(a, b):
    return bytes([ x^^y for (x,y) in zip(a, b)])

# bit flip
def BF(H,code,synd,cnt_iter=r, errs=0):
    mycode = code
    mysynd = synd

    while cnt_iter > 0 and hamm(mysynd) > errs:
        cnt_iter   = cnt_iter - 1

        unsats     = [hamm(mask(mysynd,H[i])) for i in range(n)]
        max_unsats = max(unsats)

        for i in range(n):
```

```
            if unsats[i] == max_unsats:
                mycode[i]  += um                ## bit-flip
                mysynd     += H[i]

    return mycode


def sparse_pol(sparse=3):
    coeffs = [1]*sparse + [0]*(r-2-sparse)
    rn.shuffle(coeffs)
    return Rr([1]+coeffs+[1])

# geração das chaves pública e privada
def keyGen():
    while True:
        h0 = sparse_pol(); h1 = sparse_pol()
        if h0 != h1 and h0.is_unit() and h1.is_unit(): # primeira parte da␣
 ↪chave privada
            break
    h = h0 * h1.inverse_of_unit() # chave pública
    sigma = geraBits(l) # segunda parte da chave privada
    return h0,h1,sigma,h

# encapsulamento
def encaps(h, key):
    m = geraBits(32)
    e = H(key, m.encode())
    e0 = e[:16]
    e1 = e[-16:]
    c = (e0 + e1, bxor(m.encode(),L(e0,e1)) )
    k = K_sha(m.encode(),c)
    return k,c

# desencapsulamento
def decaps(h0, h1, sigma, C, key):
    c0, c1 = C
    cr = (Rr(list(c0)), Rr(list(c1)))
    code = expand2(cr)
    H_matrix = block_matrix(2,1,[Rot(h0),Rot(h1)])
    synd = code * H_matrix
    e = BF(H_matrix,code,synd)
    (e0,e1) = unexpand2(e)
    m = bxor(c1, L(bytes(e0), bytes(e1)))
    if bytes(e) == H(key,m):
        k = K_sha(m,C)
    else:
        k = K_sha(sigma.encode(), C)
```

```
    return k
```

De seguida são apresentados os resultados obtidos com recurso às funções anteriores.

[4]:
```
key = os.urandom(32)
h0, h1, sigma, h = keyGen()
print("Chave pública: ", h, "\n")
print("Chave privada: h0=", h0, " h1=", h1, " sigma=", sigma, "\n")
k, C = encaps(h, key)
print("Resultado do encapsulamento: k=",k," C=",C, "\n")
desencapsulation = decaps(h0,h1,sigma,C, key)
print("Resultado do desencapsulamento:", desencapsulation)
```

```
Chave pública:  wbar^256 + wbar^255 + wbar^254 + wbar^253 + wbar^252 + wbar^251
+ wbar^250 + wbar^249 + wbar^248 + wbar^246 + wbar^244 + wbar^243 + wbar^242 +
wbar^239 + wbar^237 + wbar^234 + wbar^231 + wbar^229 + wbar^228 + wbar^224 +
wbar^223 + wbar^221 + wbar^220 + wbar^218 + wbar^217 + wbar^215 + wbar^214 +
wbar^212 + wbar^210 + wbar^209 + wbar^207 + wbar^205 + wbar^203 + wbar^202 +
wbar^200 + wbar^198 + wbar^196 + wbar^195 + wbar^192 + wbar^185 + wbar^184 +
wbar^183 + wbar^181 + wbar^180 + wbar^179 + wbar^178 + wbar^177 + wbar^176 +
wbar^174 + wbar^169 + wbar^168 + wbar^166 + wbar^162 + wbar^158 + wbar^157 +
wbar^154 + wbar^151 + wbar^150 + wbar^149 + wbar^148 + wbar^146 + wbar^145 +
wbar^144 + wbar^143 + wbar^142 + wbar^139 + wbar^138 + wbar^137 + wbar^136 +
wbar^134 + wbar^133 + wbar^130 + wbar^127 + wbar^124 + wbar^123 + wbar^122 +
wbar^118 + wbar^115 + wbar^114 + wbar^112 + wbar^111 + wbar^108 + wbar^106 +
wbar^104 + wbar^103 + wbar^102 + wbar^97 + wbar^95 + wbar^92 + wbar^91 + wbar^90
+ wbar^88 + wbar^87 + wbar^86 + wbar^84 + wbar^80 + wbar^78 + wbar^76 + wbar^72
+ wbar^69 + wbar^68 + wbar^64 + wbar^62 + wbar^60 + wbar^59 + wbar^57 + wbar^54
+ wbar^53 + wbar^52 + wbar^51 + wbar^50 + wbar^49 + wbar^48 + wbar^47 + wbar^46
+ wbar^42 + wbar^40 + wbar^38 + wbar^36 + wbar^34 + wbar^32 + wbar^31 + wbar^29
+ wbar^28 + wbar^27 + wbar^22 + wbar^19 + wbar^18 + wbar^16 + wbar^14 + wbar^12
+ wbar^10 + wbar^5 + wbar^4 + wbar^3

Chave privada: h0= wbar^256 + wbar^190 + wbar^189 + wbar^164 + 1  h1= wbar^256 +
wbar^252 + wbar^152 + wbar^136 + 1  sigma= 00111010000011001011000000111001100000
00111100011100111010010011111000011110010000110110101100000101000001010111100000 1
01100100111001111110101000111110001010100010001001011001101100011010100101101101
00100000101100110111000000111111111111010001011111011011001

Resultado do encapsulamento: k= b'0yN\xae\x89/Yr\x03\x0cn\xa7\x1a\x05\xf2\xe7EG\
xc9l\x19w7\xf4ra\x977\xa2\xbb\xf1\x15\xc0\xe9\xb9\xbe\xd1iP\xb9\xce\x0c\x06\x1e^
\xa46A'  C= (b"\xd4\xc0\xa1\x18\xfb\xfb\x9e%\xf4\xddS$\xac\xbeM\x91\xa6\x0c\x81y
\xba\xb4'\xc6R^3/\x043\xb5\x8f", b'\xc7\x14#\x96\xfbF\xc4\x0cr\x1aT\x14;\xab\xb8
\xe5R\xd2\x8a\xd7\xf5\xb8\x8c\n\xd8\xa5\x8e\r\xf9G\x94\x7f')

Resultado do desencapsulamento: b'o#F]\xe6\xff\xe3e\\\xfe\xf0\x8f=4K\xca\xe2\xb3
%r.C[H\xb9M\xc1t\xb9\\a\x03\xc1\xad\xad\xa5\x9cj\xd0?\xe9\xaaX3HW\xc8V'
```