TP2-BIKE

May 2, 2022

1 Trabalho prático 2 - Estruturas Criptográficas

Autores: Ariana Lousada (PG47034), Cláudio Moreira (PG47844)

Grupo 12

1.1 BIKE

KEM-IND-CPA Para o desenvolvimento da solução para esta questão, foi proposta a implementação de um KEM IND-CPA seguro e um PKE IND-CCA seguro. Neste documento é exposta uma proposta de solução para o KEM IND-CPA.

Para a construção desta solução foi consultado o ficheiro pdf mais recente da implementação do BIKE(https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf).

Começou-se pela definição das constantes do BIKE: - ***r*** : tamanho de cada bloco. De acordo com a especificação mais recente, este valor corresponde a 257 - ***n*** : conde lenght. Corresponde ao dobro do tamanho de cada bloco. Neste caso corresponde a 514. - ***t*** : error weight, que corresponde a 16 - ***l*** : tamanho do segredo partilhado. De acordo com a especificação do BIKE mais recente, este valor corresponde a 256.

De seguida definiram-se as funções H, K e L: - ***H*** : definida com base no SHAKE256 - ***K_func*** : definida com base no SHA384 - ***L*** : definida com base no SHA384

Por fim construíram-se as três principais funções de geração do par de chaves, encapsulamento e desencapsulamento: - ***keyGenerator*** : função responsável pela criação do par de chaves pública e privada. A variável *h* corresponde à chave pública, enquanto que a chave privada é divida por duas partes: a primeira é constituída por *h0* e *h1* e a segunda constituída pela variável *sigma*, que é calculada a partir da função auxiliar *gerabits*; - ***encap*** : função responsável pelo encapsulamento de uma dada chave. Recorre às funções H e K do BIKE e à chave pública gerada pela função anterior. - ***decap*** : função responsável pelo desencapsulamento de uma dada chave. Recorre a ambas as partes da chave privada gerada pela função *keyGenerator* anteriormente mencionada.

[20]:
```
# 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
from Crypto.Hash.SHAKE256 import SHAKE256_XOF
```

[21]:
```
# constantes BIKE
r = 257            # block size
n = 514            # code length (2*r)
t = 16             # error weight
l = 256            # shared secret size

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])

# gera uma palavra de bit aleatória com tamanho recebido como parâmetro
def geraBits(l):
    bits = ""
    for i in range(l):
        bits = bits + str(rn.randint(0,1))
    return bits

# função que recorre ao SHAKE256
def H(key, m):
    digest = hashes.Hash(hashes.SHAKE256(int(l)))
    digest.update(m)
    r = digest.finalize()
    return r

# hash function que utiliza SHA384
def K_func(m,C):
    c0, c1 = C
    digest = hashes.Hash(hashes.SHA384())
    digest.update(m)
    digest.update(c0)
    digest.update(c1)
```

```
    r = digest.finalize()
    return r

# hash function que utiliza SHA384
def L(e0,e1):
    digest = hashes.Hash(hashes.SHA384())
    digest.update(e0)
    digest.update(e1)
    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 keyGenerator():
    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 encap(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_func(m.encode(),c)
    return k,c

# desencapsulamento
def decap(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_func(m,C)
    else:
        k = K_func(sigma.encode(), C)
    return k
```

[22]:
```
key = os.urandom(32)
h0, h1, sigma, h = keyGenerator()
print("Chave pública: ", h, "\n")
print("Chave privada: h0=", h0, " h1=", h1, " sigma=", sigma, "\n")
k, C = encap(h, key)
print("Resultado do encapsulamento: k=",k," C=",C, "\n")
desencapsulation = decap(h0,h1,sigma,C, key)
print("Resultado do desencapsulamento:", desencapsulation)
```

```
Chave pública:  wbar^255 + wbar^254 + wbar^251 + wbar^250 + wbar^248 + wbar^247
+ wbar^246 + wbar^242 + wbar^235 + wbar^233 + wbar^228 + wbar^227 + wbar^224 +
wbar^222 + wbar^219 + wbar^218 + wbar^217 + wbar^212 + wbar^205 + wbar^204 +
wbar^200 + wbar^199 + wbar^198 + wbar^197 + wbar^196 + wbar^195 + wbar^193 +
wbar^192 + wbar^190 + wbar^189 + wbar^187 + wbar^186 + wbar^184 + wbar^182 +
wbar^180 + wbar^177 + wbar^175 + wbar^174 + wbar^173 + wbar^172 + wbar^171 +
wbar^169 + wbar^168 + wbar^167 + wbar^166 + wbar^165 + wbar^164 + wbar^162 +
wbar^159 + wbar^158 + wbar^156 + wbar^155 + wbar^154 + wbar^152 + wbar^151 +
wbar^149 + wbar^148 + wbar^147 + wbar^146 + wbar^145 + wbar^142 + wbar^141 +
wbar^138 + wbar^134 + wbar^131 + wbar^128 + wbar^127 + wbar^126 + wbar^124 +
wbar^121 + wbar^117 + wbar^114 + wbar^112 + wbar^110 + wbar^109 + wbar^107 +
wbar^103 + wbar^100 + wbar^97 + wbar^96 + wbar^94 + wbar^93 + wbar^92 + wbar^90
+ wbar^89 + wbar^87 + wbar^86 + wbar^83 + wbar^82 + wbar^81 + wbar^79 + wbar^78
+ wbar^67 + wbar^63 + wbar^62 + wbar^59 + wbar^58 + wbar^52 + wbar^48 + wbar^47
+ wbar^46 + wbar^45 + wbar^42 + wbar^40 + wbar^39 + wbar^38 + wbar^36 + wbar^33
+ wbar^32 + wbar^30 + wbar^28 + wbar^27 + wbar^24 + wbar^20 + wbar^19 + wbar^18
+ wbar^17 + wbar^13 + wbar^10 + wbar^9 + wbar^8 + wbar^7 + wbar^6 + wbar^4 +
wbar^3 + wbar^2 + wbar

Chave privada: h0= wbar^256 + wbar^240 + wbar^187 + wbar^164 + 1  h1= wbar^256 +
wbar^164 + wbar^134 + wbar^123 + 1  sigma= 00101000110110101111101010000011101100
00100010101101111001011101010110001001111101110000011100111000000100000001010001
10000100110100110110001100000110101110001101111001011010001001011000011100010000
01100000000001111110101001110100101101011011100000001000100

Resultado do encapsulamento: k= b'\x7fY\x8b\xdd\xa0->\x10\x18E\xfeB\xb4\xc6\xba\
x81\x8a7yy\xf3\xdaT!Cz\x16\x9bB\xc8d\x85T\xa0\xe8\xb3A\x87;\xcb+\xfe\xb1\xee\x95
```

\x10\x8c ' C= (b'\x84\xdf\xbc\n\xbf\xe8\xd4\x15\x88\xd1\x96.\x9d\\\xb8X!6jJ4\xb
9\xc0\xd5\xc1\xcb\x8f\xed\x1a\xa2n2',
b'\xca\x92\x01\xcb]\xaf\x1b\x92\x1dpY\xe0\xba\xf6\x86-Dh\xa8\xb4\x96\xa8\x82*
\x93>yC\xe2\xf9(')

Resultado do desencapsulamento: b'^(\xcd\x03\xc6\x0e\x8d\xe9Sf\xfc\xabuA"\x0c]\x
f0\xab\xcb\xc8;\x9e^\xcf\x1e\xdb\xc1\xe4\xf7\x13\x8bw\xa2\xe4\x83"\xd5\xcb_\xb7\
x03\xf3\x12y\'\x1f\x87'