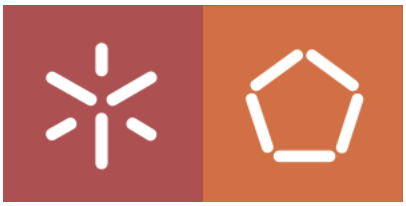

Universidade do Minho
Escola de Engenharia

Estruturas Criptográficas

Esquemas pós-quânticos de Assinaturas Digitais

Implementação protótipos de esquemas de Assinatura Digital em Python/SageMath

Submitted To:
José Valença
Professor Catedrático
Tecnologias da Informação e Segurança

Submitted By :
Diogo Araújo, A78485
Diogo Nogueira, A78957
Group 4

Conteúdo

1 Implementação do qTesla com o *SageMath*

O esquema qTesla é um esquema RLWE, ou seja, adota o problema computacional de `Ring Learning with Errors` para criar algoritmo criptográfico potente, capaz de se proteger a criptanálise de computadores quânticos.

[1]:
```
import numpy as np
import hashlib
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
```

1.1 Parâmetros para utilizar

n: dimensão

k: nº de amostras

q: módulo

h: nº de entradas diferentes de 0 nos `returns` da função `Enc`

K: comprimento do output da função `H` e do input da `GenA` e `Enc`

Le: limite de `checkE`

Ls: limite de `checkS`

B: determina o intervalo de aleatoriedade durante a assinatura

d: bits aleaórios

[2]:
```
K = 256
k = 1
h = 30
Le = 1586
Ls = 1586
B = 2^20-1
d = 21
n = 512
q = 4205569

Zx.<x> = ZZ[]
Gq.<z> = GF(q)[]

R.<x> = Zx.quotient(x^n+1) # R
Rq.<z> = Gq.quotient(z^n+1) # R/q
```

1.2 Funções Auxiliares

[3]:
```
def criarhash(s):
    h = hashlib.sha256()
    h.update(s)
    return h.digest()

def binary(tamanho=n):
    return list(np.random.choice([0,1],tamanho))

def GenA():
    return Rq.random_element()

def _center_lift(x):
    return lift(x + q//2) - q//2

def _round(w):
    return Zx(map(lambda x: _center_lift(x), w.list()))

def checkS(s):
    sum = 0
    ls = list(s)
    ls.sort(reverse=True)
    for i in range(0, h):
        sum += ls[i]
    if sum > Ls:
        return 1
    else:
        return 0

def checkE(e):
    sum = 0
    ls = list(e)
    ls.sort(reverse=True)
    for i in range(0, h):
        sum += ls[i]
    if sum > Le:
        return 1
    else:
        return 0

def H(v, hash_m):
    w = [0] * n
    for i in range(n):
```

```
        val = v[i] % 2^d
        if val > 2^(d-1):
            val = val - 2^d
        w[i] = (v[i] - val)/2^d
    return criarhash(str(w)+hash_m)
```

[4]:
```
def setup():
    a = GenA()
    s = None
    while True:
        s = Rq.random_element(distribution="gaussian")
        if checkS(s) == 0:
            break

    e = None
    while True:
        e = Rq.random_element(distribution="gaussian")
        if checkE(e) == 0:
            break
    t = a*s + e
    secret_key = (s, e, a)
    public_key = (a, t)
    return secret_key, public_key

def sign(mensagem, secret_key):
    s, e, a = secret_key
    y = Rq.random_element(x=-B, y=B+1, distribution="uniform")
    v = _round(a*y)
    c1 = H(v, criarhash(str(m)))
    c2 = Enc(c1)
    z = y + s*c2
    return (z, c1)

def verify(mensagem, assinatura, public_key):
    z, c1 = s
    c2 = Enc(c1)
    a, t = public_key
    w = _round(a*z - t)
    if c1 != H(w, hash(str(mensagem))):
        return False
    else:
        return True
```

[ ]:
```
mensagem = binary(n)

secret_key, public_key = setup()

assinatura = sign(mensagem, secret_key)

resultado = verify(mensagem, assinatura, public_key)

print(res)
```