qTesla

June 7, 2021

1 Implementação qTesla

Abaixo temos a implementação do esquema de assinaturas pós-quantico qTesla conforme as especificações da segunda submissão para o concurso do NIST.

[24]:
```
import os
import random

from cryptography.hazmat.primitives import hashes
```

Parâmetros do qTesla Abaixo temos um dos conjuntos pré definidos de parâmetros para o qTesla.

[34]:
```
'''
Lambda = 95
kappa = 256
n = 1024
k = 4
q = 3435776577
sigma = 8.5
LE = 554, 2.61
LS = 554, 2.61
E = 554
S = 554
B = 2^19 - 1
d = 22
h = 25
bGenA = 108
rateXOF = 168
CDTParams
beta = 64
precision 63
t = 78
size = 624
'''
```

[34]:
```
'\nLambda = 95\nkappa = 256\nn = 1024\nk = 4\nq = 343576577\nsigma = 8.5\nLE =
554, 2.61\nLS = 554, 2.61\nE = 554\nS = 554\nB = 2^19 - 1\nd = 22\nh = 25\nbGenA
= 108\nrateXOF = 168\nCDTParams \nn: 64 \ncdt_v: 63 beta: 78 624\n'
```

1.0.1 Geração do Par de Chaves

[17]:
```python
def gen():
    # 1
    counter = 1
    # 2
    pre_seed = os.urandom(32)
    # 3
    seed = PRF1(pre_seed)
    seeda = seed[len(seed) - 2]
    seedy = seed[len(seed) - 1]
    # 4
    a = GenA(seeda)

    # 6
    s = GaussSampler(seed[0], counter)
    # 7
    counter += 1

    # 5
    while checkS(s) != 0:
        # 6
        s = GaussSampler(seed[0], counter)
        # 7
        counter += 1

    # 8
    e = []
    t = []
    # 9
    for i in range(1, 4+1):
        # 11
        e.append(GaussSampler(seed[i], counter))
        # 12
        counter += 1
        # 13
        while checkE(e[i]) != 0:
            # 11
            e[i] = GaussSampler(seed[i], counter)
            # 12
            counter += 1
        # 14
        t.append((a[i - 1]*s + e[i]) % 343576577)
```

```
    g = G(t)

    sk = (s, e, seeda, seedy, g)

    pk = (t, seeda)

    return sk, pk
```

1.0.2 Assinatura

[3]:
```
def sign(m, sk):
    (s, e, seeda, seedy, g) = sk
    # 1
    counter = 1
    # 2
    r = os.urandom(32)
    # 3
    rand = PRF2(seedy, r, G(m))

    stop = 0
    while stop != 1:
        # 4
        y = ySampler(rand, counter)
        # 5
        a = GenA(seeda)
        # 6
        v = []
        for i in range(4):
            # 7
            v.append((a[i] * y) % 343576577)
            v[i] = v[i] - (343576577//2)

        # 9
        cl = H(v, G(m), g)
        # 10
        c = Enc(cl)
        # 11
        z = y + s*c
        stop = 1
        # 12 TODO condition1
        if condition1 :
            counter += 1
            stop = 0

        # 16
        w = []
```

```
        stop = 1
        for i in range(4):
            # 17
            w.append((v[i] - (e[i]*c)) % 343576577)
            w[i] = w[i] - (343576577//2)
            # 18 TODO condition2, condition3
            if condition2 | condition3:
                # 19
                counter += 1
                # 20
                stop = 0

    return (z, cl)
```

1.0.3 Verificação

[4]:
```
def verify(m, signature, pk):
    (z, cl) = signature
    (t, seeda) = pk
    # 1
    c = Enc(cl)
    # 2
    a = GenA(seeda)

    w = []
    # 3
    for i in range(k):
        # 4
        w.append((((a[i]*z) - (t[i]*c)) % 343576577))
        w[i] = w[i] - (343576577//2)

    # 6 TODO condition1
    if condition1 | cl != H(w, G(m), G(t)):
        return -1
    return 0
```

1.0.4 Execução de um Teste

[ ]:
```
pk, sk = Gen()

M = os.urandom(32)

signature = Sign(M, sk)

done = Verify(M, signature, pk)
```

```
if done == 0:
    print('True')
else:
    print('False')
```

1.0.5 PRF1

[32]:
```
def PRF1(pre_seed):
    n = 4+3
    digest = hashes.Hash(hashes.SHAKE128(int((256*n)//8)))
    digest.update(pre_seed)
    buffer = digest.finalize()

    n = 32

    return [buffer[x: x+n] for x in range(0, len(buffer), n)]
```

1.0.6 G

[25]:
```
def G(string):
    digest = hashes.Hash(hashes.SHAKE128(int(40))
    digest.update(string)
    return digest.finalize()
```

1.0.7 PRF2

[7]:
```
def PRF2(seedy, r, m):
    string = seedy + r + m
    digest = hashes.Hash(hashes.SHAKE128(int(32))
    digest.update(string)
    return digest.finalize()
```

1.0.8 GenA

[8]:
```
def GenA(seeda):
    D = 0

    b = logb(q,2) / 8
    b = b.ceil()
    bl = 108

    c = cSHAKE128(seeda, 168*bl, D)

    i = 0
    pos = 0
    while(i < 4096):
```

```
        x = (168*bl) // b
        if pos > x - 1:
            D += 1
            pos = 0
            bl = 1
            c = cSHAKE128(seeda, 168*bl, D)

        x = logb(q,2)
        x = x.ceil()
        x = 2^x
        x = c[pos] % x
        if x < 343576577:
            # TODO linha 10
            i += 1
        pos += 1

    return a
```

1.0.9 GaussSampler

[9]:
```
def GaussSampler(seed, D):
    Dl = D * (2^8)
    z = []

    for i in range(1024 * 512):
        z.append(0)

    for i in range(1024):
        # TODO / ?
        r = XOF(seed, (624*64)//8, Dl)
        Dl += 1
        # TODO chunk size????
        for j in range(512):
            # TODO / ?
            sign = r[j]//(2^(78-1))

            # TODO remove bits
            val = r[j]

            for k in range(78):
                # TODO condition4
                if condition4:
                    z[i+j] += 1

            if sign == 1:
                z[i+j] = -z[i+j]
```

```
        i += c

    return z
```

1.0.10 Enc

[10]:
```
def Enc(cl):
    D = 0
    cnt = 0

    r = cSHAKE128(cl, 168, D)


    c = []
    pos_list = []
    sign_list = []
    for i in range(25):
        c.append(0)
        pos_list.append(0)
        sign_list.append(0)

    i = 0
    while i < 25:
        if cnt > 168 - 3:
            D += 1
            cnt = 0

            r = cSHAKE128(cl, 168, D)

        pos = ((r[cnt] * (2^8)) + (r[cnt+1])) % 1024
        if c[pos] == 0:
            if (r[cnt + 2] % 2) == 1:
                c[pos] = -1
            else:
                c[pos] = 1
            pos_list[i] = pos
            sign_list[i] = c[pos]
            i += 1

        cnt += 3

    return pos_list, sign_list
```

1.0.11 ySampler

[11]:
```
def ySampler(rand, counter):
    pos = 0
    nl = 1024
    Dl = D * (2^8)
    b = logb(q,2) / 8
    b = b.ceil()

    c = XOF(rand, b*nl, Dl)

    i = 0
    y = []

    for i in range(1024):
        y[i] = 0

    while i < n:
        if pos >= nl:
            Dl += 1
            pos = 0
            nl = 168//b

            c = XOF(rand, 168, Dl)

        x = logb(2^19 - 1,2)
        x = x.ceil()
        x = 2^(x + 1) - (2^19 - 1)
        x = c[pos] % x
        y[i] = x
        if y[i] != (2^19):
            i += 1
        pos += 1
    return y
```

1.0.12 H

[ ]:
```
def H(v, m, t):
    w = []

    for i in range((4 * 1024) + 79):
        w.append(0)

    for i in range(1, 5):
        for j in range(n):
            val = v[i][j] % (2^22)
```

```
            if val > (2^22) - 1:
                val = val - (2^22)

            w[(i - 1)*n+j] = (v[i][j] - val)/(2^22)


    w1 = G(m)
    w2 = G(t)
    w = w1 + w2
    cl = SHAKE(w, 32)

    return cl
```

1.0.13 checkE

[13]:
```
def checkE(e):
    b = maxi(e[0])
    for i in range(1, k):
        b += maxi(e[i])

    if b > 554:
        return 1
    return 0
```

1.0.14 checkS

[15]:
```
def checkS(s):
    b = maxi(s[0])
    for i in range(1, k):
        b += maxi(s[i])

    if b > 554:
        return 1
    return 0
```

1.0.15 maxi

Função para ser usada nos dois checks

[ ]:
```
def maxi(e):
    return 0
```

1.0.16 XOF's

Provavelmente não é bem isto o que é pedido.

[ ]:
```
def cSHAKE128(seed, x, D):
    digest = hashes.Hash(hashes.SHAKE128(int(x))
    digest.update(seed)
    return digest.finalize()
```

[33]:
```
def XOF(seed, x, Dl):
    digest = hashes.Hash(hashes.SHAKE128(int(x))
    digest.update(seed)
    return digest.finalize()
```