A Visit to KAZ Attack: Finding a Minor Flaw and a Simplified Lattice Construction

Yongbo Hu, Chen Zhang, Guomiao Zhou
yongbohu@spric.org.cn, chenzhang@spric.org.cn, gmzhou@spric.org.cn
Shanghai Pudong Research Institute of Cryptology

Abstract

Inspired by a recent paper from Shanghai Jiao Tong University and China Telecom Quantum Information Technology Group [1]—which demonstrated a full break of the KAZ algorithm family submitted to Malaysia' s MySEAL 2.0 standardization—we focus specifically on its signature component. Within the same core theoretical framework, we have observed a subtle inaccuracy in the formula given in the original work. While this does not prevent the final private-key recovery via lattice reduction, it leads to incorrect derivation of the intermediate sensitive signature data e_1 and e_2. Building on this observation, we propose a refined lattice construction that successfully reproduces the original attack while eliminating the need for an additional step: computing the greatest common divisor (GCD) between the signature component S_2 and the modulus $\phi(N)$. This new construction is equally capable of recovering the private key using two signatures.

Keywords: KAZ, PQC, Lattice Reduction

Reproduction and Modification of the Original Matrix

In the original paper, the lattice matrix to attack the private key used in signature generation scheme is constructed as follow,

$$B_1 = \begin{pmatrix} \phi_1 & 0 & 0 & 0 \\ 0 & \phi_2 & 0 & 0 \\ -a_1 & -a_2 & 1 & 0 \\ b_1 & b_2 & 1 & 1 \end{pmatrix} \qquad \text{-------(1)}$$

And

$$g^{(i)} = \gcd\left(S_2^{(i)}, \phi(N)\right), \phi_i = \frac{\phi(N)}{g^{(i)}}, \widetilde{S_2}^{(i)} = \frac{S_2^{(i)}}{g^{(i)}} \qquad \text{-------(2)}$$

$$a_i = \frac{S_1^{(i)}}{\widetilde{S_2}^{(i)}} \ (mod \phi_i),\ b_i = \frac{h_i}{\widetilde{S_2}^{(i)}} \ (mod \phi_i) \qquad \text{-------(3)}$$

The formula (10) in original paper concludes that

$$e_1^{(i)} = b_i - s * a_i \% \phi_i \qquad \text{-------(4)}$$

However this is not correct, the signature scheme indicates

$$\begin{aligned} & s * S_1 + e_1 * S_2 = h \ \% \phi(N) \\ & => s * S_1^{(i)} + e_1^{(i)} * g^{(i)} * \widetilde{S_2}^{(i)} = h_i \ \% (g^{(i)} \phi_i) \\ & => e_1^{(i)} * g^{(i)} * \widetilde{S_2}^{(i)} = h_i - s * S_1^{(i)} + l * g^{(i)} \phi_i \\ & => e_1^{(i)} * \widetilde{S_2}^{(i)} = \frac{h_i - s * S_1^{(i)}}{g^{(i)}} + l * \phi_i \end{aligned}$$

$$=> e_1^{(i)} * \widetilde{S_2}^{(i)} = \frac{h_i - s * S_1^{(i)}}{g^{(i)}} \% \phi_i$$

$$=> e_1^{(i)} = \frac{b_i - s * a_i}{g^{(i)}} \% \phi_i \quad \text{-------(5)}$$

Hence the formula (10) in the original paper misses the $g^{(i)}$ part. Therefore, we made slight adjustments to the construction, modifying it as follows

$$\widetilde{B_1} = \begin{pmatrix} \phi_1 & 0 & 0 & 0 \\ 0 & \phi_2 & 0 & 0 \\ -a_1/g_1 & -a_2/g_2 & 1 & 0 \\ b_1/g_1 & b_2/g_2 & 0 & 1 \end{pmatrix} \quad \text{-------(6)}$$

Similar to the original conclusion, the proper (k_1, k_2) will make sure that $(k_1, k_2, s, 1) * \widetilde{B_1}$ yield a vector $V = \left(\frac{e_1}{g_1}, \frac{e_2}{g_2}, s, 1\right)$. Hence the secret s can be obtained after the lattice reduction. And the intermediate nonce can be calculated via

$$e_1 = \frac{V[0]}{V[3]} * g_1 \quad \text{-------(7)}$$

It is noteworthy that the original lattice construction remains effective for private-key recovery. This is because the reduced lattice vector directly yields the secret s within the tuple $(e_1 * g_1, e_2 * g_2, s, 1)$. However, this success masks an inaccuracy: the derived intermediate values e_1 and e_2 are incorrect.

The updated lattice detailed in formula (6) is implemented in the attached function <func: attack_sign_raw>

New lattice construction

The original construction requires handling of the $\gcd\left(S_2^{(i)}, \phi(N)\right)$. In fact, a new matrix can be constructed that eliminates the need for this special consideration entirely.

$$M = \begin{pmatrix} \phi(N) & 0 & 0 & 0 & 0 & 0 \\ 0 & \phi(N) & 0 & 0 & 0 & 0 \\ S_1^1 & S_1^2 & 1/s_{max} & 0 & 0 & 0 \\ S_2^1 & 0 & 0 & 1/e_{max} & 0 & 0 \\ 0 & S_2^2 & 0 & 0 & 1/e_{max} & 0 \\ h_1 & h_2 & 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{-------(8)}$$

Similar to the original proposal, the proper (k_1, k_2) will make sure that $(k_1, k_2, s, e_1^1, e_1^2, -1) * M$ yield a vector $V = \left(0, 0, \frac{s}{s_{max}}, \frac{e_1}{e_{max}}, \frac{e_2}{e_{max}}, -1\right)$, and the secret s can be obtained with following method

$$s = \left|\frac{V[2]}{V[5]} * s_{max}\right|, e_1 = \left|\frac{V[3]}{V[5]} * e_{max}\right| \quad \text{-------(9)}$$

In the attached sage scripts, a function named < func: attack_sign> indicate the case when $s_{max} = e_{max} = 2^{128}$ for reference.

Conclusion

Based on the breakthrough attack against the KAZ post-quantum cryptographic algorithm achieved by the China Telecom PQC research team in collaboration with Professor Yu Yu' s team at Shanghai Jiao Tong University, the team of our institution has conducted in-depth integration between our independently

developed research and the core algorithm of the original paper. This effort has led to results that are both more precise and widely applicable, which fully demonstrates the superiority of our institute' s improved approach. The outcome not only strengthens our research foundation in the field of PQC cryptographic algorithm security but also significantly advances our institute' s subsequent research capabilities in this direction.

Reference

[1] Zhuo Huang, Chongrong Li, Yu Yu, and Xiaogang Zhou, Breaking the KAZ Suite: Practical Key Recovery Attacks on MySEAL 2.0' s Post-Quantum Candidates.
https://eprint.iacr.org/archive/2026/064/1770049352.pdf

Reference sage code

```
def attack_sign_raw(sigs, N,phiN, g1,g2, erange):
    #Reproducing the implementation in the paper.
    #Input:
    #    sigs: signature tuple: (h,S1,S2,S3)
    #    g1,g2,N,phiN: the public information
    #    erange: max(e1),max(e2), 2**128, 2**192 or 2**256
    #return
    #    s,t: s,t candidates.
    h1,S1_1,S2_1,S3_1 = sigs[0]
    h2,S1_2,S2_2,S3_2 = sigs[1]
    g_1 = gcd(S2_1, phiN)
    g_2 = gcd(S2_2, phiN)
    phiN_r1,phiN_r2 = phiN//g_1, phiN//g_2
    S2_1_r, S2_2_r = S2_1//g_1, S2_2//g_2
    a1 = (S1_1 * inverse_mod(S2_1_r,phiN_r1)) % phiN_r1
    a2 = (S1_2 * inverse_mod(S2_2_r,phiN_r2)) % phiN_r2
    b1 = (h1   * inverse_mod(S2_1_r,phiN_r1)) % phiN_r1
    b2 = (h2   * inverse_mod(S2_2_r,phiN_r2)) % phiN_r2
    M = Matrix([[phiN_r1,0,0,0],[0,phiN_r2,0,0],[-a1/g_1,-
a2/g_2,1,0],[b1/g_1,b2/g_2,0,1]])
    L = M.LLL()
    s_rec = None
    for rows in L:
        a,x=abs(int(rows[2])),abs(int(rows[3]))
        if a%x==0:
            s_rec = a//x
            break
    c1 = ((h1-S3_1) * inverse_mod(S2_1_r, phiN_r1)) % phiN_r1
    c2 = ((h2-S3_2) * inverse_mod(S2_2_r, phiN_r2)) % phiN_r2
    M = Matrix([[phiN_r1,0,0,0],[0,phiN_r2,0,0],[-a1/g_1,-
a2/g_2,1,0],[c1/g_1,c2/g_2,0,1]])
    L = M.LLL()
    t_rec = None
    for rows in L:
        a,x=abs(int(rows[2])),abs(int(rows[3]))
        if a%x==0:
            t_rec = a//x
            break
    return s_rec, t_rec
```

```python
def attack_sign(sigs, N,phiN, g1,g2,erange):
    #A  lattice construction without special care of GCD(S2,phiN)
    #Input:
    #    sigs: signature tuple: (h,S1,S2,S3)
    #    g1,g2,N,phiN: the public information
    #    erange: max(e1),max(e2), 2**128, 2**192 or 2**256
    #return
    #    s,t: s,t candidates.
    one = 1/erange
    M = [[0]*(2*len(sigs)+2) for _ in range(2*len(sigs)+2)]
    for i in range(len(sigs)):
        M[len(sigs)][i]=sigs[i][1] #S1
        M[len(sigs)+1+i][i]=sigs[i][2] #S2
        M[len(sigs)*2+1][i]=sigs[i][0] #h
        M[i][i]=phiN  #phi(N)
        M[i+len(sigs)][i+len(sigs)]=one
    M[len(sigs)*2][len(sigs)*2]=one
    M[len(sigs)*2+1][len(sigs)*2+1]=one
    L = Matrix(M).LLL()
    s_rec = []
    for rows in L: #filter out the s candidate
        if(max([abs(xx) for xx in rows[:len(sigs)]])==0):
            s_rec.append(int(rows[len(sigs)]/one))
    one = 1/erange
    M = [[0]*(2*len(sigs)+2) for _ in range(2*len(sigs)+2)]
    for i in range(len(sigs)):
        M[len(sigs)][i]=sigs[i][1] #S1
        M[len(sigs)+1+i][i]=sigs[i][2] #S2
        M[len(sigs)*2+1][i]=sigs[i][0]-sigs[i][3]#h
        M[i][i]=phiN  #phi(N)
        M[i+len(sigs)][i+len(sigs)]=one
    M[len(sigs)*2][len(sigs)*2]=one
    M[len(sigs)*2+1][len(sigs)*2+1]=one
    L = Matrix(M).LLL()
    t_rec = []
    for rows in L:#filter out the s candidate
        if(max([abs(xx) for xx in rows[:len(sigs)]])==0):
            t_rec.append(int(rows[len(sigs)]/one))
    return s_rec, t_rec
```