Leak the Secret Key of ML-DSA in liboqs via Rowhammer

1 Analysing the ML-DSA scheme

ML-DSA is a post quantum signature standard as round 3 finalist in NIST competition. The underlying hard problem is the LWE over modulus lattice. ML-DSA has recently been implemented in liboqs. To demonstrate our attack over ML-DSA, we first give an simplified description of the scheme.

keygen(1^λ): This algorithm generates pk, sk related to RLWE problem through a random seed ρ. During the generation, number theoretical transform (NTT) is used for acceleration.

$$\zeta \leftarrow \{0,1\}^n, \ \ (\rho, \rho^{'}, K) \leftarrow H(\zeta) \tag{1}$$

compute NTT form of LWE matrix A, and its corresponding secret vector s_1, s_2 from the random seeds precomputed.

$$\begin{aligned} \hat{A} \in R_q^{k\times l} &\leftarrow ExpandA(\rho) \\ (s_1, s_2) \in S_\eta^l \times S_\eta^k &\leftarrow ExpandS(\rho^{'}) \end{aligned} \tag{2}$$

compute t as part of public key by NTT

$$t \leftarrow NTT^{-1}(\hat{A} \cdot NTT(s_1)) + s_2 \tag{3}$$

output secret key and public key as:

$$pk \leftarrow (\hat{A}, t), \quad sk \leftarrow (s_1, s_2, \zeta) \tag{4}$$

Sign(m, sk, pk): This function takes pk, sk and message m as inputs, and generate a signature (c, z) as follows:

$$\hat{s_1}, \hat{s_2}, \hat{t} \leftarrow NTT(s_1), NTT(s_2), NTT(t) \tag{5}$$

then generate a hash based on the pk and message and compute the private seed

$$\begin{aligned} \mu &\leftarrow Hash(m, pk) \\ \rho^{'} &\leftarrow H(K, u, rnd) \end{aligned} \tag{6}$$

compute (z, h) until they satisfies $||z||_\infty \geq \gamma_1 - \beta$ or $||r_0||_\infty \geq \gamma_2 - \beta$

$$
\begin{aligned}
y \in S_{\gamma_1} &\leftarrow ExpandMask(\rho^{'}, \kappa) \\
w &\leftarrow NTT^{-1}(\hat{A} \cdot NTT(y)) \\
c &\leftarrow Sample(Hash(\mu, w)) \\
< cs_1 > &\leftarrow NTT^{-1}(\hat{c} \cdot \hat{s_1}) \\
< cs_2 > &\leftarrow NTT^{-1}(\hat{c} \cdot \hat{s_2}) \\
z &\leftarrow y + < cs_1 > \\
r_0 &\leftarrow LowBits(w - < cs_2 >)
\end{aligned}
\tag{7}
$$

Then generate h as a hint for verifier to compute the correct w. Check if h is valid, if not, regenerate (z, c)

$$
h \leftarrow MakeHint(t, w, s_2) \tag{8}
$$

Finally, output the signature as σ

$$
\sigma \leftarrow Encode(c, z, h) \tag{9}
$$

Vrfy(m, pk, σ): This function takes a signature $\sigma = (c, z, h)$, message m and $pk = (h, q, n)$ as input, first computes w for further checking:

$$
\begin{aligned}
&\mu \leftarrow Hash(m, pk) \\
&w^{'} \leftarrow NTT^{-1}(\hat{A} \cdot NTT(z) - NTT(c) \cdot NTT(t)) \\
&w \leftarrow UseHint(h, w^{'})
\end{aligned}
\tag{10}
$$

Check if $||z||_\infty \geq \gamma_1 - \beta$ and $c \leftarrow Sample(Hash(u, w))$ and h is valid

We classified the parameters used in ML-DSA into two sets, public available parameters $pp = (A, t, \mu, c)$ and secret kept parameters $sp = (s_1, s_2, \zeta, y)$. We find that both DA and SCA can apply to ML-DSA scheme and the derivation is as follows.

Faulting to public available parameters (DA): Parse pp as A, t, μ, c, we find that when a single fault occurs to c, the output signature is converted to $z^{'} = y + c^{'} s_1$, compared to a valid one $z = y + cs_1$, we can compute secret key s_1 as:

$$
s_1 = (c^{'} - c)^{-1} \cdot (z^{'} - z) \tag{11}
$$

As there is a high probability that $(c^{'} - c)$ is invertible, we can easily compute s_1 with two signature queries. We can also conclude that parameters that serves as input when computing c can also be vulnerable, such as m, μ, pk.

Faulting to secret kept parameters (SCA): Parse sp as s_1, s_2, ζ, y, we only manage to perform a SCA when faulting to s_1, if one bit flip occurs in s_1 before the signature generation, the faulty signature becomes $z^{'} = y + cs_1^{'}$. The difference between the faulty signature and its valid one is $\Delta z = c(s_1 - s_1^{'}) = c\Delta s_1$. Note that s_1 is defined over S_ζ^l and its elements are all polynomials. Without loss of generality, let the one bit fault occurs in the j-th coefficient in the i-th element in s_1, denoted as a_{ij}, the the fault component is $\sum_{k=0}^{n-1} c_k x^k \cdot \overline{a_{ij}} x^j$. Since c can be computed without secret key, the attacker can eliminate the fault term to yield a valid signature, resulting leakage of the secret key.

2 Mitigation

An effective mitigation is *check after decryption*, which requires the secret owner to check if the result is a valid plaintext before releasing it. This mitigation has been adopted by WolfSSL and OpenSSL to fix similar vulnerabilities [2–4]. Specifically, we check if the secret key is faulted by checking whether $h = g^x$. If not, it means a secret key has been faulted and the process should abort.

References

[1] *CVE-2019-19962*. Available from MITRE. 2019.

[2] *EdDsa: check private value after sign*. 2024. URL: https://github.com/wolfSSL/wolfssl/commit/c8d0bb0bd8fcd3dd177ec04e9a659a006df51b73.

[3] *Openssl commit: Add a protection against fault attack on message v2*. 2018. URL: https://github.com/openssl/openssl/pull/7225/commits/02534c1ee3e84a1d6c59a887a67bd5ee81bcf6cf.

[4] *RSA Decryption: check private value after decryption*. 2024. URL: https://github.com/wolfSSL/wolfssl/commit/de4a6f9e00f6fbcaa7e20ed7bd89b5d50179e634.