



When PQC meets Fault Injection: Practical attacks on ML-KEM and ML-DSA

Thomas Techer | Security Evaluator | 1st October 2026

brightsign

By **SGS**



Our numbers

11

Locations around the globe

50+

Security standards and schemes recognitions

40

Years of experience in security evaluations

250+

Security evaluation experts

700+

Security projects performed each year

brightsight

By **SGS**

BrightSight services



Pre-evaluation

Risk mitigation by early pre-assessment

Examples:

- Readiness validation
- HW pre-evaluation
- Design review
- Documentation pre-analysis
- Pre-testing
- Gap analysis

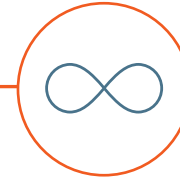


Security evaluation

Compliance assessment and testing of security requirements

Examples:

- Common Criteria
- SESIP (IOT)
- EMVCo (payment)
- SBMP



Activities

Different activities are performed as part of evaluations

Examples:

- Hardware design review
- Fault injection attacks
- Side-channel attacks
- Software attacks
- Code review

brightSight

By **SGS**

PQC Transition: SCA & FIA Considerations

- **NIST PQC Standardization Process** selected two schemes:

ML-KEM based on Kyber

- Key establishment scheme
- Separated in three algorithms
 - Key generation
 - Encapsulate
 - Decapsulate

ML-DSA based on Dilithium

- Digital signature scheme
- Separated in three algorithms
 - Key generation
 - Sign
 - Verify

- PQC's **mathematical resilience** against quantum computers **is not equivalent** to **resistance** against **physical implementation attacks**
 - NIST PQC Competition **primarily focused** on **evaluating** the **mathematical security** of algorithms against classical and quantum adversaries. **Side-channel** and **fault-injection** resistance were largely **left to post-standardization research**

State of the art

During security evaluations, the effort needed to perform attacks is evaluated following a rating table considering different factors:

- Time
- Type of equipment
- Access to DUTs (number of samples)

As attacks on lattice based schemes can be complex, effort of state of the art attacks should be studied

ML-KEM

- 9 attacks studied
- 4 attacks out of 9 are considered practical in security evaluations

ML-DSA

- 13 attacks studied
- 6 attacks out of 13 are considered practical in security evaluations

State of the art

Attack	Algorithm	Practicality	Fault Requirement
Forging Dilithium and Falcon Signatures by Single Fault Injection [1]	ML-DSA	A single instruction skip is needed to perform the attack	1 fault
Safe-Error Analysis of Post-Quantum Cryptography Mechanisms [4]	ML-DSA	A fault needs to be injected on each coefficient of the secret, which requires very precise fault. This is not considered practical in a reasonable amount of time.	1 fault per coefficient of the secret
When Randomness Isn't Random, Practical Fault Attack on PQ Lattice Standards [2]	ML-KEM	Single pointer corruption during key generation	1 fault
Number "Not Used" Once-Practical fault attack on pqm4 implementations of NIST candidates [3]	ML-KEM	Four faults are needed (for security level 2). Each incrementation of the nonce used during the sampling of secret and error must be skipped.	4 faults
Carry Your Fault: A Fault Propagation Attack on Side-Channel Protected LWE-based KEM [5]	ML-KEM	Fault model must be constant and large number of faults needed	1,8 millions of faults

● Practical

● Practical with our variant

● Not practical

Experimental setup

Target platform

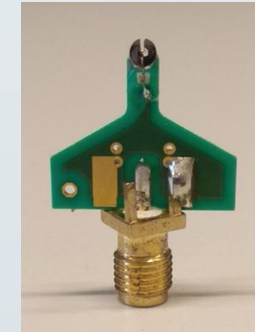
- **Cortex M4**
- Clock frequency: 84 MHz
- Supply voltage(Vcc): 3.3v
- Communication protocol: UART

Implementation

- **ML-KEM** and **ML-DSA** implemented based on the **pqm4** library
- **No countermeasures** implemented

EMFI setup

- **Semi-invasive** penetration test from **through the package**
- Standard **Brightsight EMFI setup**, including in-house **EMFI Pulser**
- In-house **crescent coil**



brightsight

By **SGS**

ML-DSA / Dilithium

FI on ML-DSA Verify

➤ Forging Dilithium and Falcon signatures by Single Fault Injection

- Forging ML_DSA signature with Single Fault Injection. One instruction skip in ML-DSA verify can make a forged signature accepted.

- Prepare the attack
 - Forge a signature for message $\mathbf{m}'$
 - Use $(\mathbf{z}, \mathbf{h})$ from a genuine signature
 - Public key is $(\rho, \mathbf{t}_1)$

1. Compute μ' (as in Algorithm 1)
2. Compute $\mathbf{w}_1'' = \text{UseHint}(\mathbf{h}, \mathbf{A}\mathbf{z}, 2\gamma_2)$
3. Compute $c' = \text{SHAKE256}(\mu' || \mathbf{w}_1'', 256)$

Algorithm 1 DILITHIUM.VERIFY(m, sig, pk)

Require: A message m a signature $\text{sig} = (\tilde{c}, \mathbf{z}, \mathbf{h})$, a public key $\text{pk} = (\rho, \mathbf{t}_1)$.

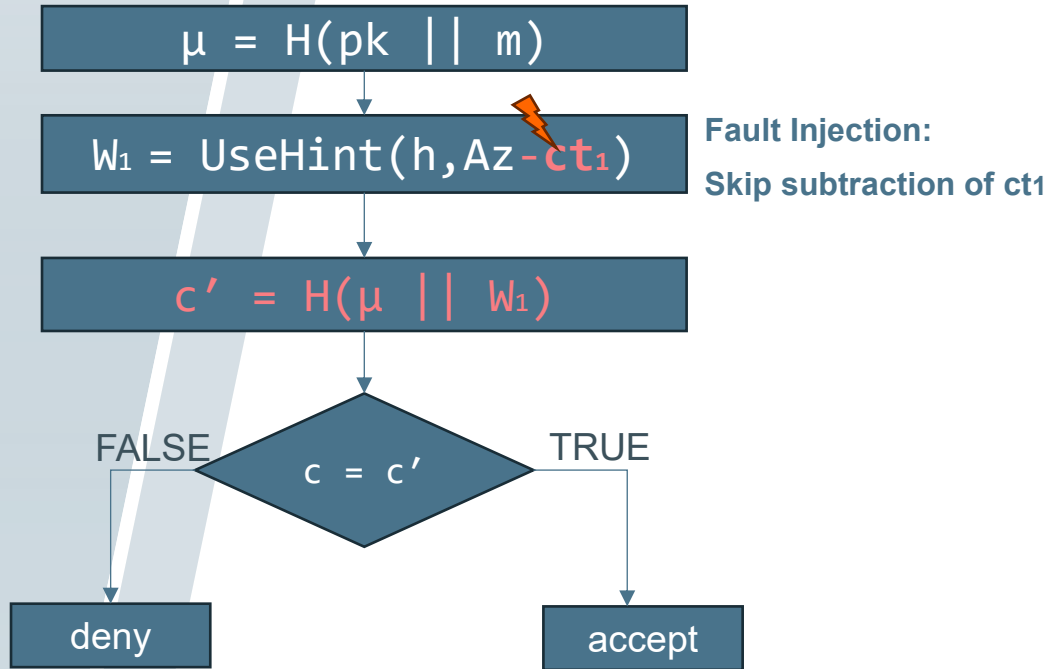
Ensure: Accept or reject

```
1:  $A \leftarrow \text{ExpandA}(\rho)$ 
2:  $\mu \leftarrow \text{SHAKE256}(\text{SHAKE256}(\rho || \mathbf{t}_1, 256) || m, 512)$ 
3:  $c \leftarrow \text{SampleInBall}(\tilde{c})$ 
4:  $\mathbf{w}_1' \leftarrow \text{UseHint}(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t}_1, 2\gamma_2)$ 
5: if  $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$  and  $\tilde{c} = \text{SHAKE256}(\mu || \mathbf{w}_1', 256)$  and
   the number of 1s in  $\mathbf{h}$  is  $\leq \omega$  then
6:   accept
7: else
8:   reject
```

⚡ Skip subtraction

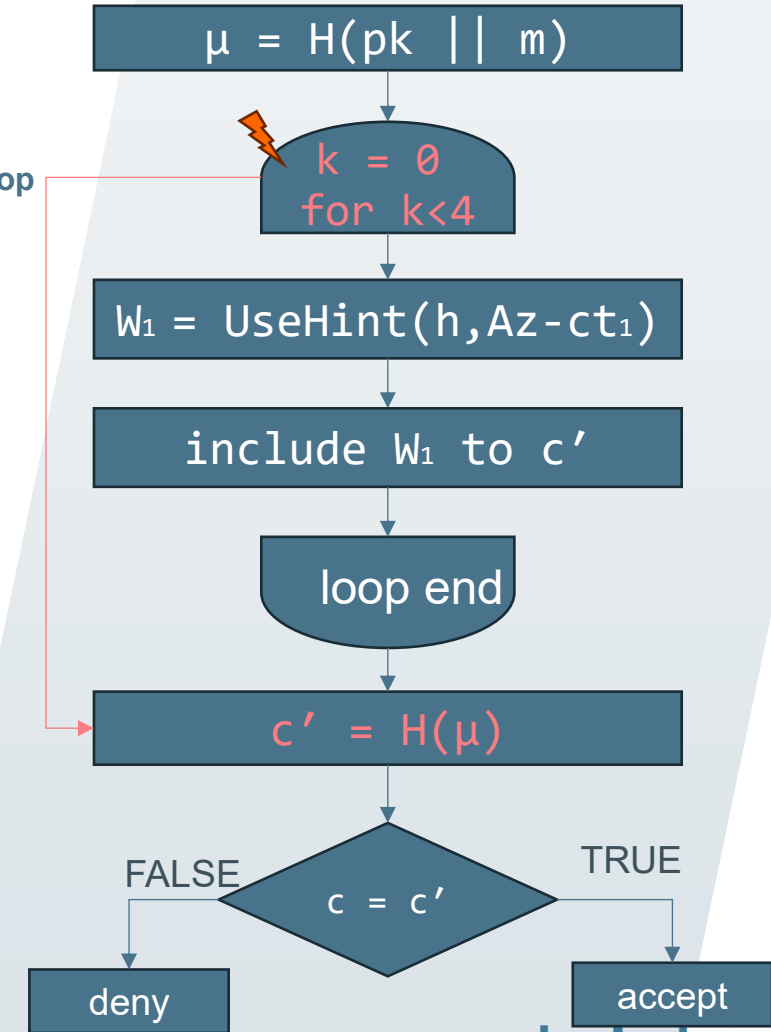
Adapting the fault attack to standardized ML-DSA

Paper ML-DSA Verification



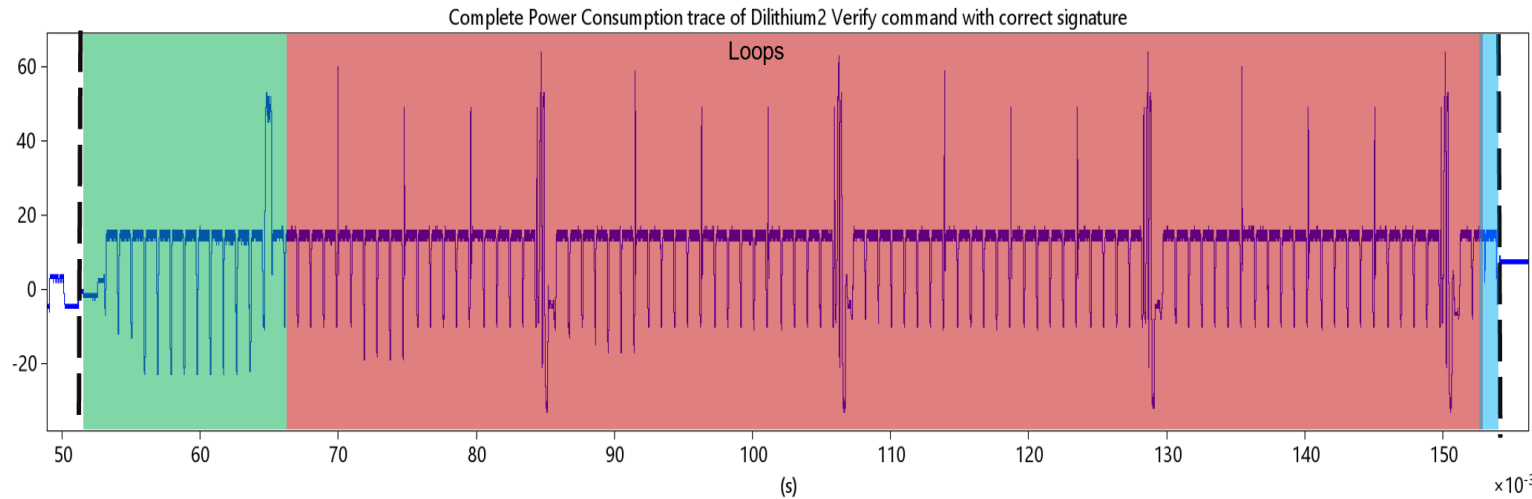
Standardized ML-DSA Verification

Fault Injection:
Skip complete Loop
execution



brightsignal

Identification phase



Power consumption trace of ML-DSA Verify

- Green = pre-loop execution
- Red = loop execution
- Blue = post-loop execution

Scans were performed targeting the whole chip surface to find:

- The best location
- The best pulser voltage and width
- The best pulse delay

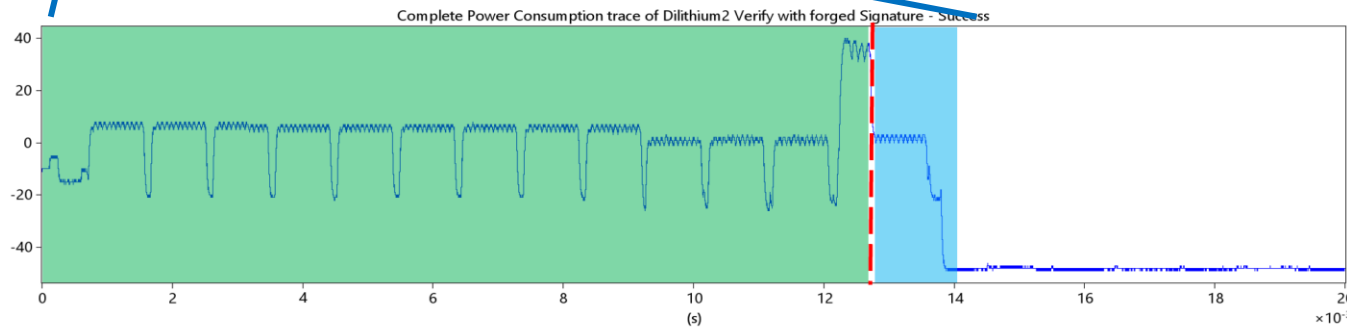
Target operation is right before red area

bright sight

Success Rate of the attack

Target command: ML-DSA Verify with a **forged** signature as input

Objective: Determine the success rate of the attack using the best parameters of the previous experiment



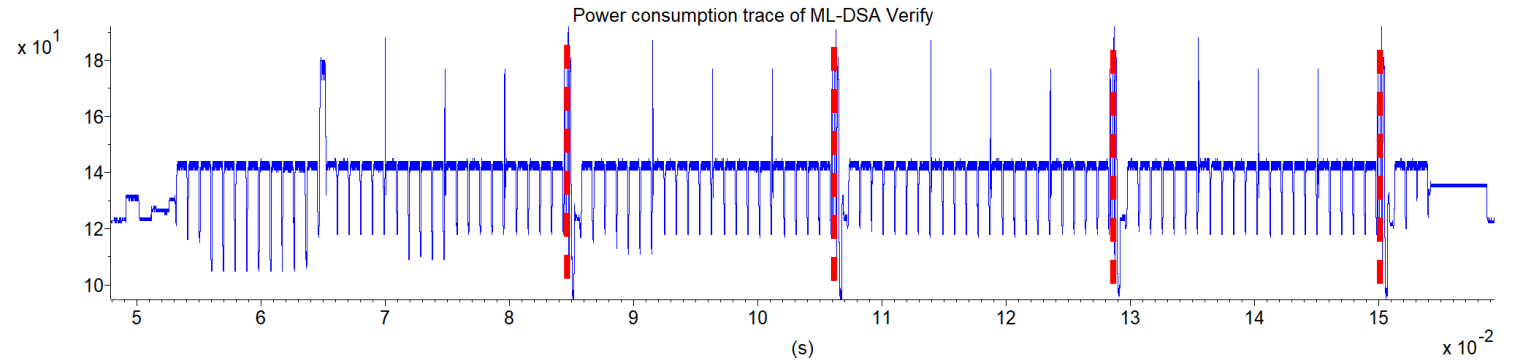
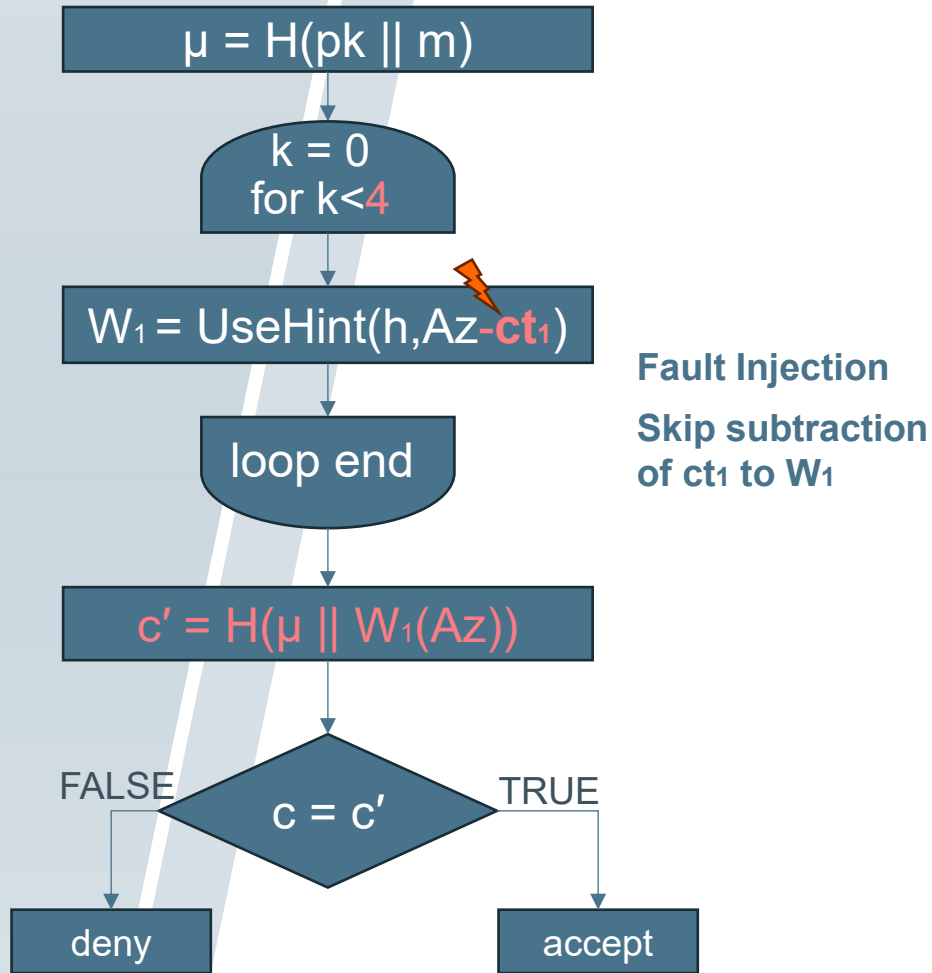
(2) Power consumption trace of ML-DSA Verify for a success

- Green = pre-loop execution
- Red = **Skipped**
- Blue = post-loop execution

Over 1000 manipulation attempts, the success rate was **92.7%**

brightsign

Second Attack scenario – Quadruple fault



Power consumption trace of ML-DSA Verify

- Red dashed lines show where the target operation is executed (4 times)
- The quadruple fault experiments were successful, the **forged signature was accepted** with a **success rate of 3%**.

ML-KEM / Kyber

FI on ML-KEM Key generation

➤ When Randomness Isn't Random, Practical Fault Attack on PQ Lattice Standards

Algorithm 13 $\text{K-PKE.KeyGen}(d)$

Uses randomness to generate an encryption key and a corresponding decryption key.

Input: randomness $d \in \mathbb{B}^{32}$.

Output: encryption key $\text{ek}_{\text{PKE}} \in \mathbb{B}^{384k+32}$.

Output: decryption key $\text{dk}_{\text{PKE}} \in \mathbb{B}^{384k}$.

```
1:  $(\rho, \sigma) \leftarrow \mathcal{G}(d \| k)$   $\triangleright$  expand 32+1 bytes to two pseudorandom 32-byte seeds1
2:  $N \leftarrow 0$ 
3: for  $(i \leftarrow 0; i < k; i++)$   $\triangleright$  generate matrix  $\hat{\mathbf{A}} \in (\mathbb{Z}_q^{256})^{k \times k}$ 
4:   for  $(j \leftarrow 0; j < k; j++)$ 
5:      $\hat{\mathbf{A}}[i, j] \leftarrow \text{SampleNTT}(\rho \| j \| i)$   $\triangleright j$  and  $i$  are bytes 33 and 34 of the input
6:   end for
7: end for
8: for  $(i \leftarrow 0; i < k; i++)$   $\triangleright$  generate  $\mathbf{s} \in (\mathbb{Z}_q^{256})^k$ 
9:    $\mathbf{s}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$   $\triangleright \mathbf{s}[i] \in \mathbb{Z}_q^{256}$  sampled from CBD
10:   $N \leftarrow N + 1$ 
11: end for
12: for  $(i \leftarrow 0; i < k; i++)$   $\triangleright$  generate  $\mathbf{e} \in (\mathbb{Z}_q^{256})^k$ 
13:    $\mathbf{e}[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$   $\triangleright \mathbf{e}[i] \in \mathbb{Z}_q^{256}$  sampled from CBD
14:   $N \leftarrow N + 1$ 
15: end for
16:  $\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$   $\triangleright$  run NTT  $k$  times (once for each coordinate of  $\mathbf{s}$ )
17:  $\hat{\mathbf{e}} \leftarrow \text{NTT}(\mathbf{e})$   $\triangleright$  run NTT  $k$  times
18:  $\hat{\mathbf{t}} \leftarrow \hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}}$   $\triangleright$  noisy linear system in NTT domain
19:  $\text{ek}_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{t}}) \| \rho$   $\triangleright$  run  $\text{ByteEncode}_{12}$   $k$  times, then append  $\hat{\mathbf{A}}$ -seed
20:  $\text{dk}_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{\mathbf{s}})$   $\triangleright$  run  $\text{ByteEncode}_{12}$   $k$  times
21: return  $(\text{ek}_{\text{PKE}}, \text{dk}_{\text{PKE}})$ 
```

Green: Initialization of noise seed and public seed

The noise seed initialization is a **single point of failure** for key generation:
Manipulating this step allows an attacker to easily **recover the key**

This is our target!

brightsignal

By **SGS**

FI on ML-KEM Key generation

- The paper demonstrates a key recovery attack scenario targeting the **noiseseed**.
- The secret can be computed with a known noiseseed, resulting in secret key recovery.
- The **noiseseed pointer initialization** can be targeted through the following fault models:
 - instruction skip on the addition of KYBER_SYMBYTES to the *noiseseed pointer
 - redirection to **constant** memory
 - redirection to **zero-filled** memory (0x00 or 0xff)

Algorithm Kyber_KeyGeneration(publicseed, noiseseed)

```
unsigned char buf [2 * KYBER_SYMBYTES];
```

```
unsigned char *publicseed = buf;
```

```
unsigned char *noiseseed = buf + KYBER_SYMBYTES;
```

```
unsigned char nonce = 0;
```

```
K512_hash_g(buf, coins, KYBER512_SYMBYTES);
```

```
for (i = 0; i < KYBER512_K; i++) {  
    k512_poly_getnoise_eta1(skpvec + i,  
        noiseseed, nonce ++);
```

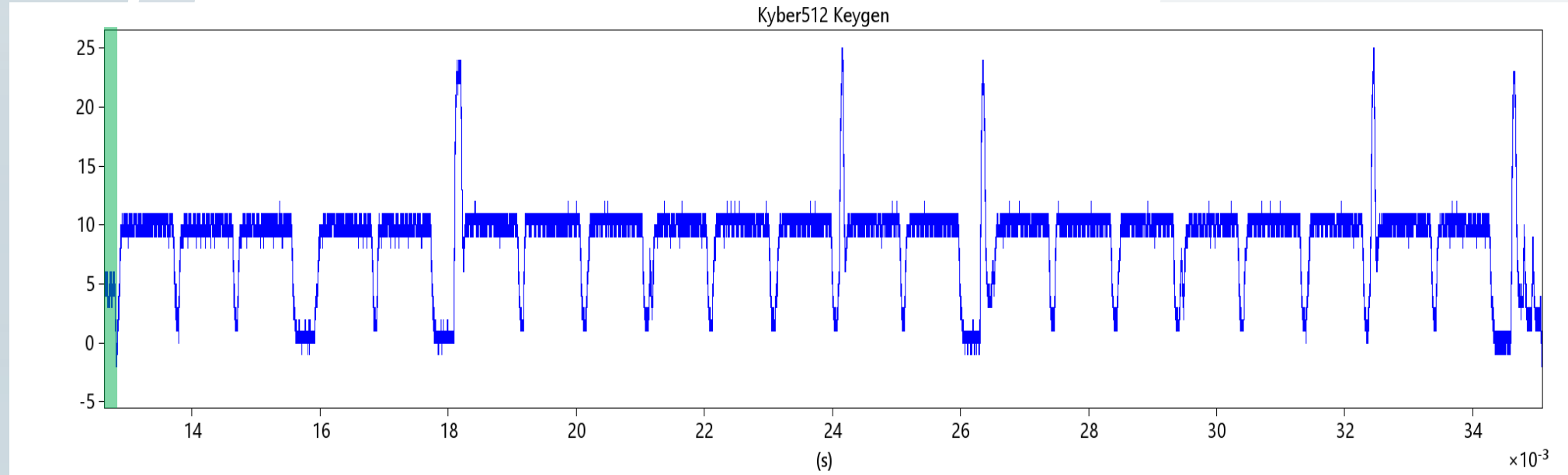
```
}
```

*relevant part for this attack

brightsign

By **SGS**

Timing identification



Green: Seed initializations = **target**

Fault models achieved

- Two fault models were identified during the experiments:
 - Corrupt** the pointer to point to **zero filled memory**
 - Skip** the noiseseed pointer update
- Respective results
 - All the bytes of **noiseseed** are equal to **0x00**
 - noiseseed** is equal to **publicseed**
- In both cases the noiseseed is therefore known by the attacker that can **recover the secret key**

```
unsigned char *noiseseed = buf + KYBER512_SYMBYTES;
80182b6: f107 0310 add.w  r3, r7, #16
80182ba: 3320     adds   r3, #32
80182bc: f8c7 3650 str.w  r3, [r7, #1616] @ 0x650
```

Manipulated assembly lines for both fault models

To obtain the **second fault model**, the timing for the attack was shifted by **~45ns (~4 clock cycles)** compared to the timing used for the **first fault model**

bright sight

Success Rates

- Success rate experiments were performed using the best parameters identified during scans; for each fault model.
- For **corrupting the noiseseed pointer** to point at a zero filled memory the success rate was **23.2%**.
- For **skipping the pointer update** of noiseseed the success rate was **21%**.

FI on ML-KEM Key generation

- Number "Not Used" Once- Practical fault attack on pqm4 implementations of NIST candidates

Algorithm 13 $K\text{-PKE.KeyGen}(d)$

Uses randomness to generate an encryption key and a corresponding decryption key.

Input: randomness $d \in \mathbb{B}^{32}$.

Output: encryption key $ek_{\text{PKE}} \in \mathbb{B}^{384k+32}$.

Output: decryption key $dk_{\text{PKE}} \in \mathbb{B}^{384k}$.

```
1:  $(\rho, \sigma) \leftarrow G(d\|k)$   $\triangleright$  expand 32+1 bytes to two pseudorandom 32-byte seeds1
2:  $N \leftarrow 0$ 
3: for  $(i \leftarrow 0; i < k; i++)$   $\triangleright$  generate matrix  $\hat{A} \in (\mathbb{Z}_q^{256})^{k \times k}$ 
4:   for  $(j \leftarrow 0; j < k; j++)$ 
5:      $\hat{A}[i, j] \leftarrow \text{SampleNTT}(\rho\|j\|i)$   $\triangleright j$  and  $i$  are bytes 33 and 34 of the input
6:   end for
7: end for
8: for  $(i \leftarrow 0; i < k; i++)$   $\triangleright$  generate  $s \in (\mathbb{Z}_q^{256})^k$ 
9:    $s[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$   $\triangleright s[i] \in \mathbb{Z}_q^{256}$  sampled from CBD
10:   $N \leftarrow N + 1$ 
11: end for
12: for  $(i \leftarrow 0; i < k; i++)$   $\triangleright$  generate  $e \in (\mathbb{Z}_q^{256})^k$ 
13:    $e[i] \leftarrow \text{SamplePolyCBD}_{\eta_1}(\text{PRF}_{\eta_1}(\sigma, N))$   $\triangleright e[i] \in \mathbb{Z}_q^{256}$  sampled from CBD
14:   $N \leftarrow N + 1$ 
15: end for
16:  $\hat{s} \leftarrow \text{NTT}(s)$   $\triangleright$  run NTT  $k$  times (once for each coordinate of  $s$ )
17:  $\hat{e} \leftarrow \text{NTT}(e)$   $\triangleright$  run NTT  $k$  times
18:  $\hat{t} \leftarrow \hat{A} \circ \hat{s} + \hat{e}$   $\triangleright$  noisy linear system in NTT domain
19:  $ek_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{t})\|\rho$   $\triangleright$  run  $\text{ByteEncode}_{12}$   $k$  times, then append  $\hat{A}$ -seed
20:  $dk_{\text{PKE}} \leftarrow \text{ByteEncode}_{12}(\hat{s})$   $\triangleright$  run  $\text{ByteEncode}_{12}$   $k$  times
21: return  $(ek_{\text{PKE}}, dk_{\text{PKE}})$ 
```

Green: noise sampling related parts

Sampling of secret s

Sampling of error e

Manipulating the noise sampling process may allow an attacker to **recover the key**

brightsign

FI on ML-KEM Key generation

- The sampling function deterministically generates a polynomial from the same noiseseed and nonce (N)
- The paper demonstrates that by causing the generation of s_0 , e_0 and s_1 , e_1 to use the same nonce values accordingly.
- Then the LWE problem that key generation relies, $t = A * s + e$ downgrades to **$t = As + s$** which is mathematically solvable.
- The paper proposes to perform instruction **skip** of the nonce incrementation **4 times**.

```
Algorithm Kyber_KeyGeneration(publicseed,  
noiseseed)
```

```
nonce = 0
```

```
 $S_0$  = get_noise(noiseseed, nonce++)
```

```
 $S_1$  = get_noise(noiseseed, nonce++)
```

```
 $A * S$  = matacc(publicseed)
```

```
 $A_0 S_0 + E_0$  = add_noise(noiseseed, nonce++)
```

```
 $A_1 S_1 + E_1$  = add_noise(noiseseed, nonce++)
```

*relevant part for this attack

bright sight

By SGS

Fault models identified to make $S_0 = E_0$ and $S_1 = E_1$

- Single fault: Setting **nonce 3** to 0.
 - Result: $S_0 = E_0$ and $S_1 = E_1$
- Triple fault: The **nonce incrementations for nonce 2, nonce 3 and nonce 4 are skipped**.
 - Result: $S_0 = E_0 = S_1 = E_1$
- With the above fault models the LWE problem $t = A * S + e$, with two unknowns S and E becomes $t = A * S + S$ with only S unknown which can be solved mathematically.

```
Algorithm Kyber_KeyGeneration(publicseed,
noiseseed)
nonce = 0
S0 = get_noise(noiseseed, nonce++)
S1 = get_noise(noiseseed, nonce++)

A * S = matacc(publicseed)

A0S0 + E0 = add_noise(noiseseed, nonce++)
A1S1 + E0 = add_noise(noiseseed, nonce++)
```

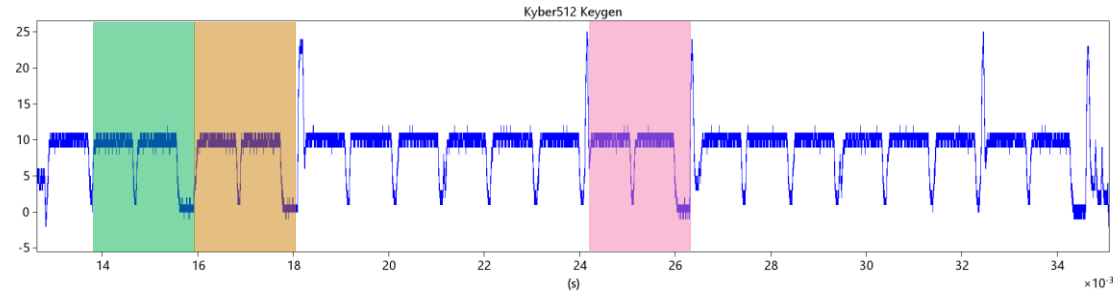
Annotations in the original image:

- nonce 1 assignment (green text, arrow to `nonce = 0`)
- nonce 2 assignment (yellow text, arrow to `nonce++` in `S0 = get_noise`)
- nonce 3 assignment (pink text, arrow to `nonce++` in `S1 = get_noise`)
- nonce 4 assignment (black text, arrow to `nonce++` in `A0S0 + E0 = add_noise`)

*relevant part for this attack

brightsignal

Timing identification and results



- First noise sampling call with nonce 1
- Second noise sampling call with nonce 2
- Third sampling call with nonce 3

Single fault scenario

- The target is **right before the pink area**
- nonce 3 was forced to zero with a success rate of 14.6%.

Triple fault scenario

- The targets are **right before each of the coloured area**
- The three consecutive incrementations were skipped with a success rate of 1.9%

Conclusion

Attack	Algorithm	Contribution	Success rate
Forging Dilithium and Falcon Signatures by Single Fault Injection [1]	ML-DSA	• New attack path aiming at bypassing the loop inside Verify • A quadruple fault attack was conducted to match the effect of the attack proposed in the paper	Single fault: 92,7% Quadruple fault: 3%
When Randomness Isn't Random, Practical Fault Attack on PQ Lattice Standards [2]	ML-KEM	• Reproduced two fault models proposed in the paper with EMFI	1st fault model: 23.2% 2nd fault model: 21%
Number "Not Used" Once- Practical fault attack on pqm4 implementations of NIST candidates [3]	ML-KEM	• Adapted paper attack scenario from quadruple fault to triple fault • New attack path to threaten the algorithm with single fault • Both attacks were successfully exploited and led to key recovery	Single fault: 14.6% Triple fault: 1.9%

References

- [1] “Forging Dilithium and Falcon Signatures by Single Fault Injection”, Sven Bauer, Fabrizio De Santis, FDTTC 2023
- [2] “When randomness Isn't Random: Practical Fault Attack on PQ Lattice Standards”, Hariprasad Kelassery, Prasanna Ravi, Shivam Bhasin, IACR 2025
- [3] “Number Not Used Once – Practical fault attack on pqm4 implementations of NIST candidates”, Prasanna Ravi et al., IACR 2018
- [4] “Safe-Error Analysis of Post-Quantum Cryptography Mechanisms”, Luke Bettale et al., IACR 2021
- [5] “Carry Your Fault: A Fault Propagation Attack on Side-Channel Protected LWE-based KEM“, Suparna Kundu et al., TCHES 2024

bright sight

Questions?

bright sight

By **SGS**



Thank you!

Do you have any questions?

thomas.techer@sgs.com

www.brightsight.com

brightsight

By **SGS**