

# Post Quantum Cryptography

DFA on PQC (Dilithium) Implementation for Key Recovery

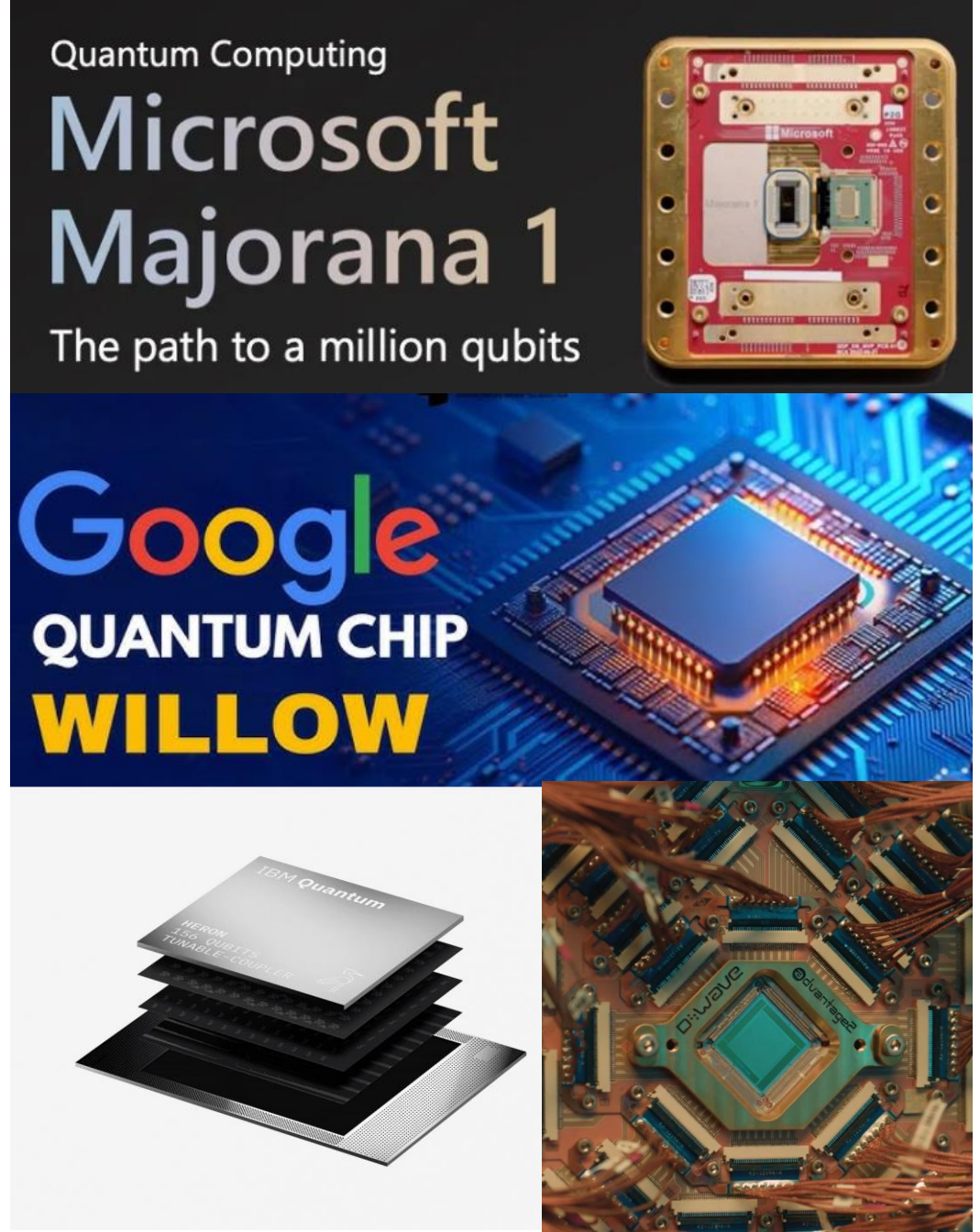
## Q-DAY

Quantum Computer advancing fast

- Quantum chips
- Quantum computers: D-Wave: quantum annealing computers
- PQC is new encryption algorithms designed to resist attacks from quantum computers.
- Post Quantum Cryptography will eventually replace classical cryptography used today

Q-day is the day someone builds a quantum computer that can crack the most widely used forms of encryption. Preparation date was set at 2035.

**Q-Day is sooner than expected!**





# Standardization and regulatory developments

## NIST recommendations

- ML-KEM (Kyber) for key agreement (FIPS-203)
- ML-DSA (Dilithium) for digital signatures (FIPS-204)
- SLH-DSA (SPHINCS+) for digital signatures (FIPS-205)
- HQC algorithm was selected March 11, 2025 as a backup KEM
- Flacon to be published soon

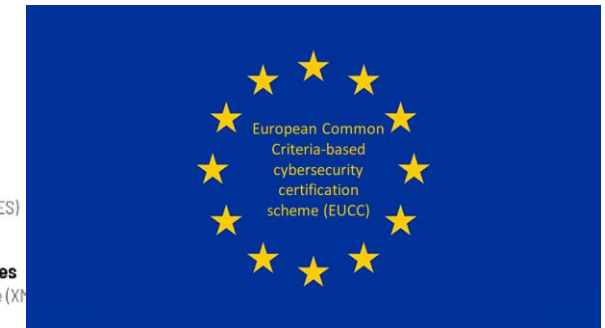
## ISO standards

- eXtended Merkle Signature Scheme (XMSS)- 2018
- Leighton-Micali hash-based Signature (LMS) - 2019
- TVLA
- Note: ISO published the stateful hash-based signatures – 2020

## EUCC

- The PQC algorithms is now a guidance document for the European certification scheme EUCC

## Adoption

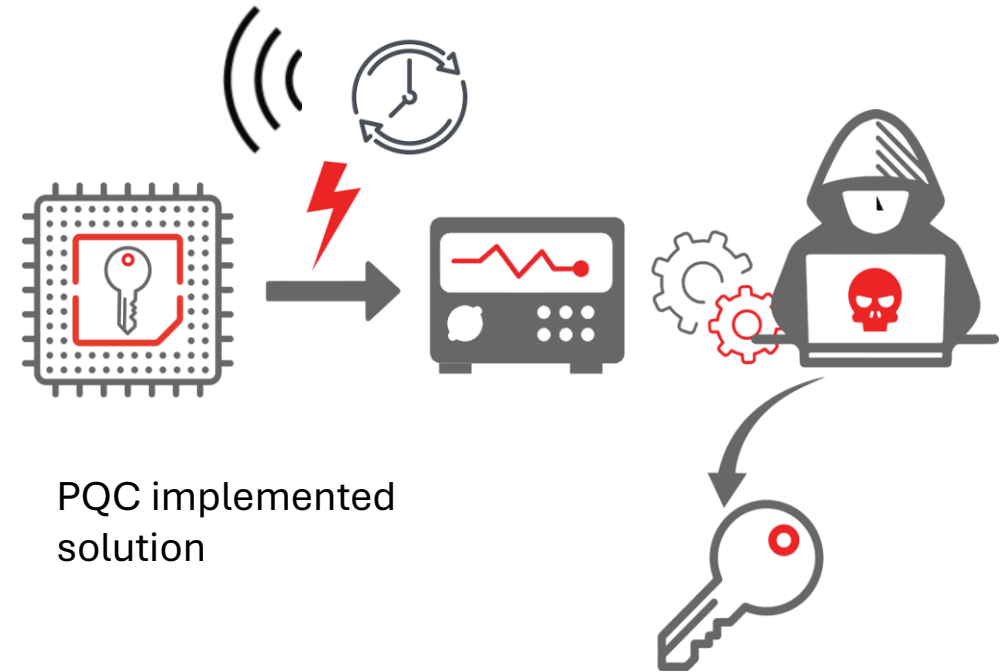


**Regulatory requirements and compliance is emerging**

# WHY SECURITY MATTERS DESPITE MIGRATION TO PQC?

## Key risks for PQC implementations

- Quantum Computers are still in early development, but PQC is already required now to protect against "Store now Decrypt later" attacks.
- In practice, as with traditional Ciphers (AES, RSA, ECC, ..) PQC implementations **are proven to be vulnerable** to **Side-Channel** attacks and **Fault Injection** (FI) attacks!



**Secure algorithm != Secure Implementation**

The background is a dark, abstract composition. It features a dense network of thin, glowing purple lines that originate from a central point at the top and fan out towards the bottom. Interspersed among these lines are numerous small, bright red dots. The overall effect is one of dynamic energy and complex connectivity, reminiscent of a neural network or a data visualization.

**FI on Dilithium**

# Dilithium

Dilithium PQC algorithm has 3 main stages:

- Key Generation:
  - public key ( $pk$ )
  - private key ( $sk$ )
- Signature of a message
- Verification of the message

Gen

```
01  $\mathbf{A} \leftarrow R_q^{k \times \ell}$   
02  $(s_1, s_2) \leftarrow S_\eta^\ell \times S_\eta^k$   
03  $\mathbf{t} := \mathbf{A}s_1 + s_2$   
04 return  $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, s_1, s_2))$ 
```

Sign( $sk, M$ )

```
05  $\mathbf{z} := \perp$   
06 while  $\mathbf{z} = \perp$  do  
07    $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$   
08    $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$   
09    $c \in B_r := H(M \parallel \mathbf{w}_1)$   
10    $\mathbf{z} := \mathbf{y} + cs_1$   
11   if  $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$  or  $\|\text{LowBits}(\mathbf{A}\mathbf{y} - cs_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ , then  $\mathbf{z} := \perp$   
12 return  $\sigma = (\mathbf{z}, c)$ 
```

Verify( $pk, M, \sigma = (\mathbf{z}, c)$ )

```
13  $\mathbf{w}'_1 := \text{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$   
14 if return  $\llbracket \|\mathbf{z}\|_\infty < \gamma_1 - \beta \rrbracket$  and  $\llbracket c = H(M \parallel \mathbf{w}'_1) \rrbracket$ 
```

# Dilithium SIGNING

In this demonstration we aim to corrupt the signature process and then recover the secret key using advanced mathematics.

- Signatures rely on a commitment value  $y$  at line 10.
- Line 10 uses the private key  $s_1$
- Attacker needs to recover  $s_1$

Gen

```
01  $\mathbf{A} \leftarrow R_q^{k \times \ell}$   
02  $(s_1, s_2) \leftarrow S_\eta^\ell \times S_\eta^k$   
03  $\mathbf{t} := \mathbf{A}s_1 + s_2$   
04 return  $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, s_1, s_2))$ 
```

Sign( $sk, M$ )

```
05  $\mathbf{z} := \perp$   
06 while  $\mathbf{z} = \perp$  do  
07    $y \leftarrow S_{\gamma_1 - 1}^\ell$   
08    $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}y, 2\gamma_2)$   
09    $c \in B_\tau := H(M \parallel \mathbf{w}_1)$   
10    $\mathbf{z} := y + cs_1$   
11   if  $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$  or  $\|\text{LowBits}(\mathbf{A}y - cs_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ , then  $\mathbf{z} := \perp$   
12 return  $\sigma = (\mathbf{z}, c)$ 
```

Verify( $pk, M, \sigma = (\mathbf{z}, c)$ )

```
13  $\mathbf{w}'_1 := \text{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$   
14 if return  $\llbracket \|\mathbf{z}\|_\infty < \gamma_1 - \beta \rrbracket$  and  $\llbracket c = H(M \parallel \mathbf{w}'_1) \rrbracket$ 
```

# Fault Target

- Case 1:
  - Sampling of  $y$  vector is skipped – zero vector
  - Line 10: becomes
    - $z = c * s_1$
  - $s_1$  can be trivially computed
    - $s_1 = z * c^{-1}$
- Case 2:
  - Some of the coefficients in polynomial is set, rest are 0
  - Polynomials are order of  $n' \ll n$
  - $s_1$  still can be recovered

Gen

```

01  $\mathbf{A} \leftarrow R_q^{k \times \ell}$ 
02  $(s_1, s_2) \leftarrow S_\eta^\ell \times S_\eta^k$ 
03  $\mathbf{t} := \mathbf{A}s_1 + s_2$ 
04 return  $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, s_1, s_2))$ 



```

Sign( $sk, M$ )

```

05  $\mathbf{z} := \perp$ 
06 while  $\mathbf{z} = \perp$  do
07    $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$ 
08    $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$ 
09    $c \in B_T := H(M \parallel \mathbf{w}_1)$ 
10    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ 
11   if  $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$  or  $\|\text{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ , then  $\mathbf{z} := \perp$ 
12 return  $\sigma = (\mathbf{z}, c)$ 

```

  **2 cases**

Verify( $pk, M, \sigma = (\mathbf{z}, c)$ )

```

13  $\mathbf{w}'_1 := \text{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$ 
14 if return  $\llbracket \|\mathbf{z}\|_\infty < \gamma_1 - \beta \rrbracket$  and  $\llbracket c = H(M \parallel \mathbf{w}'_1) \rrbracket$ 

```



Thomas Espitau<sup>4</sup>, Pierre-Alain Fouque<sup>2</sup>,  
Benoît Gérard<sup>1</sup>, and Mehdi Tibouchi<sup>3</sup>

<sup>1</sup> DGA-MI & IRISA <sup>†</sup>

<sup>2</sup> NTT Secure Platform Laboratories <sup>‡</sup>

<sup>3</sup> Institut Universitaire de France & IRISA & Université de Rennes I <sup>¶</sup>

<sup>4</sup> Ecole Normale Supérieure de Cachan & Sorbonne Universités, UPMC Univ Paris 06, LIP6<sup>§</sup>

**Abstract.** As the advent of general-purpose quantum computers appears to be drawing closer, agencies and advisory bodies have started recommending that we prepare the transition away from factoring and discrete logarithm-based cryptography, and towards postquantum secure constructions, such as lattice-based schemes.

Almost all primitives of classical cryptography (and more!) can be realized with lattices, and the efficiency of primitives like encryption and signatures has gradually improved to the point that key sizes are competitive with RSA at similar security levels, and fast performance can be achieved both in software and hardware. However, little research has been conducted on physical attacks targeting concrete implementations of postquantum cryptography in general and lattice-based schemes in particular, and such research is essential if lattices are going to replace RSA and elliptic curves in our devices and smart cards.

In this paper, we look in particular at fault attacks against implementations of lattice-based signature schemes, looking both at Fiat-Shamir type constructions (particularly BLISS, but also GLP, PASSing and Ring-TESLA) and at hash-and-sign schemes (particularly the GPV-based scheme of Ducas-Prest-Lyubashevsky). These schemes include essentially all practical lattice-based signatures, and achieve the best efficiency to date in both software and hardware. We present several fault attacks against those schemes yielding a full key recovery with only a few or even a single faulty signature, and discuss possible countermeasures to protect against these attacks.

**Keywords:** Fault Attacks, Digital Signatures, Postquantum Cryptography, Lattices, BLISS, GPV.

## 1 Introduction

**Lattice-based cryptography.** Recent progress in quantum computation [11], the NSA advisory memorandum recommending the transition away from Suite B and to postquantum cryptography [1], as well as the announcement of the NIST standardization process for postquantum cryptography [9] all suggest that research on postquantum schemes, which is already plentiful but mostly focused on theoretical constructions and asymptotic security, should increasingly take into account real world implementation issues.

Among all postquantum directions, lattice-based cryptography occupies a position of particular interest, as it relies on well-studied problems and comes with uniquely strong security guarantees, such as worst-case to average-case reductions [43]. A number of works have also focused on improving the performance of lattice-based schemes, and actual implementation results suggest that properly optimized schemes may be competitive with, or even outperform, classical factoring- and discrete logarithm-based cryptography.

## Math behind Loop-Abort Fault

- If order of polynomial  $y_i \in R_q$ , is  $n' \ll n$ , then the problem can be reduced to a lattice reduction problem
- Attacker knows  $z$  and challenge vector  $c$
- Sampling of  $y_i$  was aborted early and need to recover  $s_1$
- The theory attack require knowledge of  $n'$ , but in our practical attack we can guess  $n'$  with high a probability by observing coefficients of polynomials in  $z$
- No matter the security level
- No matter deterministic or non-deterministic version

# Lattice Reduction Approach

The mathematics used by our attack modules

- Consider:

$$z_i = y_i + c * s_{1,i}$$

$$z_i * c^{-1} = y_i * c^{-1} + s_{1,i}$$

$$z_i * c^{-1} - s_{1,i} = y_i * c^{-1} \equiv \sum_{j=0}^{m-1} y_{i,j} * c^{-1} * x^j$$

$m$  - index of first faulted coefficient

$i$  - polynomial index in vector

$j$  - coefficient index in polynomial

# Target Function

- Each polynomial in vector  $y$  is sampled in the loop
- Sampled values are unpacked into the polynomial structure

Gen

```
01  $A \leftarrow R_q^{k \times \ell}$   
02  $(s_1, s_2) \leftarrow S_\eta^\ell \times S_\eta^k$   
03  $t := As_1 + s_2$   
04 return  $(pk = (A, t), sk = (A, t, s_1, s_2))$ 
```

Sign( $sk, M$ )

```
05  $z := \perp$   
06 while  $z = \perp$  do  
07    $y \leftarrow S_{\gamma_1 - 1}^\ell$   
08    $w_1 := \text{HighBits}(Ay, 2\gamma_2)$   
09    $c \in B_\tau := H(M \parallel w_1)$   
10    $z := y + cs_1$   
11   if  $\|z\|_\infty \geq \gamma_1 - \beta$  or  $\|\text{LowBits}(Ay - cs_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ , then  $z := \perp$   
12 return  $\sigma = (z, c)$ 
```

Verify( $pk, M, \sigma = (z, c)$ )

```
13  $w'_1 := \text{HighBits}(Az - ct, 2\gamma_2)$   
14 if return  $\llbracket \|z\|_\infty < \gamma_1 - \beta \rrbracket$  and  $\llbracket c = H(M \parallel w'_1) \rrbracket$ 
```

```
void polyvec1_uniform_gamma1(polyvec1 *v, const uint8_t seed[CRHBYTES], uint16_t nonce) {  
    unsigned int i;  
  
    for(i = 0; i < L; ++i)  
        poly_uniform_gamma1(&v->vec[i], seed, L*nonce + i);  
}  
  
#define POLY_UNIFORM_GAMMA1_NBLOCKS ((POLYZ_PACKEDBYTES + STREAM256_BLOCKBYTES - 1)/STREAM256_BLOCKBYTES)  
void poly_uniform_gamma1(poly *a,  
    const uint8_t seed[CRHBYTES],  
    uint16_t nonce)  
{  
    uint8_t buf[POLY_UNIFORM_GAMMA1_NBLOCKS*STREAM256_BLOCKBYTES];  
    stream256_state state;  
  
    stream256_init(&state, seed, nonce);  
    stream256_squeezeblocks(buf, POLY_UNIFORM_GAMMA1_NBLOCKS, &state);  
    polyz_unpack(a, buf);  
}
```

# Target Function

- Goal :
  - Abort Loop as soon as possible in *polyz\_unpack* function
- No matter the medium:
  - Voltage FI
  - Electromagnetic FI
  - Laser FI

```
void polyz_unpack(poly *r, const uint8_t *a) {
    unsigned int i;
    DBENCH_START();

    #if GAMMA1 == (1 << 17)
        for(i = 0; i < N/4; ++i) {
            r->coeffs[4*i+0] = a[9*i+0];
            r->coeffs[4*i+0] |= (uint32_t)a[9*i+1] << 8;
            r->coeffs[4*i+0] |= (uint32_t)a[9*i+2] << 16;
            r->coeffs[4*i+0] &= 0x3FFFF;

            r->coeffs[4*i+1] = a[9*i+2] >> 2;
            r->coeffs[4*i+1] |= (uint32_t)a[9*i+3] << 6;
            r->coeffs[4*i+1] |= (uint32_t)a[9*i+4] << 14;
            r->coeffs[4*i+1] &= 0x3FFFF;

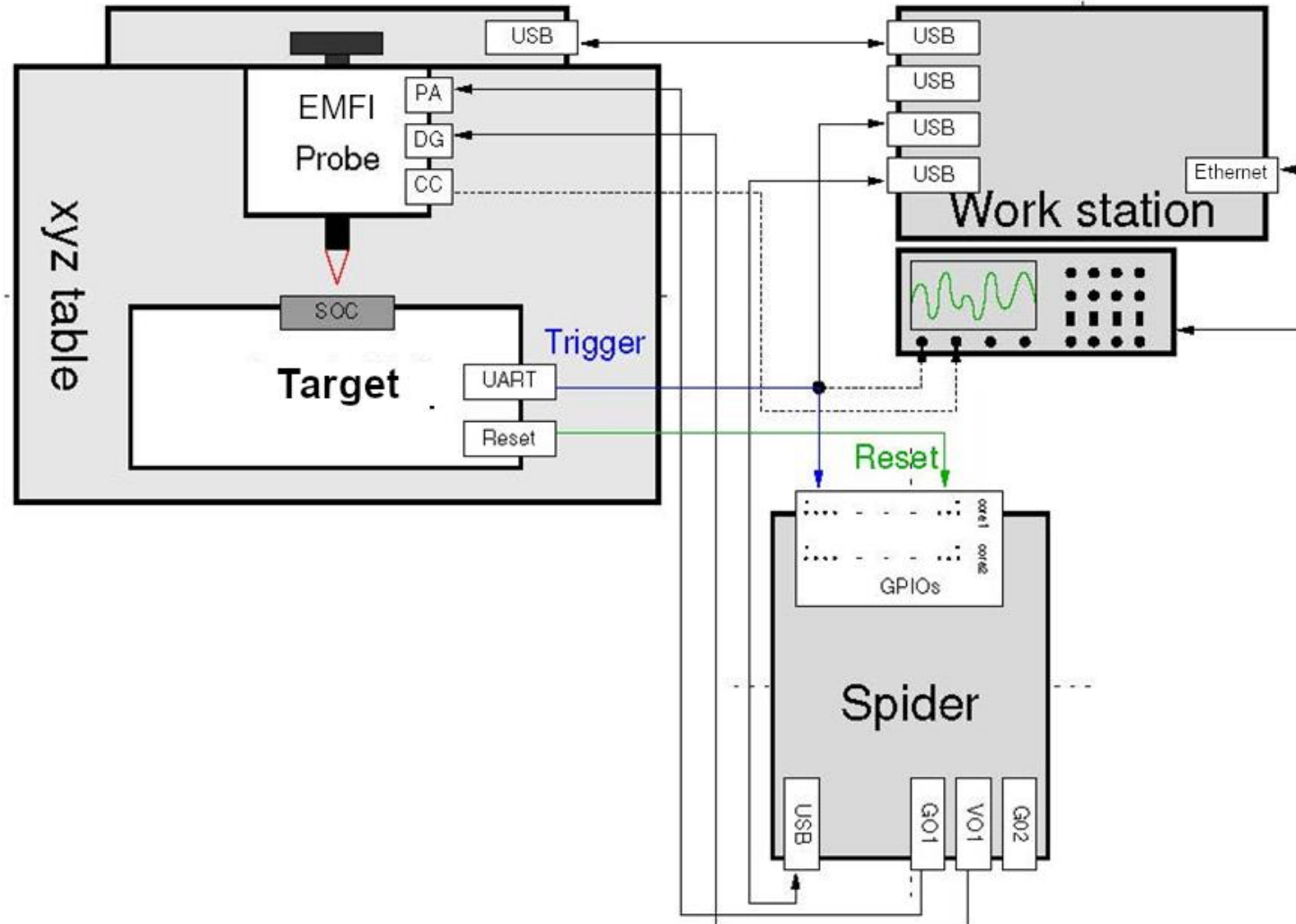
            r->coeffs[4*i+2] = a[9*i+4] >> 4;
            r->coeffs[4*i+2] |= (uint32_t)a[9*i+5] << 4;
            r->coeffs[4*i+2] |= (uint32_t)a[9*i+6] << 12;
            r->coeffs[4*i+2] &= 0x3FFFF;

            r->coeffs[4*i+3] = a[9*i+6] >> 6;
            r->coeffs[4*i+3] |= (uint32_t)a[9*i+7] << 2;
            r->coeffs[4*i+3] |= (uint32_t)a[9*i+8] << 10;
            r->coeffs[4*i+3] &= 0x3FFFF;

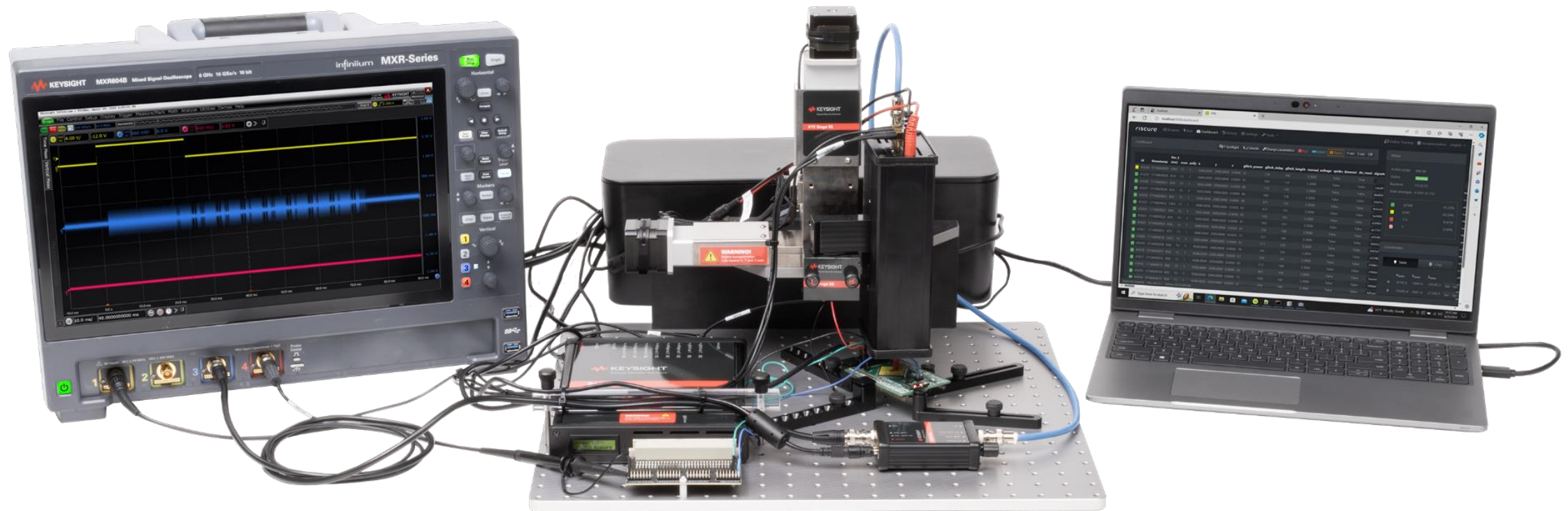
            r->coeffs[4*i+0] = GAMMA1 - r->coeffs[4*i+0];
            r->coeffs[4*i+1] = GAMMA1 - r->coeffs[4*i+1];
            r->coeffs[4*i+2] = GAMMA1 - r->coeffs[4*i+2];
            r->coeffs[4*i+3] = GAMMA1 - r->coeffs[4*i+3];
        }
    }
```



## Setup used



## Setup used

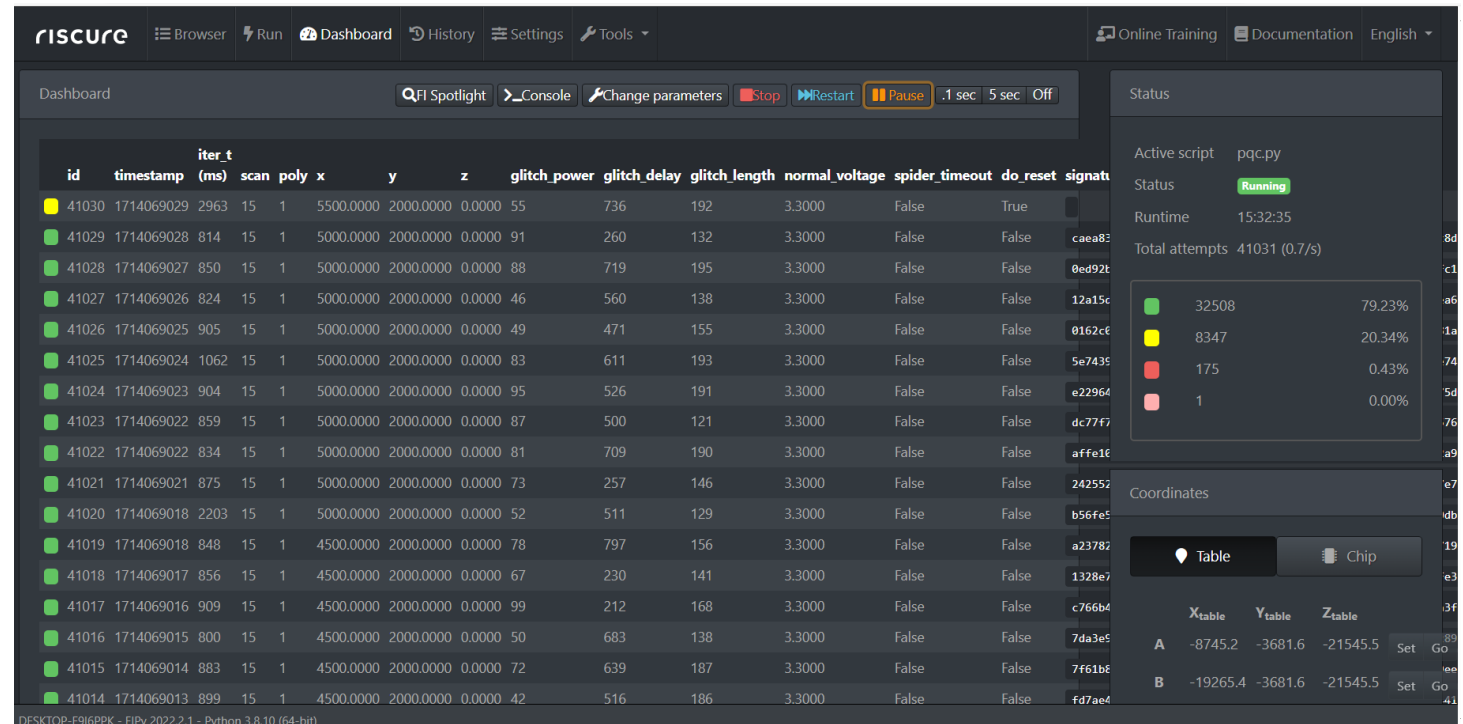


## Differential Fault Attack on Dilithium (video)

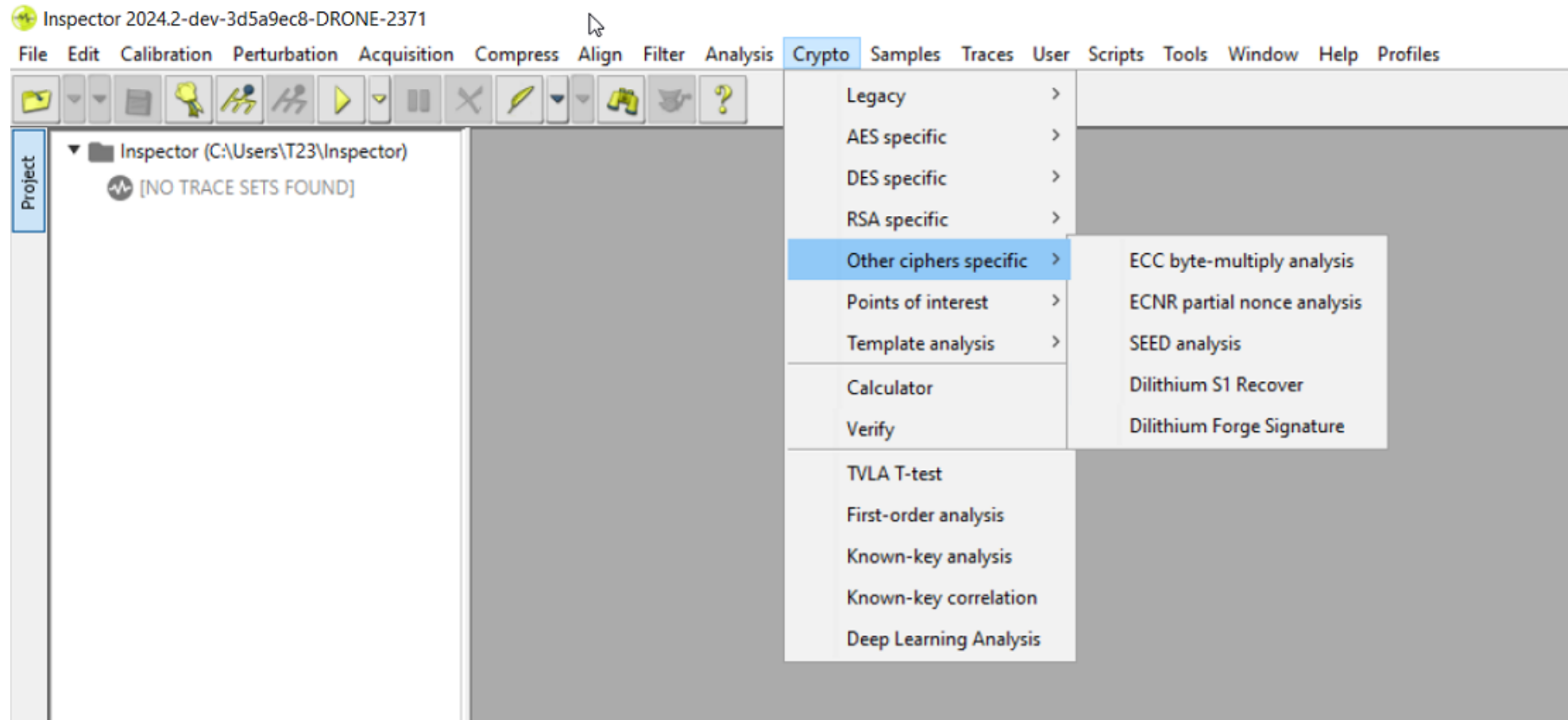
# Fault injection campaign

After obtaining a few successful glitches we apply math to recover the secret key

- Green: Glitch attempts were the target returned the expected response
- Yellow: The target mutes
- Red: Successful glitches where the result of the target function has changed



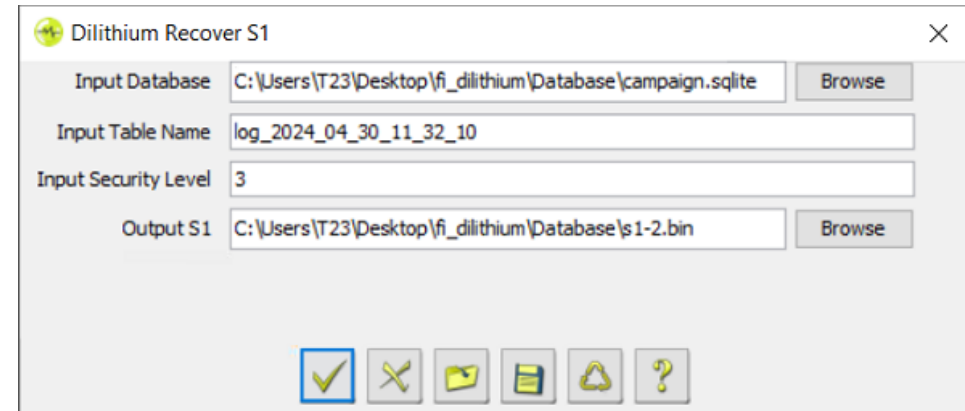
# Recovering private key





## Recovering private key

- We need to point to the database containing the faulty signatures
- We need to say where to place the recovered private signature



## Forging a Signature

- Recovered  $s_1$  can be used to sign arbitrary message

$$t = As_1 + s_2$$

$$t_1 * 2^d + t_0 = As_1 + s_2$$

$$A * s_1 - t_1 * 2^d = t_0 - s_2$$

$A, t_1, d$  - public knowledge  
 $t_0, s_2$  - private knowledge  
 $s_1$  - recovered

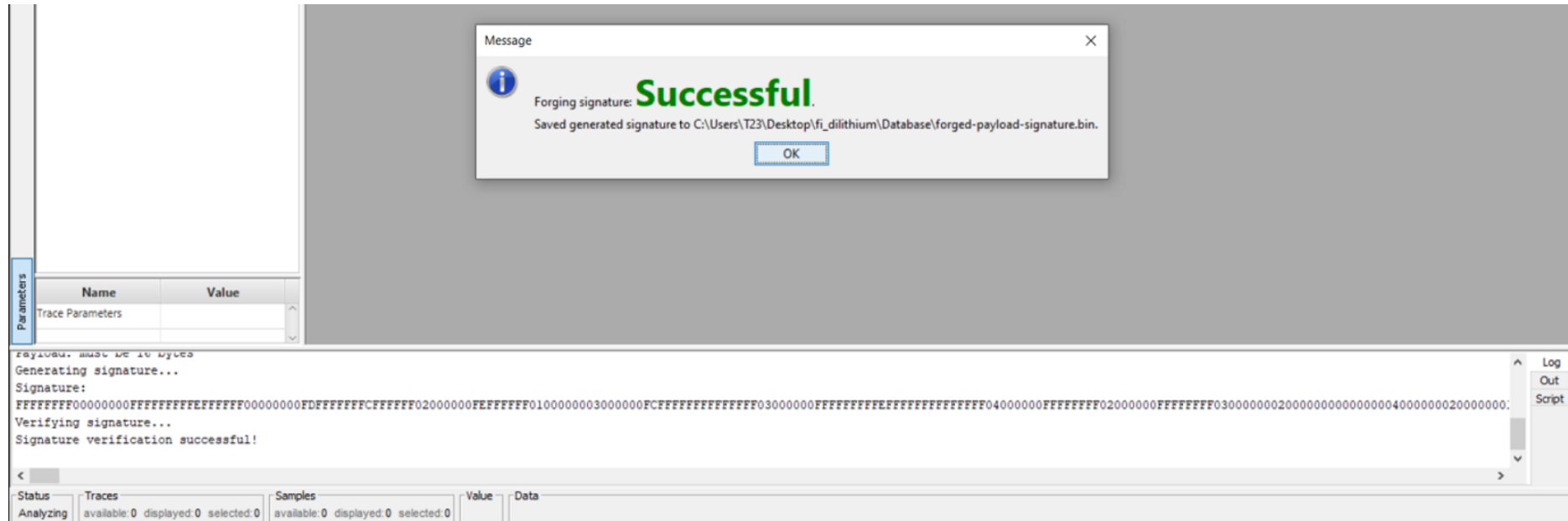
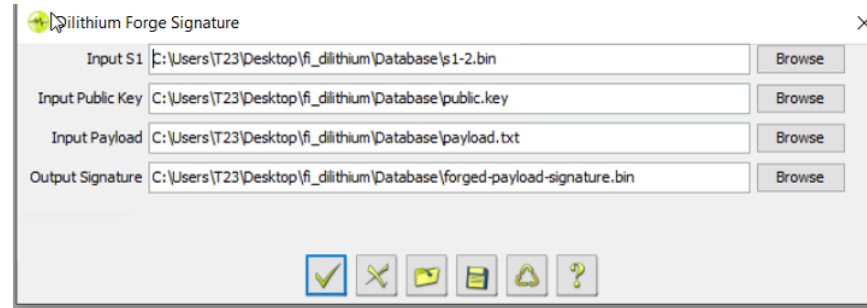
- Attacker can generate arbitrary  $s_2'$  small polynomial and calculate new  $t_0'$

$$A * s_1 - t_1 * 2^d + s_2' = t_0'$$

- If  $t_0' - s_2' = A * s_1 - t_1 * 2^d$  secret key is **successfully forged**

# Forging a Message

The recover signature allows to forge new msgs/payload.



## Conclusions

- Secure algorithm  $\neq$  secure implementation.
- Cryptographic implementations require thorough security testing to avoid surprises.
- Dilithium is still a secure algorithm but needs to be implemented properly with countermeasures.



# Thank you



[Durga.lakshmi-ramachandran@keysight.com](mailto:Durga.lakshmi-ramachandran@keysight.com)