



AUGUST 6-7, 2025

MANDALAY BAY / LAS VEGAS

Bypassing PQC Signature Verification with Fault Injection: Dilithium, XMSS, SPHINCS+

Fikret Garipay

Hello!

- Security Engineer at Keysight
Device Security Testing
- Passionate about software
exploitation and hardware attacks
- Twitter: @erd0spy



Fikret Garipay

Agenda

- Introduction to Post Quantum Cryptography
- Target Implementation
- Voltage Fault Injection in Practice
- Fault Injection Attacks on Dilithium Verification
- Fault Injection Attacks on WOTS+ in XMSS and SPHINCS+
- Fault Injection on Fault Resistance XMSS Library
- Key Takeaways and Conclusions

Introduction to Post Quantum Cryptography

Post-Quantum Crypto Is Getting Real

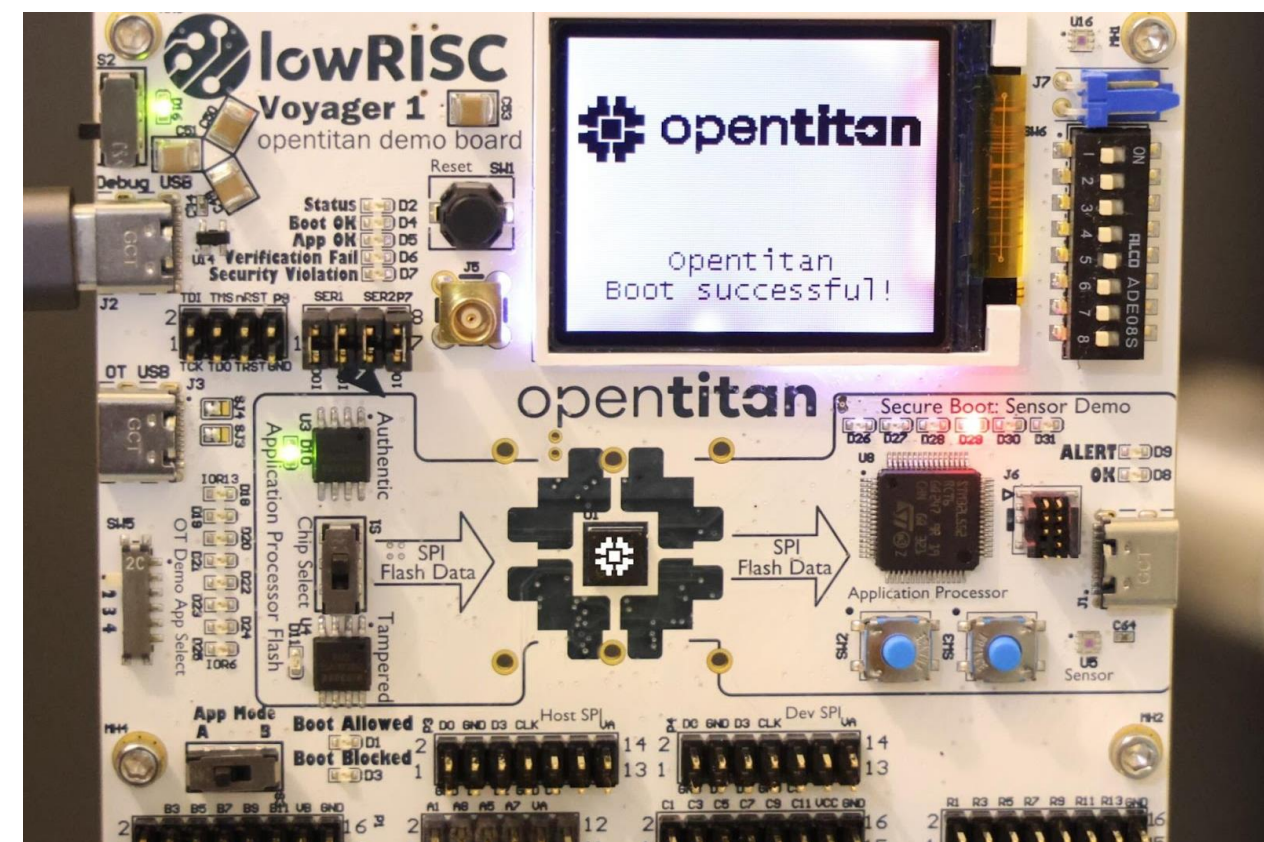
- Quantum computers aren't breaking crypto yet.
- The shift is underway with **standards, vendors, firmware**
- PQC is set to replace RSA, ECC in **secure boot, firmware signing, and more.**
- That makes PQC **fresh attack surface.**

PQC Signatures Are Becoming Global Standards

Algorithm	Signature Scheme Type	CNSA 2.0 (NSA)	Standard
Dilithium	Lattice-based	Required for all digital signatures (general use)	NIST FIPS 204 (ML-DSA)
LMS	Stateful hash-based	Approved for firmware/software signing	ISO/IEC 14888-4:2024
XMSS	Stateful hash-based	Approved for firmware/software signing	ISO/IEC 14888-4:2024
SPHINCS+	Stateless hash-based	Not approved for any use in NSS	NIST FIPS 205 (SLH-DSA)

PQC Signatures in Industry

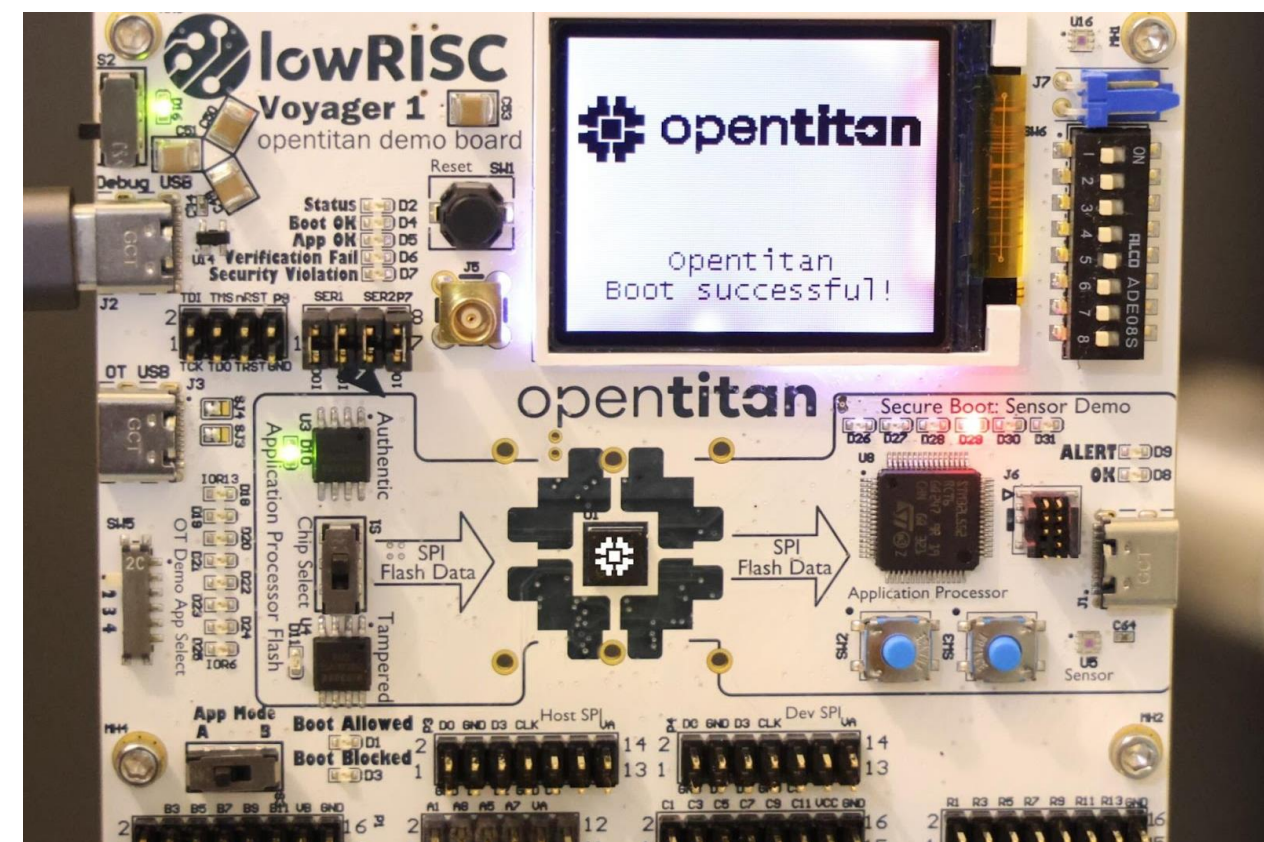
- Multiple vendors now offer PQC solutions for **Firmware Update, Secure Boot, Signature Verification**



Source: [Fabrication begins for production OpenTitan silicon](#)

PQC Signatures in Industry

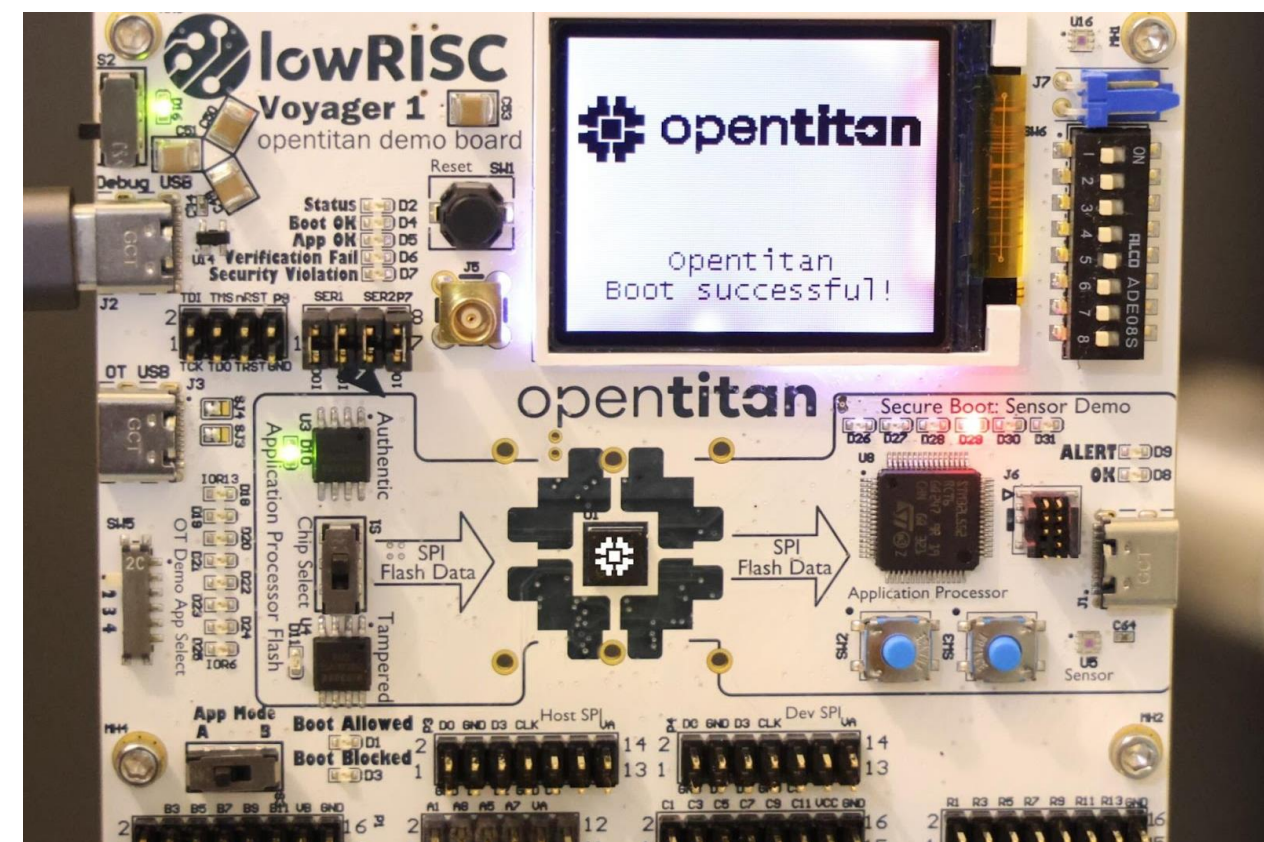
- Multiple vendors now offer PQC solutions for **Firmware Update, Secure Boot, Signature Verification**
- **OpenTitan** chip uses **SPHINCS+** for PQC secure boot



Source: [Fabrication begins for production OpenTitan silicon](#)

PQC Signatures in Industry

- Multiple vendors now offer PQC solutions for **Firmware Update, Secure Boot, Signature Verification**
- OpenTitan chip uses **SPHINCS+** for PQC secure boot
- **Caliptra 2.0** is adding PQC secure boot with **Dilithium** and **Kyber**



Source: [Fabrication begins for production OpenTitan silicon](#)

Making the Attacks Real

- We reviewed dozens of papers
- Focused on **practical attacks** AGAINST PQC signature verification

Making the Attacks Real

- We reviewed dozens of papers
- Focused on **practical attacks** AGAINST PQC signature verification
- Public PQC targets aren't widely deployed yet
- So, we:
 - Ported public PQC libs to bare-metal firmware

Target Implementation

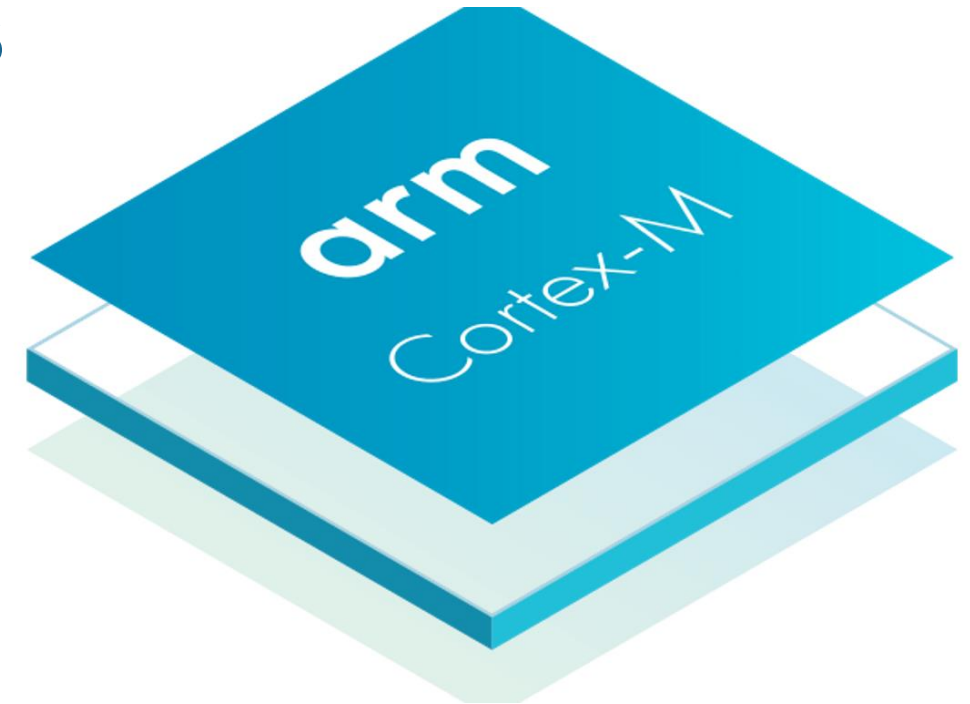
Target Implementation – Libraries

Dilithium: [pqm4](#)

- PQC library for the ARM Cortex-M4

XMSS: [xmss-reference](#)

SPHINCS+: [sphincsplus](#)



Target Implementation – Libraries

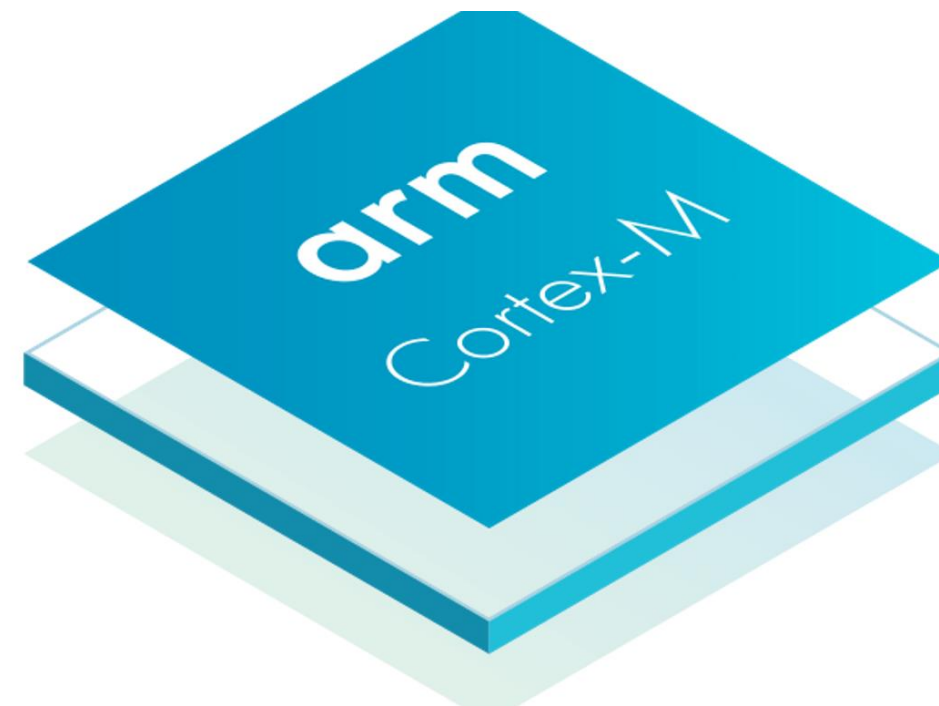
Dilithium: [pqm4](#)

- PQC library for the ARM Cortex-M4

XMSS: [xmss-reference](#)

SPHINCS+: [sphincsplus](#)

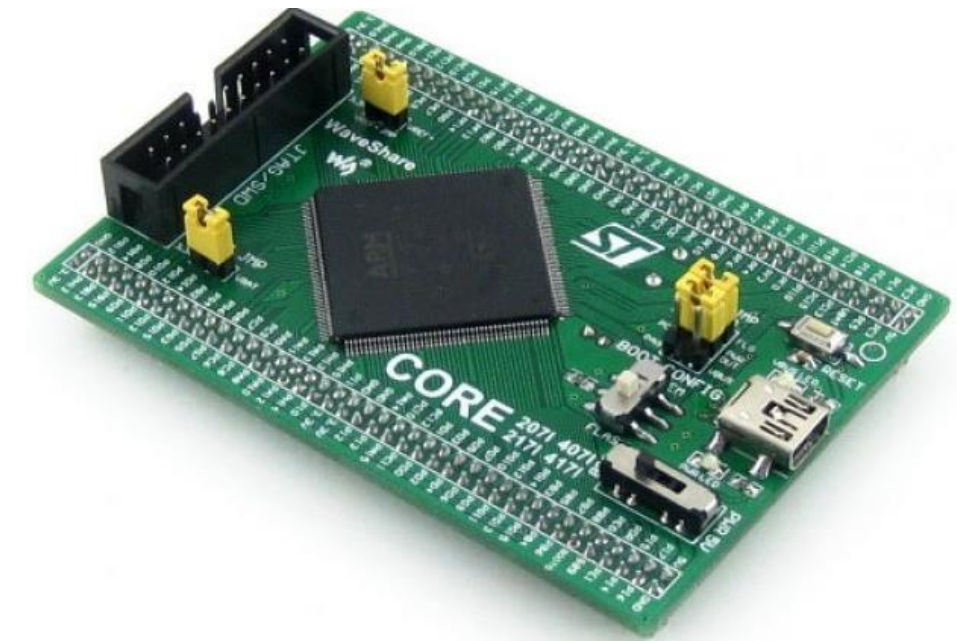
FI focus: cryptographic logic only, no generic bypasses like memcmp() skips or forced returns



Target Implementation – Firmware

STM32F417, Arm Cortex-M4 core

Running **bare metal firmware** (open source on [GitHub](#))

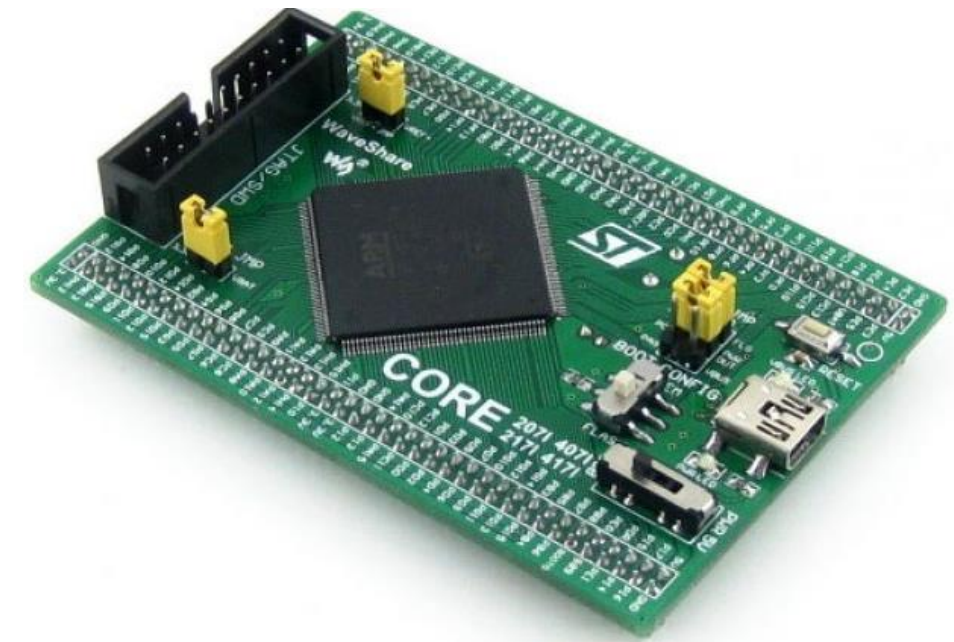


Target Implementation – Firmware

STM32F417, Arm Cortex-M4 core

Running **bare metal firmware** (open source on [GitHub](#))

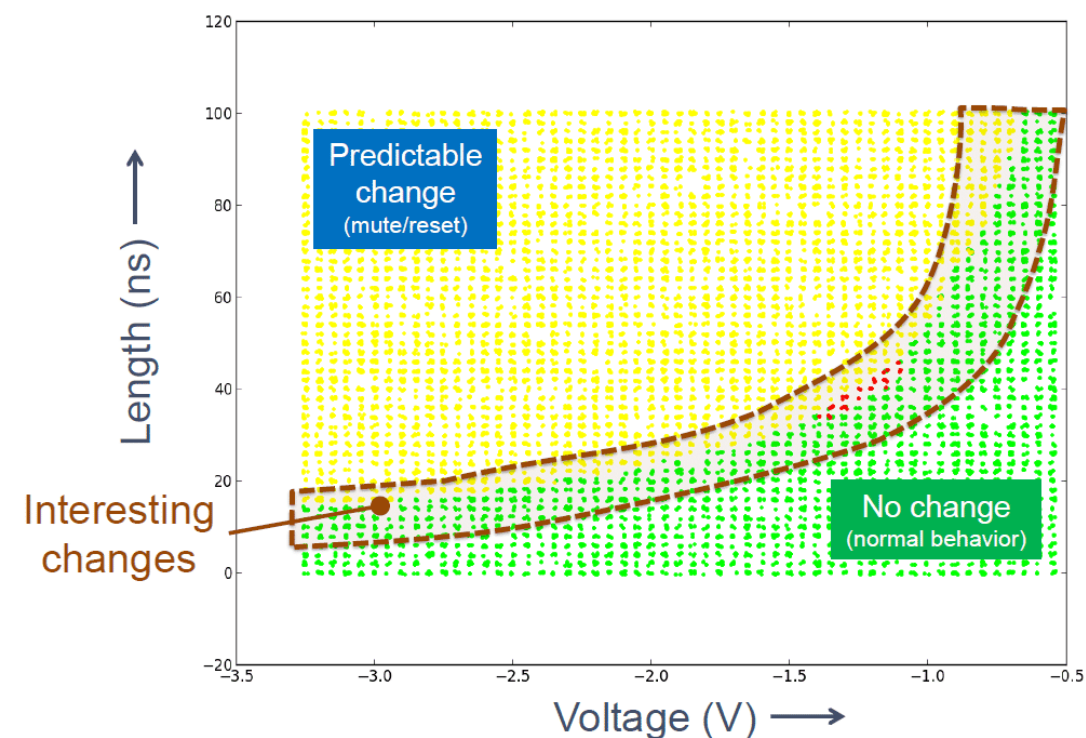
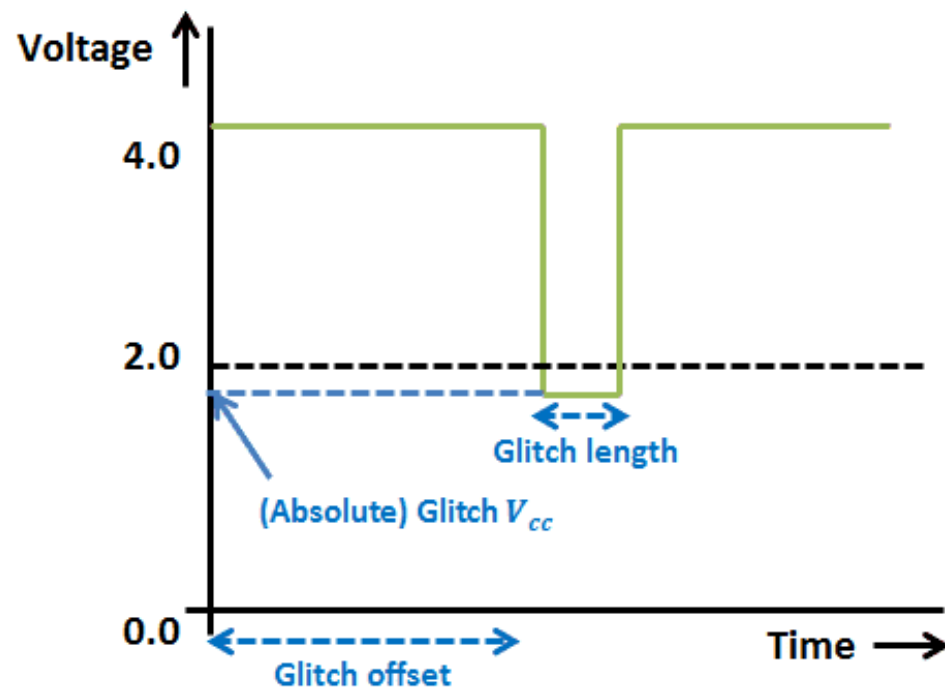
```
case CMD_SW_DILITHIUM_VERIFY: {  
    uint8_t* signedMessageBuffer = DilithiumState_getScratchPad(&dilithium);  
    get_bytes(DILITHIUM_SIGNED_MESSAGE_SIZE, signedMessageBuffer);  
    // Handle the request.  
    BEGIN_INTERESTING_STUFF; // []-> Rising Edge Trigger  
    int result = DilithiumState_verify(&dilithium, signedMessageBuffer);  
    END_INTERESTING_STUFF; // []-> Falling Edge Trigger  
    send_char(result == 0 ? 0 : 1);  
    break;  
}
```



Voltage Fault Injection on Practice

Voltage Fault Injection – Concepts

- Lower the voltage at the right time to trigger faults
- Not 'too soft'; Not 'too hard'



Voltage Fault Injection – Effects on Device

Inject fault(s) to disturb the device, then see what happens:

- Nothing
- Device resets, stops working, or dies

Voltage Fault Injection – Effects on Device

Inject fault(s) to disturb the device, then see what happens:

- Nothing
- Device resets, stops working, or dies
- A **change** in software decision
- A computational **fault**

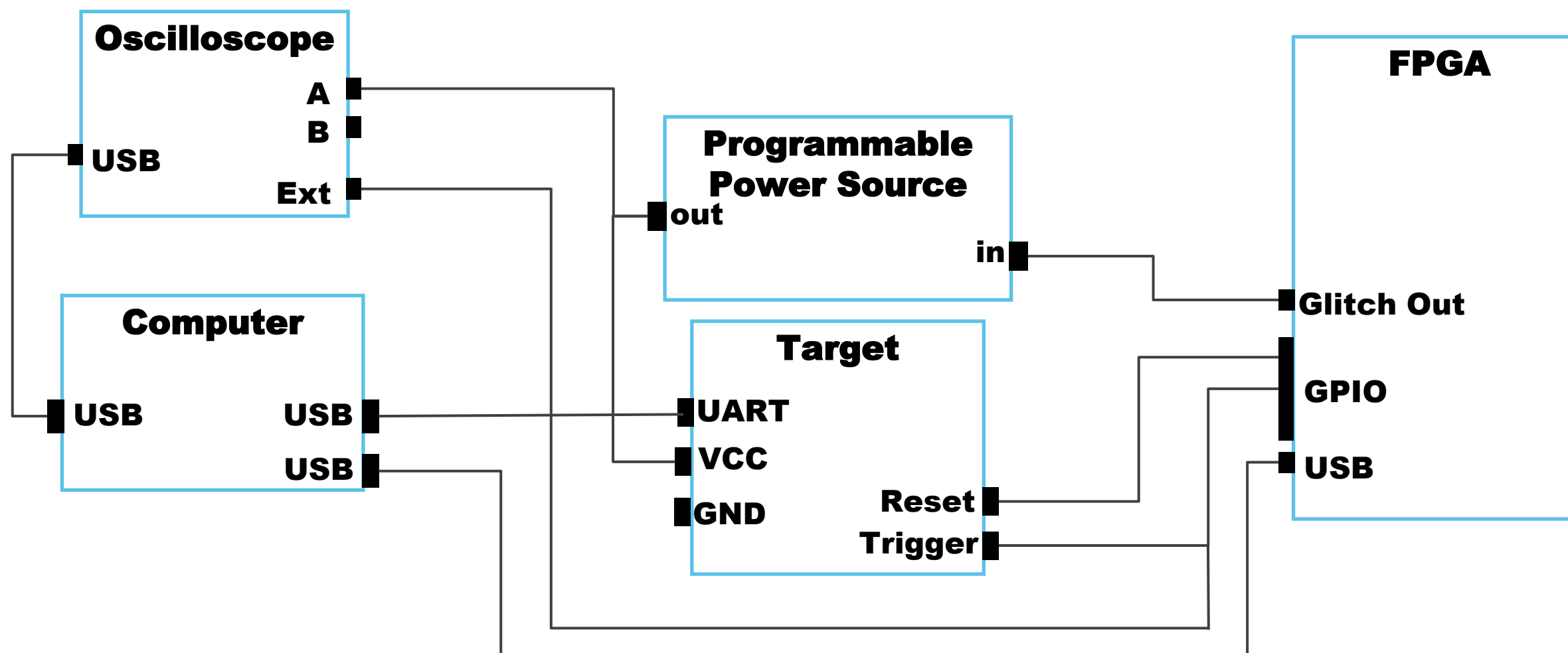
Voltage Fault Injection – Effects on Device

Inject fault(s) to disturb the device, then see what happens:

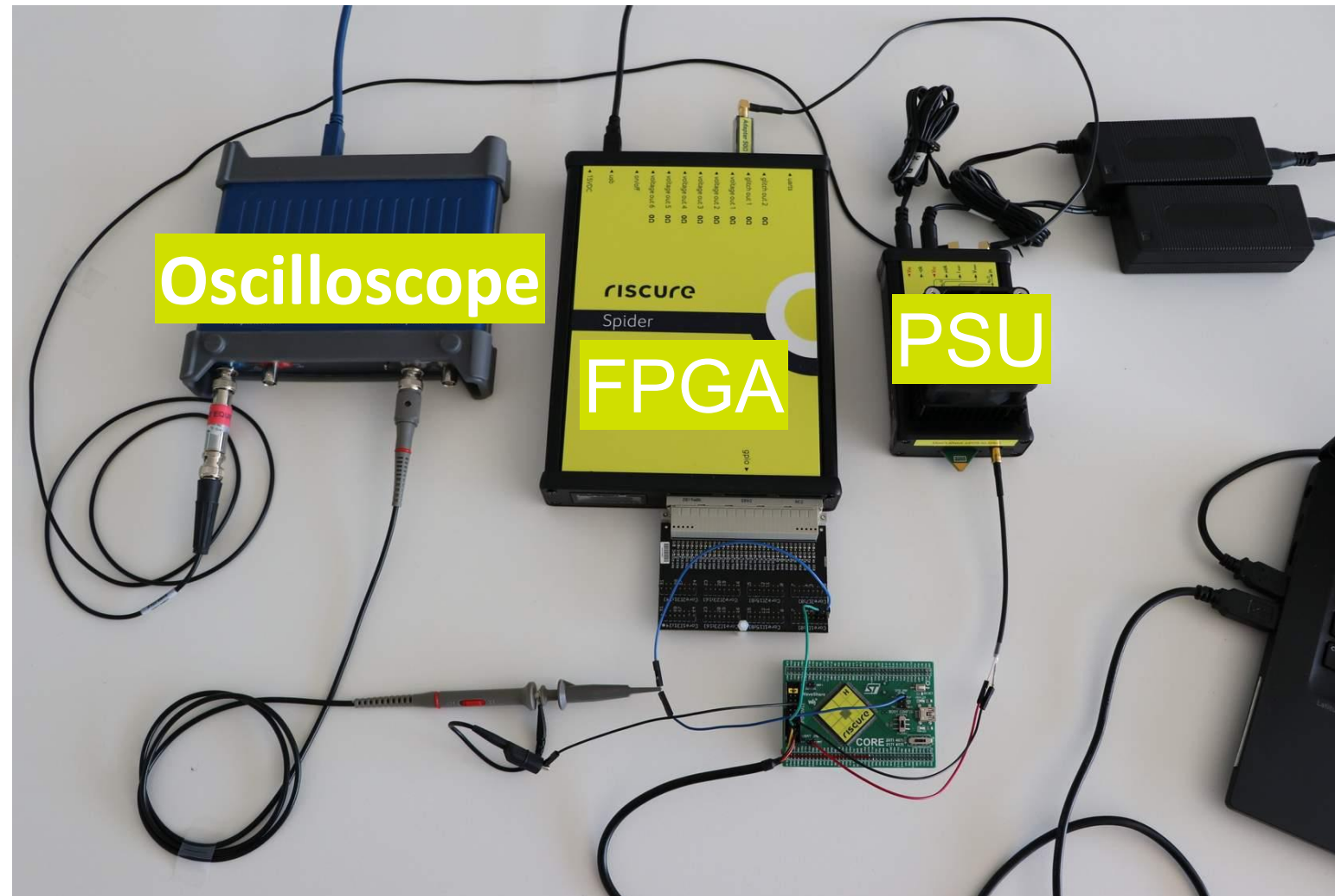
- Nothing
- Device resets, stops working, or dies
- A **change** in software decision
- A computational **fault**

May compromise the device!

Voltage Fault Injection – Setup Overview



Voltage Fault Injection – Real World



Fault Injection Attacks on Dilithium Verification

Introduction to Dilithium

- Lattice-based digital signature scheme, designed to resist quantum attacks.
- Three security levels: Dilithium-2, Dilithium-3, Dilithium-5
- Supports deterministic and randomized signing for flexibility.
- Optimized using Number Theoretic Transform (NTT) for efficiency.

Keygen

1. $\mathbf{A} \leftarrow R_q^{k \times l}$
2. $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
3. $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
4. $(\mathbf{t}_1, \mathbf{t}_0) = \text{Power2Round}(\mathbf{t}, d)$
5. **return** $(pk = (\mathbf{A}, \mathbf{t}_1), sk = (\mathbf{A}, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

Keygen

1. $A \leftarrow R_q^{k \times l}$
2. $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
3. $\mathbf{t} := A\mathbf{s}_1 + \mathbf{s}_2$
4. $(\mathbf{t}_1, \mathbf{t}_0) = \text{Power2Round}(\mathbf{t}, d)$
5. **return** $(pk = (A, \mathbf{t}_1), sk = (A, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

- Expand matrix A from public seed.

Keygen

1. $\mathbf{A} \leftarrow R_q^{k \times l}$
2. $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
3. $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
4. $(\mathbf{t}_1, \mathbf{t}_0) = \text{Power2Round}(\mathbf{t}, d)$
5. **return** $(pk = (\mathbf{A}, \mathbf{t}_1), sk = (\mathbf{A}, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

- Expand matrix \mathbf{A} from public seed.
- Sample secret vectors \mathbf{s}_1 and \mathbf{s}_2

Keygen

1. $A \leftarrow R_q^{k \times l}$
2. $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
3. $\mathbf{t} := A\mathbf{s}_1 + \mathbf{s}_2$
4. $(\mathbf{t}_1, \mathbf{t}_0) = \text{Power2Round}(\mathbf{t}, d)$
5. **return** ($pk = (A, \mathbf{t}_1)$, $sk = (A, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$)

- Expand matrix A from public seed.
- Sample secret vectors \mathbf{s}_1 and \mathbf{s}_2
- Compute \mathbf{t} , then split into \mathbf{t}_1 (**public**) and \mathbf{t}_0 (**secret**).

Keygen

1. $A \leftarrow R_q^{k \times l}$
2. $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^l \times S_\eta^k$
3. $\mathbf{t} := A\mathbf{s}_1 + \mathbf{s}_2$
4. $(\mathbf{t}_1, \mathbf{t}_0) = \text{Power2Round}(\mathbf{t}, d)$
5. **return** $(pk = (A, \mathbf{t}_1), sk = (A, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$

- Expand matrix A from public seed.
- Sample secret vectors \mathbf{s}_1 and \mathbf{s}_2
- Compute \mathbf{t} , then split into \mathbf{t}_1 (**public**) and \mathbf{t}_0 (**secret**).

Sign ($M, sk = (A, \mathbf{s}_1, \mathbf{s}_2, t_0)$)

1. $(\mathbf{z}, h) := \perp$
2. **while** $(\mathbf{z}, h) = \perp$ **do**
3. $\mathbf{y} \leftarrow S_{\gamma_1}^l$
4. $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$
5. $c \in B_\tau := H(M \parallel \mathbf{w}_1)$
6. $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
7. **if** rejection conditions met \rightarrow **then** $\mathbf{z} := \perp$
8. **else**
9. $h = \text{MakeHint}(-ct_0, \mathbf{w} - c\mathbf{s}_2 + ct_0)$
10. **if** $\|ct_0\|_\infty \geq \gamma_2$, **then** $(\mathbf{z}, h) = \perp$
11. **return** $\sigma = (c, \mathbf{z}, h)$

Sign ($M, sk = (A, \mathbf{s}_1, \mathbf{s}_2, t_0)$)

1. $(\mathbf{z}, h) := \perp$
2. **while** $(\mathbf{z}, h) = \perp$ **do**
3. $\mathbf{y} \leftarrow S_{\gamma_1}^l$
4. $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$
5. $c \in B_\tau := H(M \parallel \mathbf{w}_1)$
6. $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
7. **if** rejection conditions met \rightarrow **then** $\mathbf{z} := \perp$
8. **else**
9. $h = \text{MakeHint}(-ct_0, \mathbf{w} - c\mathbf{s}_2 + ct_0)$
10. **if** $\|ct_0\|_\infty \geq \gamma_2$, **then** $(\mathbf{z}, h) = \perp$
11. **return** $\sigma = (c, \mathbf{z}, h)$

- Rejection loop samples \mathbf{y} , computes challenge c , computes \mathbf{z} , and verifies constraints.

Sign ($M, sk = (A, \mathbf{s}_1, \mathbf{s}_2, t_0)$)

1. $(\mathbf{z}, h) := \perp$
2. **while**($\mathbf{z}, h) = \perp$ **do**
3. $\mathbf{y} \leftarrow S_{\gamma_1}^l$
4. $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$
5. $c \in B_\tau := H(M \parallel \mathbf{w}_1)$
6. $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
7. **if** rejection conditions met \rightarrow **then** $\mathbf{z} := \perp$
8. **else**
9. $h = \text{MakeHint}(-ct_0, \mathbf{w} - c\mathbf{s}_2 + ct_0)$
10. **if** $\|ct_0\|_\infty \geq \gamma_2$, **then** $(\mathbf{z}, h) = \perp$
11. **return** $\sigma = (c, \mathbf{z}, h)$

- Rejection loop samples \mathbf{y} , computes challenge c , computes \mathbf{z} , and verifies constraints.
- Verifier lacks t_0 , so hint h helps recover \mathbf{w}_1 .

Sign ($M, sk = (A, \mathbf{s}_1, \mathbf{s}_2, t_0)$)

1. $(\mathbf{z}, h) := \perp$
2. **while** $(\mathbf{z}, h) = \perp$ **do**
3. $\mathbf{y} \leftarrow S_{\gamma_1}^l$
4. $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$
5. $c \in B_\tau := H(M \parallel \mathbf{w}_1)$
6. $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
7. **if** rejection conditions met \rightarrow **then** $\mathbf{z} := \perp$
8. **else**
9. $h = \text{MakeHint}(-ct_0, \mathbf{w} - c\mathbf{s}_2 + ct_0)$
10. **if** $\|ct_0\|_\infty \geq \gamma_2$, **then** $(\mathbf{z}, h) = \perp$
11. **return** $\sigma = (c, \mathbf{z}, h)$

- Rejection loop samples \mathbf{y} , computes challenge c , computes \mathbf{z} , and verifies constraints.
- Verifier lacks t_0 , so hint h helps recover \mathbf{w}_1 .

Verify ($pk = (A, t_1), M, \sigma = (c, \mathbf{z}, h)$)

1. $\mathbf{w}'_1 := \text{UseHint}(h, A\mathbf{z} - ct_1 2^d)$
2. **if** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ **and** $c = \mathbf{H}(M \parallel \mathbf{w}'_1)$ **and** $|h|_{h_j=1} \leq \omega$
3. **return True**
4. **else**
5. **return False**

Verify ($pk = (A, t_1), M, \sigma = (c, \mathbf{z}, h)$)

1. $\mathbf{w}'_1 := \text{UseHint}(h, \mathbf{Az} - ct_1 2^d)$
2. **if** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ **and** $c = \mathbf{H}(M \parallel \mathbf{w}'_1)$ **and** $|h|_{h_j=1} \leq \omega$
3. **return True**
4. **else**
5. **return False**

- Computes high bits of $\mathbf{Az} - ct_1 2^d$

Verify ($pk = (A, t_1), M, \sigma = (c, \mathbf{z}, h)$)

1. $\mathbf{w}'_1 := \text{UseHint}(h, A\mathbf{z} - ct_1 2^d)$
2. if $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $c = H(M \parallel \mathbf{w}'_1)$ and $|h|_{h_j=1} \leq \omega$
3. **return True**
4. **else**
5. **return False**

- Computes high bits of $A\mathbf{z} - ct_1 2^d$
- Correct with hint vector h .

Verify ($pk = (A, t_1), M, \sigma = (c, \mathbf{z}, h)$)

1. $\mathbf{w}'_1 := \text{UseHint}(h, A\mathbf{z} - ct_1 2^d)$
 2. if $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $c = \text{H}(M \parallel \mathbf{w}'_1)$ and $|h|_{h_j=1} \leq \omega$
 3. **return True**
 4. **else**
 5. **return False**
- Computes high bits of $A\mathbf{z} - ct_1 2^d$
 - Correct with hint vector h .
 - Recomputes \mathbf{w}'_1 and challenge c from message.

Verify ($pk = (A, t_1), M, \sigma = (c, \mathbf{z}, h)$)

1. $\mathbf{w}'_1 := \text{UseHint}(h, A\mathbf{z} - ct_1 2^d)$
 2. **if** $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ **and** $c = \mathbf{H}(M \parallel \mathbf{w}'_1)$ **and** $|h|_{h_j=1} \leq \omega$
 3. **return True**
 4. **else**
 5. **return False**
- Computes high bits of $A\mathbf{z} - ct_1 2^d$
 - Correct with hint vector h .
 - Recomputes \mathbf{w}'_1 and challenge c from message.

Fault Attack on Dilithium Verification

- This research builds on [*Fault Attacks Sensitivity of Public Parameters in the Dilithium Verification*](#) (CARDIS 2023).

Fault Attack on Dilithium Verification

- This research builds on [*Fault Attacks Sensitivity of Public Parameters in the Dilithium Verification*](#) (CARDIS 2023).

Verification Line 1: $w'_1 := \text{UseHint}(h, Az - ct_1 2^d)$

Fault Attack on Dilithium Verification

- This research builds on [*Fault Attacks Sensitivity of Public Parameters in the Dilithium Verification*](#) (CARDIS 2023).

Verification Line 1: $w'_1 := \text{UseHint}(h, Az - ct_1 2^d)$

- The paper shows how to recover w_1 using only public inputs.

Fault Attack on Dilithium Verification

- This research builds on [*Fault Attacks Sensitivity of Public Parameters in the Dilithium Verification*](#) (CARDIS 2023).

Verification Line 1: $w'_1 := \text{UseHint}(h, Az - ct_1 2^d)$

- The paper shows how to recover w_1 using only public inputs.
- It demonstrates how to constrain $ct_1 2^d$ to minimally affect Az 's high bits.

Fault Attack on Dilithium Verification

- This research builds on [*Fault Attacks Sensitivity of Public Parameters in the Dilithium Verification*](#) (CARDIS 2023).

$$\text{Verification Line 1: } w'_1 := \text{UseHint}(h, Az - ct_1 2^d)$$

- The paper shows how to recover w_1 using only public inputs.
- It demonstrates how to constrain $ct_1 2^d$ to minimally affect Az 's high bits.
- Using this, it presents two signature verification attacks: **exploiting verification with FI.**

Attacks

If either of the following conditions is met during a successful fault injection at verification:

Attack 1.

Attack 2.

- *A signature generated using only public values **will be accepted as valid.***

Attacks

If either of the following conditions is met during a successful fault injection at verification:

Attack 1. $ct_1 2^d = 0$

Attack 2.

- *A signature generated using only public values **will be accepted as valid.***

Attacks

If either of the following conditions is met during a successful fault injection at verification:

Attack 1. $ct_1 2^d = 0$

Attack 2. $\|ct_1 2^d\|_\infty \leq \gamma_2$ and $h = \text{MakeHint}(ct_1 2^d, Az - ct_1 2^d)$

Then, $\text{HighBits}(Az - ct_1 2^d, 2\gamma_2) = \text{HighBits}(w, 2\gamma_2)$

- *A signature generated using only public values **will be accepted as valid.***

Our Signature Generation for Attack 1 ($M, pk = (A, t_1)$)

1. $(z) := \perp$
 2. **while**($z = \perp$) **do**
 3. $z \leftarrow S_{\gamma_1 - \beta}^l$
 4. $h = 0$
 5. $w_1 := \text{UseHint}(h, Az, 2\gamma_2)$
 6. $c \in B_\tau := H(M \parallel w_1)$
 7. **return** $\sigma = (c, z, h)$
- The generated signature will accept by verification if fault forces $ct_1 2^d = 0$ at **signature verification**.

Our Signature Generation for Attack 2 ($M, pk = (A, t_1)$)

1. $(z, h) := \perp$
2. **while** $(z, h) = \perp$ **do**
3. $z \leftarrow S_{\gamma_1 - \beta}^l$
4. $h = 0$
5. $w_1 := \text{HighBits}(h, Az, 2\gamma_2)$
6. $c \in B_\tau := H(M \parallel w_1)$
7. $h = \text{MakeHint}(-ct_1 2^{d'}, Az - ct_1 2^{d'})$
8. **if** $|h|_{h_j=1} > \omega$, **then** $(z, h) = \perp$
9. **return** $\sigma = (c, z, h)$

- The generated signature will accept by verification if fault forces $\|ct_1 2^d\|_\infty \leq \gamma_2$ at signature verification.

Where to target?

$$\mathbf{w}'_1 := \text{UseHint}(h, \mathbf{Az} - c \mathbf{t}_1 2^d)$$

Where to target?

$$w'_1 := \text{UseHint}(h, Az - c \mathbf{t}_1 2^d)$$

Where to target?

Scenario 1: Unpacking of Public Key*

- Attack 1 ($ct_1 2^d = 0$)

$$w'_1 := \text{UseHint}(h, Az - c \mathbf{t}_1 2^d)$$
A brown line originates from a circle drawn around the \mathbf{t}_1 term in the equation. The line extends upwards and then turns left as an arrow pointing towards the text "Attack 1" in the list above.

*Depends on implementation and compiler behavior. Requires zero-initialized \mathbf{t}_1 coefficients.

Where to target?

Scenario 1: Unpacking of Public Key*

- Attack 1 ($ct_1 2^d = 0$)

Scenario 2: Sampling of c

- Attack 1 ($ct_1 2^d = 0$)

$$w'_1 := \text{UseHint}(h, Az - \textcircled{c} t_1 2^d)$$
A thin black line originates from the bottom of the circled 'c' in the equation, extends horizontally to the right, and then turns vertically upwards, ending with an arrowhead pointing towards the 'Attack 1' text in the 'Scenario 2' section.

*Depends on implementation and compiler behavior. Requires zero-initialized t_1 coefficients.

Where to target?

Scenario 1: Unpacking of Public Key*

- **Attack 1** ($ct_1 2^d = 0$)

Scenario 2: Sampling of c

- **Attack 1** ($ct_1 2^d = 0$)

$$w'_1 := \text{UseHint}(h, Az - c t_1 2^d)$$

Scenario 3: Shift by d

- **Attack 2** ($ct_1 2^d = \|ct_1 2^d\|_\infty \leq \gamma_2$)

*Depends on implementation and compiler behavior. Requires zero-initialized t_1 coefficients.

Where to target?

Scenario 1: Unpacking of Public Key*

- **Attack 1** ($ct_1 2^d = 0$)

Scenario 2: Sampling of c

- **Attack 1** ($ct_1 2^d = 0$)

$$w'_1 := \text{UseHint}(h, Az \ominus c t_1 2^d)$$


Scenario 3: Shift by d

- **Attack 2** ($ct_1 2^d = \|ct_1 2^d\|_\infty \leq \gamma_2$)

Scenario 4: Subtraction

- **Attack 1** ($ct_1 2^d = 0$)

*Depends on implementation and compiler behavior. Requires zero-initialized t_1 coefficients.

Where to target?

Scenario 1: Unpacking of Public Key*

- **Attack 1** ($ct_1 2^d = 0$)

Scenario 2: Sampling of c

- **Attack 1** ($ct_1 2^d = 0$)

$$w'_1 := \text{UseHint}(h, Az - ct_1 2^d)$$

Scenario 3: Shift by d

- **Attack 2** ($ct_1 2^d = \|ct_1 2^d\|_\infty \leq \gamma_2$)

Scenario 4: Subtraction

- **Attack 1** ($ct_1 2^d = 0$)

*Depends on implementation and compiler behavior. Requires zero-initialized t_1 coefficients.

How to Target?



Chose the fault
injection point

How to Target?



Chose the fault injection point

1. $(z) := \perp$
2. **while** $(z) = \perp$ **do**
3. $z \leftarrow S_{Y_1-\beta}^l$
4. $h = 0$
5. $w_1 \leftarrow \text{UseHint}(h, Az, 2Y_2)$
6. $c \in B_r \leftarrow H(M \parallel w_1)$
7. **return** $\sigma = (c, z, h)$

1. $(z, h) := \perp$
2. **while** $(z, h) = \perp$ **do**
3. $z \leftarrow S_{Y_1-\beta}^l$
4. $h = 0$
5. $w_1 \leftarrow \text{HighBits}(h, Az, 2Y_2)$
6. $c \in B_r \leftarrow H(M \parallel w_1)$
7. $h \leftarrow \text{MakeHint}(-ct_1 2^{d'}, Az - ct_1 2^{d'})$
8. **if** $|h|_{h_{j-1}} > \omega$, **then** $(z, h) = \perp$
9. **return** $\sigma = (c, z, h)$

Generate a signature using public key and an **arbitrary message**

How to Target?



```

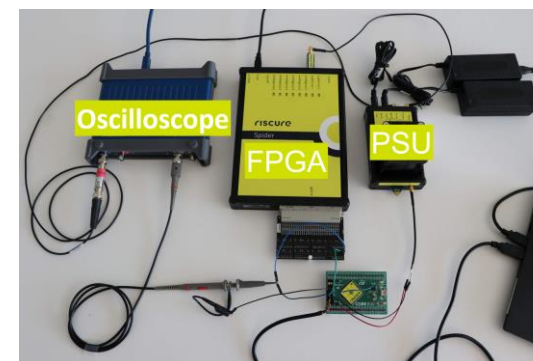
1.  $(z) := \perp$ 
2. while  $(z) = \perp$  do
3.    $z \leftarrow S_{Y_1-\beta}^l$ 
4.    $h = 0$ 
5.    $w_1 := \text{UseHint}(h, Az, 2Y_2)$ 
6.    $c \in B_r := H(M \parallel w_1)$ 
7. return  $\sigma = (c, z, h)$ 

```

```

1.  $(z, h) := \perp$ 
2. while  $(z, h) = \perp$  do
3.    $z \leftarrow S_{Y_1-\beta}^l$ 
4.    $h = 0$ 
5.    $w_1 := \text{HighBits}(h, Az, 2Y_2)$ 
6.    $c \in B_r := H(M \parallel w_1)$ 
7.    $h = \text{MakeHint}(-ct_1 z^{d'}, Az - ct_1 z^{d'})$ 
8.   if  $|h|_{h_{j-1}} > \omega$ , then  $(z, h) = \perp$ 
9. return  $\sigma = (c, z, h)$ 

```



Chose the fault injection point

Generate a signature using public key and an **arbitrary message**

Inject the fault at the target **during verification**

Fault Injection Results

Scenario 2: Sampling of c

```
1. void poly_challenge(poly *c, const
   uint8_t seed[SEEDBYTES]) {
2.     unsigned int i, b, pos;
3.     ...
4.     for(i = 0; i < N; ++i)
5.         c->coeffs[i] = 0;
```

```
6.     for(i = N-TAU; i < N; ++i) {
7.         ...
8.         c->coeffs[i] = c->coeffs[b];
9.         c->coeffs[b] = 1 - 2*(signs & 1);
10.        ...
11.    }
```

PQM4: Dilithium verification source code

Scenario 2: Sampling of c

```
1. void poly_challenge(poly *c, const
   uint8_t seed[SEEDBYTES]) {
2.     unsigned int i, b, pos;
3.     ...
4.     for(i = 0; i < N; ++i)
5.         c->coeffs[i] = 0;
```

```
6.     for(i = N-TAU; i < N; ++i) {
7.         ...
8.         c->coeffs[i] = c->coeffs[b];
9.         c->coeffs[b] = 1 - 2*(signs & 1);
10.        ...
11.    }
```

PQM4: Dilithium verification source code

Scenario 2: Sampling of c


```
1. void poly_challenge(poly *c, const
   uint8_t seed[SEEDBYTES]) {
2.     unsigned int i, b, pos;
3.     ...
4.     for(i = 0; i < N; ++i)
5.         c->coeffs[i] = 0;
```

```
6. for(i = N-TAU; i < N; ++i) {
7.     ...
8.     c->coeffs[i] = c->coeffs[b];
9.     c->coeffs[b] = 1 - 2*(signs & 1);
10.    ...
11. }
```

PQM4: Dilithium verification source code

Scenario 2: Sampling of c

```
1. void poly_challenge(poly *c, const
   uint8_t seed[SEEDBYTES]) {
2.     unsigned int i, b, pos;
3.     ...
4.     for(i = 0; i < N; ++i)
5.         c->coeffs[i] = 0;
```



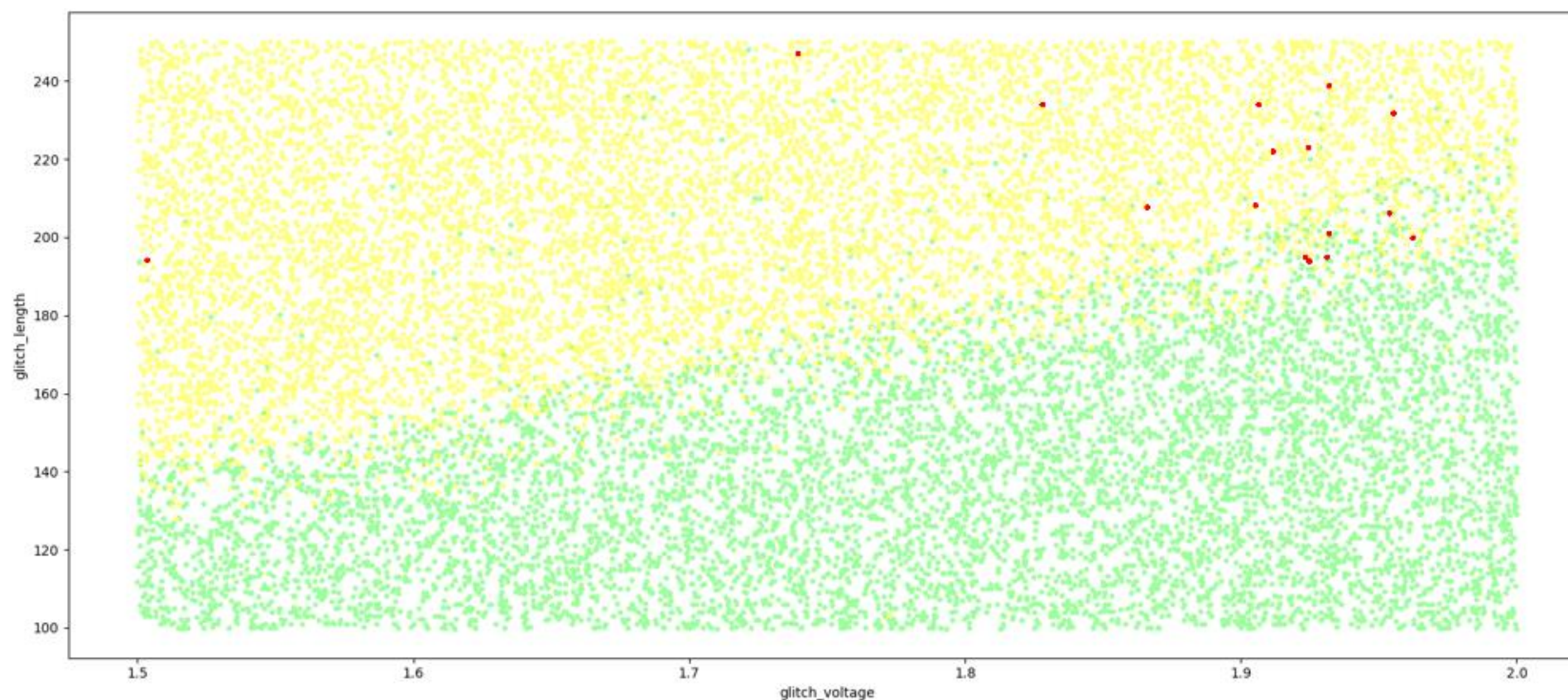
```
6. for(i = N-TAU; i < N; ++i) {
7.     ...
8.     c->coeffs[i] = c->coeffs[b];
9.     c->coeffs[b] = 1 - 2*(signs & 1);
10.    ...
11. }
```

PQM4: Dilithium verification source code

- If the fault is injected in second for loop of the **poly_challenge** function, the **for loop will be skipped**.
- As a result, each coefficient of c will be equal to **zero**, enabling **Attack 1** ($ct_1 2^d = 0$)

Scenario 2: Sampling of C

- Green: Normal Execution
- Yellow: Mute/Reset
- Red: Success



Fault Injection Plot of Sampling of C Scenario

Scenario 3: Shift by d

```
void polyveck_shifftl(polyveck *v) {  
    unsigned int i;  
  
    for(i = 0; i < K; ++i)  
        poly_shifftl(&v->vec[i]);  
}
```

```
void poly_shifftl(poly *a) {  
    ...  
    for(i = 0; i < N; ++i)  
        a->coeffs[i] <<= D;  
}
```

PQM4: Dilithium verification source code

Scenario 3: Shift by d

```
void polyveck_shifftl(polyveck *v) {  
    unsigned int i;
```

```
    for(i = 0; i < K; ++i)  
        poly_shifftl(&v->vec[i]);
```

```
void poly_shifftl(poly *a) {  
    ...  
    for(i = 0; i < N; ++i)  
        a->coeffs[i] <<= D;
```

PQM4: Dilithium verification source code

Scenario 3: Shift by d

```
void polyveck_shiftl(polyveck *v) {  
    unsigned int i;  
    for(i = 0; i < K; ++i)  
        poly_shiftl(&v->vec[i]);  
}
```



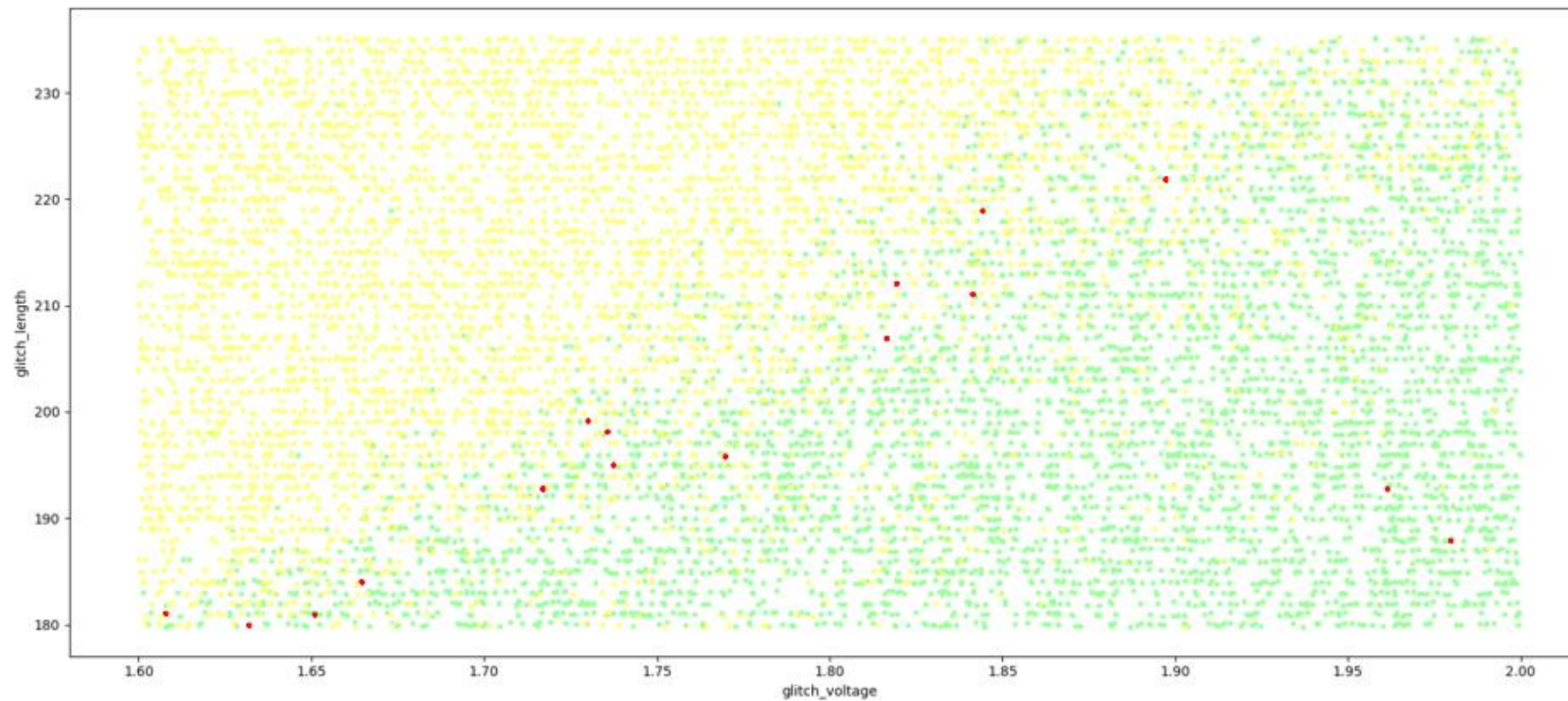
```
void poly_shiftl(poly *a) {  
    ...  
    for(i = 0; i < N; ++i)  
        a->coeffs[i] <<= D;  
}
```

PQM4: Dilithium verification source code

- If the fault is injected at **for loop** of the **polyveck_shiftl** function, the entire **for** loop will be skipped.
- As a result, **each coefficient of t_1 will remain unshifted.**
- $\|ct_1 2^d\|_\infty \leq \gamma_2$, enabling **Attack 2** with $d' = 0$

Scenario 3: Shift by d

- Green: Normal Execution
- Yellow: Mute/Reset
- Red: Success



Fault Injection Plot of Shift by d Scenario

Scenario 4: Subtraction

```
void polyveck_sub(polyveck *w, const polyveck *u, const
polyveck *v) {
    unsigned int i;
    for(i = 0; i < K; ++i) {
        send_char(dilithium_counter);
        poly_sub(&w->vec[i], &u->vec[i], &v->vec[i]); }
}
```

PQM4: Dilithium verification source code

Scenario 4: Subtraction

```
void polyveck_sub(polyveck *w, const polyveck *u, const
polyveck *v) {
    unsigned int i;
    for(i = 0; i < K; ++i) {
        send_char(dilithium_counter);
        poly_sub(&w->vec[i], &u->vec[i], &v->vec[i]); }
}
```

PQM4: Dilithium verification source code

Scenario 4: Subtraction

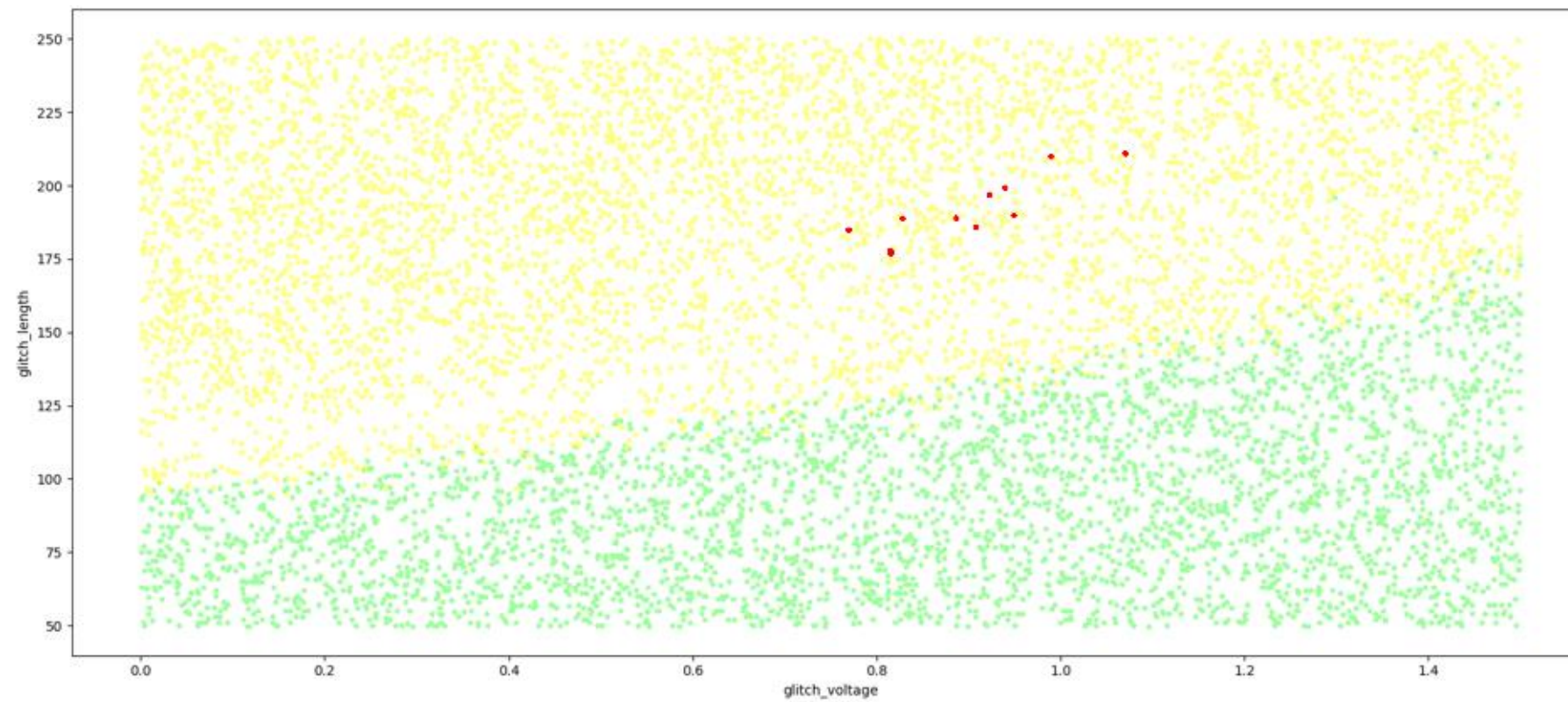
```
void polyveck_sub(polyveck *w, const polyveck *u, const
polyveck *v) {
    unsigned int i;
    for(i = 0; i < K; ++i) {
        send_char(dilithium_counter);
        poly_sub(&w->vec[i], &u->vec[i], &v->vec[i]); }
}
```

PQM4: Dilithium verification source code

- If a fault is injected at **for loop** of the **polyveck_sub** function, the entire **for** loop will be skipped
- As a result, the subtraction is not performed, so: $\mathbf{Az} - c \mathbf{t}_1 2^d = \mathbf{Az}$ (Attack 1)

Scenario 4: Subtraction

- Green: Normal Execution
- Yellow: Mute/Reset
- Red: Success



Fault Injection Plot of Subtraction Scenario

Summary of FI on Dilithium

- Fault Injection on **Dilithium**, can target verification operations like **initialization**, **challenge generation**, **shifting**, or **subtraction**.

Summary of FI on Dilithium

- Fault Injection on **Dilithium**, can target verification operations like **initialization, challenge generation, shifting, or subtraction.**
- **These faults allow bypassing the scheme's logic to accept attacker-generated signatures.**

Summary of FI on Dilithium

- Fault Injection on **Dilithium**, can target verification operations like **initialization, challenge generation, shifting, or subtraction.**
- **These faults allow bypassing the scheme's logic to accept attacker-generated signatures.**
- **The attack surface and behavior vary significantly across implementations, making fault resistance hard to generalize.**

Bypassing WOTS+ Based Hash Based Signature Verification via Fault Injection

Introduction to Hash Based Signatures

- Hash-based cryptography builds on the cryptographic hash functions.
- Since quantum computers struggle to break secure hash functions, these schemes remain resistant .
- This talk focuses on their fundamental building block: **Winternitz One-Time Signature (WOTS+)**.

Introduction to WOTS

- Winternitz Parameter:

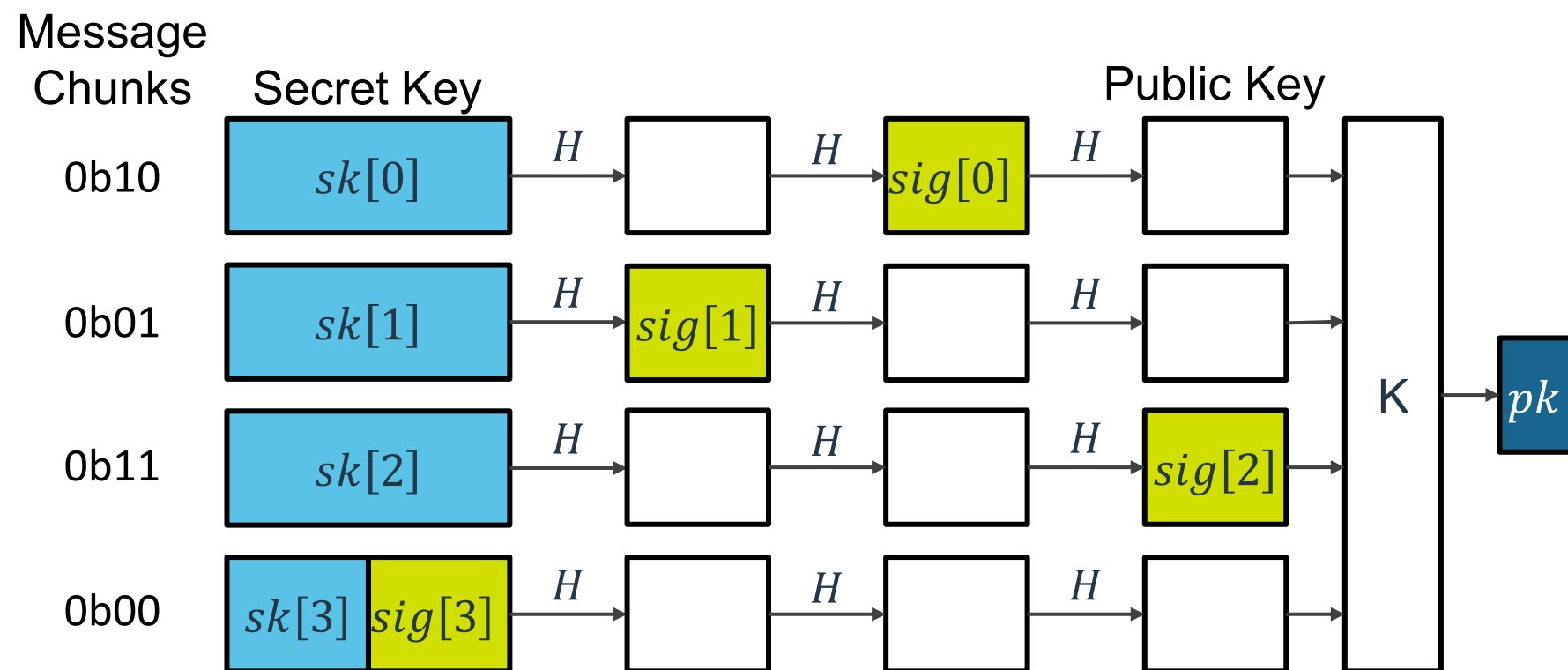
$$w = 4$$

- Hash Chain Length:

$$w - 1 = 3$$

- Chunk Bit Size:

$$\log_2(w) = 2$$



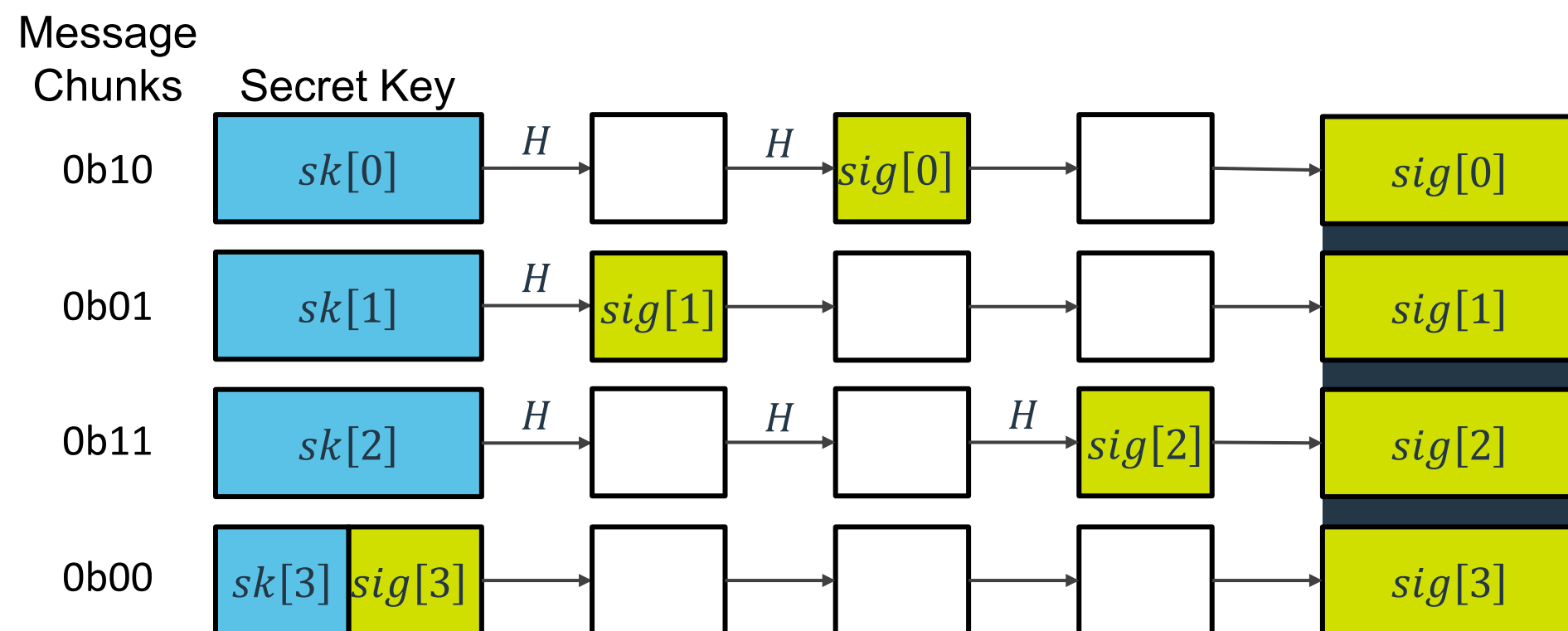
Signature Generation: Split message hash into chunks

- $m = [m_0, m_1, \dots, m_{l_1-1}]$
(Each m_i is in $[0, w - 1]$)



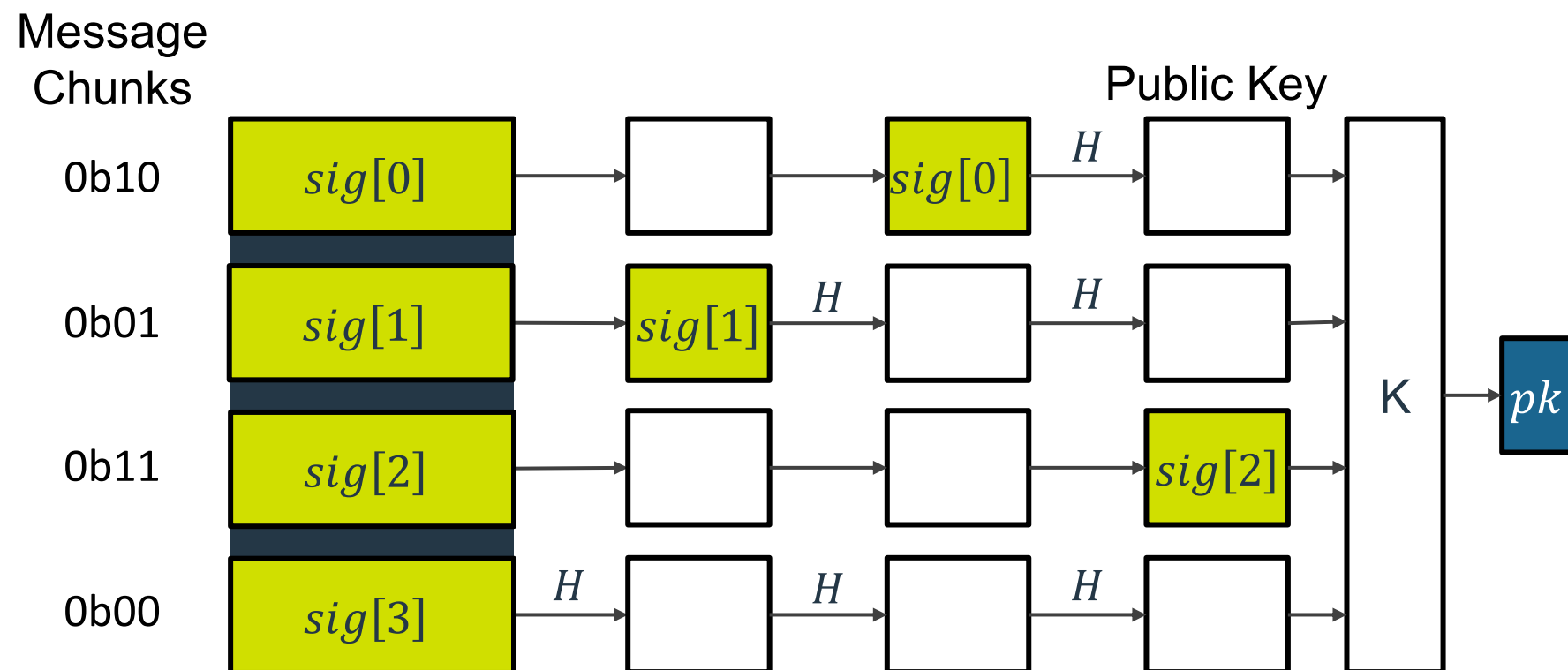
Signature Generation: Signing

- $sig[i] = hash_chain(sk[i], steps = m[i])$



Signature Verification:

- $pk[i] == hash_chain(sig[i], steps = w - 1 - m[i])$

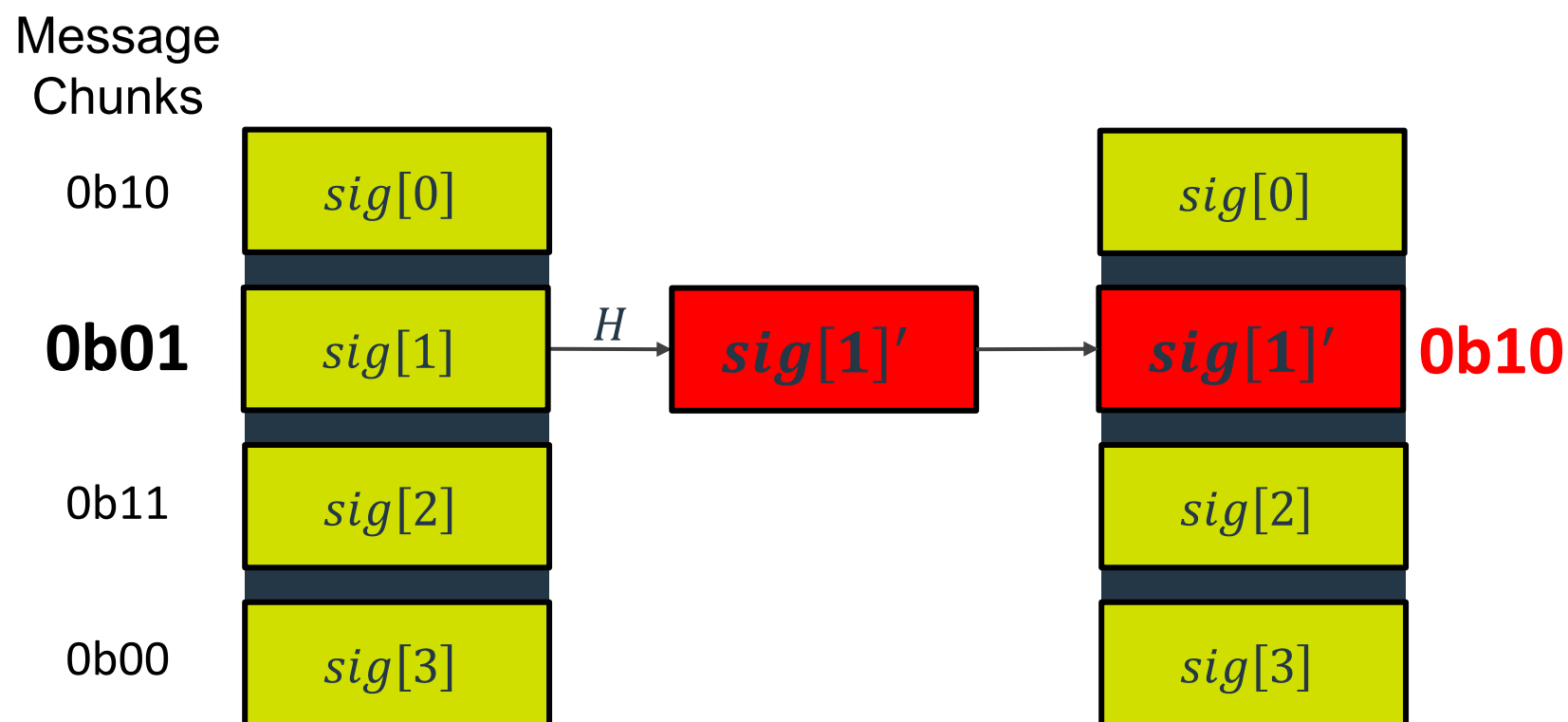


Brute-Force Forgery on WOTS

- If the attacker brute forces $m' \geq m$, they can forge a valid signature.

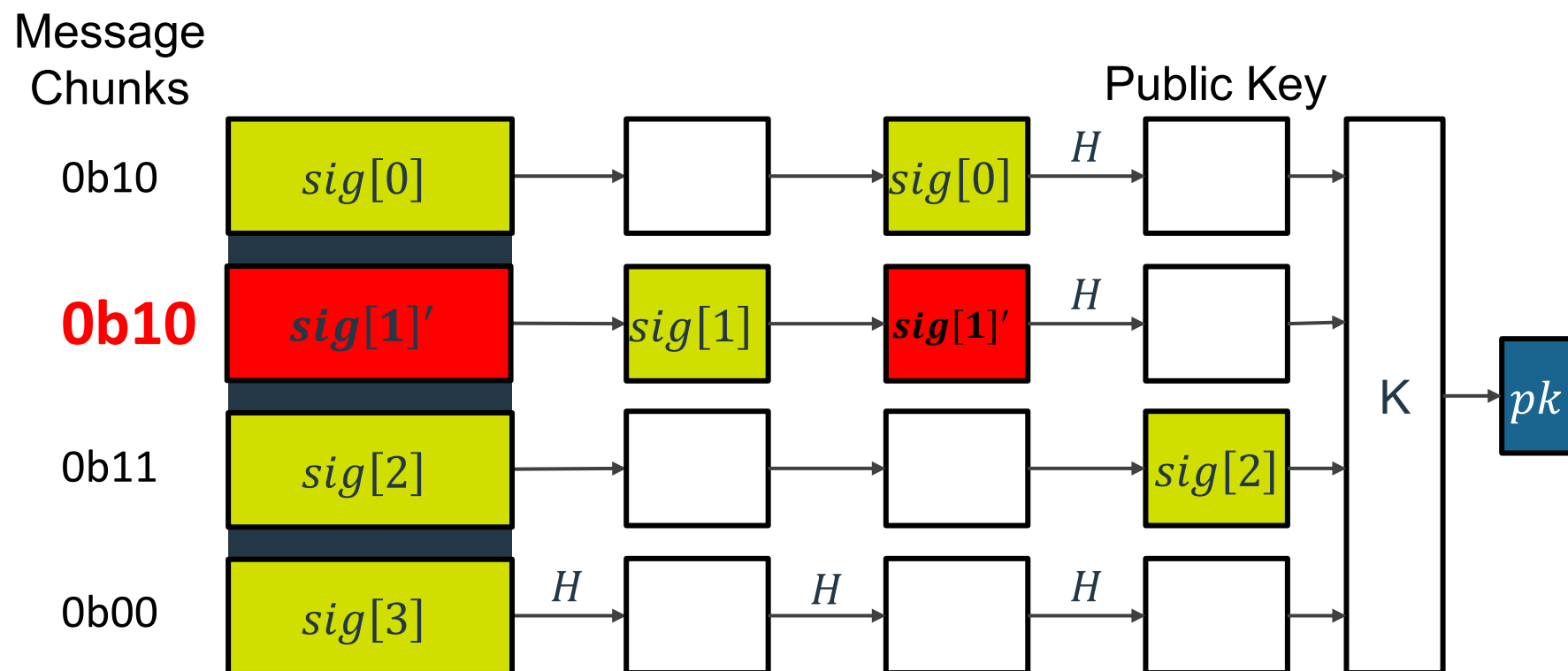
Brute-Force Forgery on WOTS

- If the attacker brute forces $m' \geq m$, they can forge a valid signature.



Brute-Force Forgery on WOTS

- If the attacker brute forces $m' \geq m$, they can forge a valid signature.



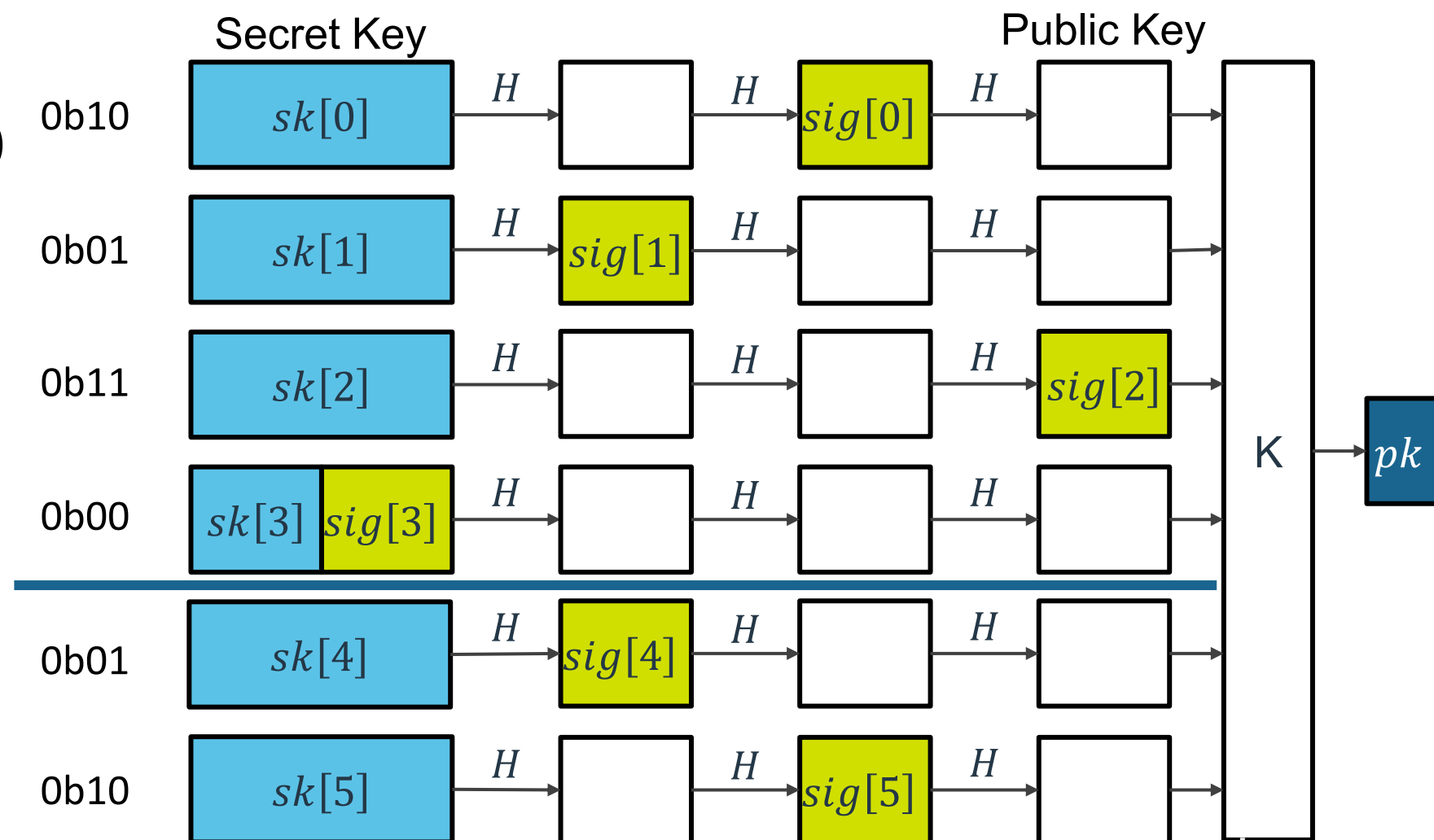
How WOTS+ Adds a Checksum

Checksum:

$$c = C(m) = \sum_{i=0}^{l_1} (w - 1 - m_i)$$

$$c = 1 + 2 + 0 + 3$$

$$c = 6 = 0b0110$$



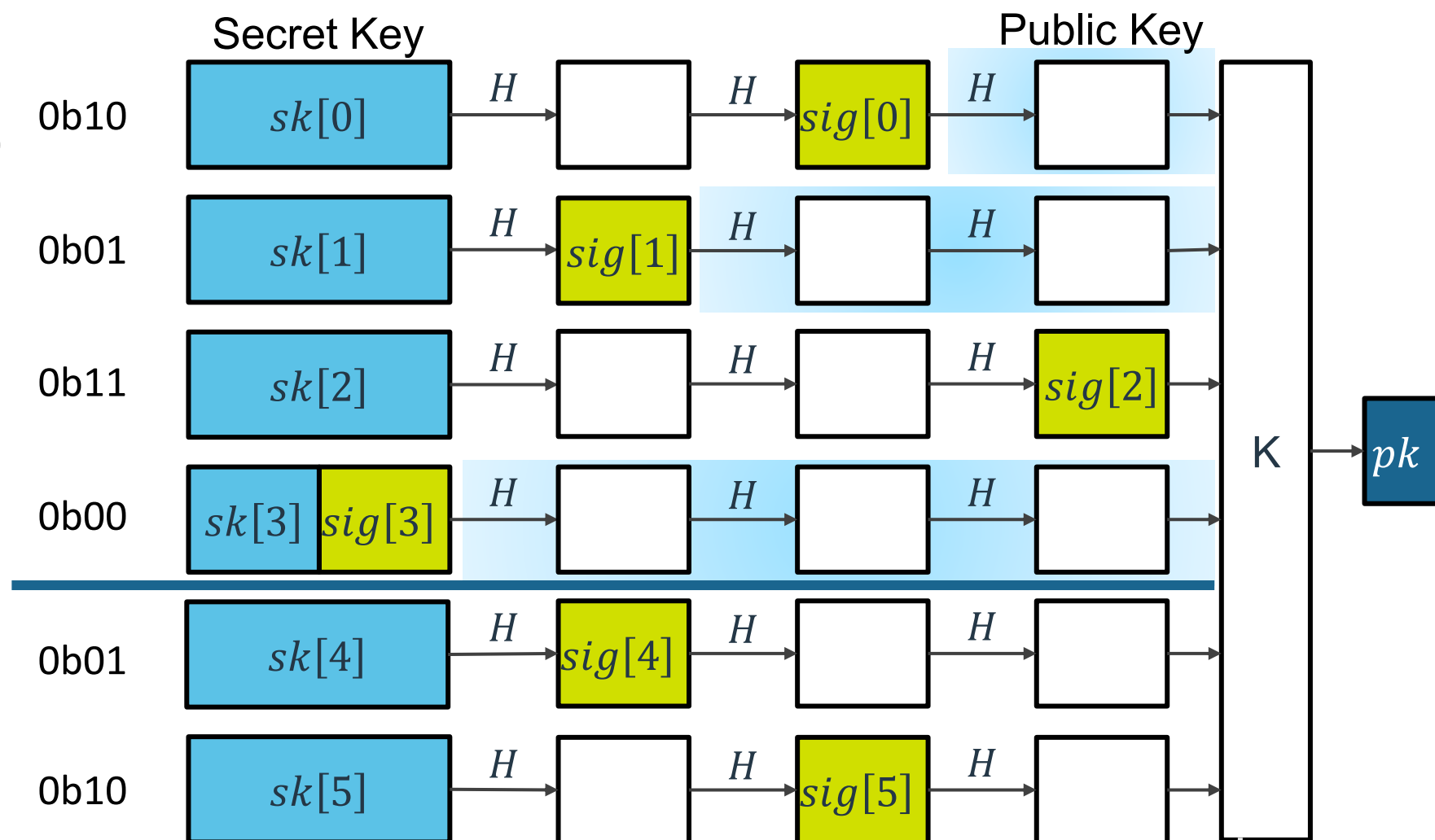
How WOTS+ Adds a Checksum

Checksum:

$$c = C(m) = \sum_{i=0}^{l_1} (w - 1 - m_i)$$

$$c = 1 + 2 + 0 + 3$$

$$c = 6 = 0b0110$$



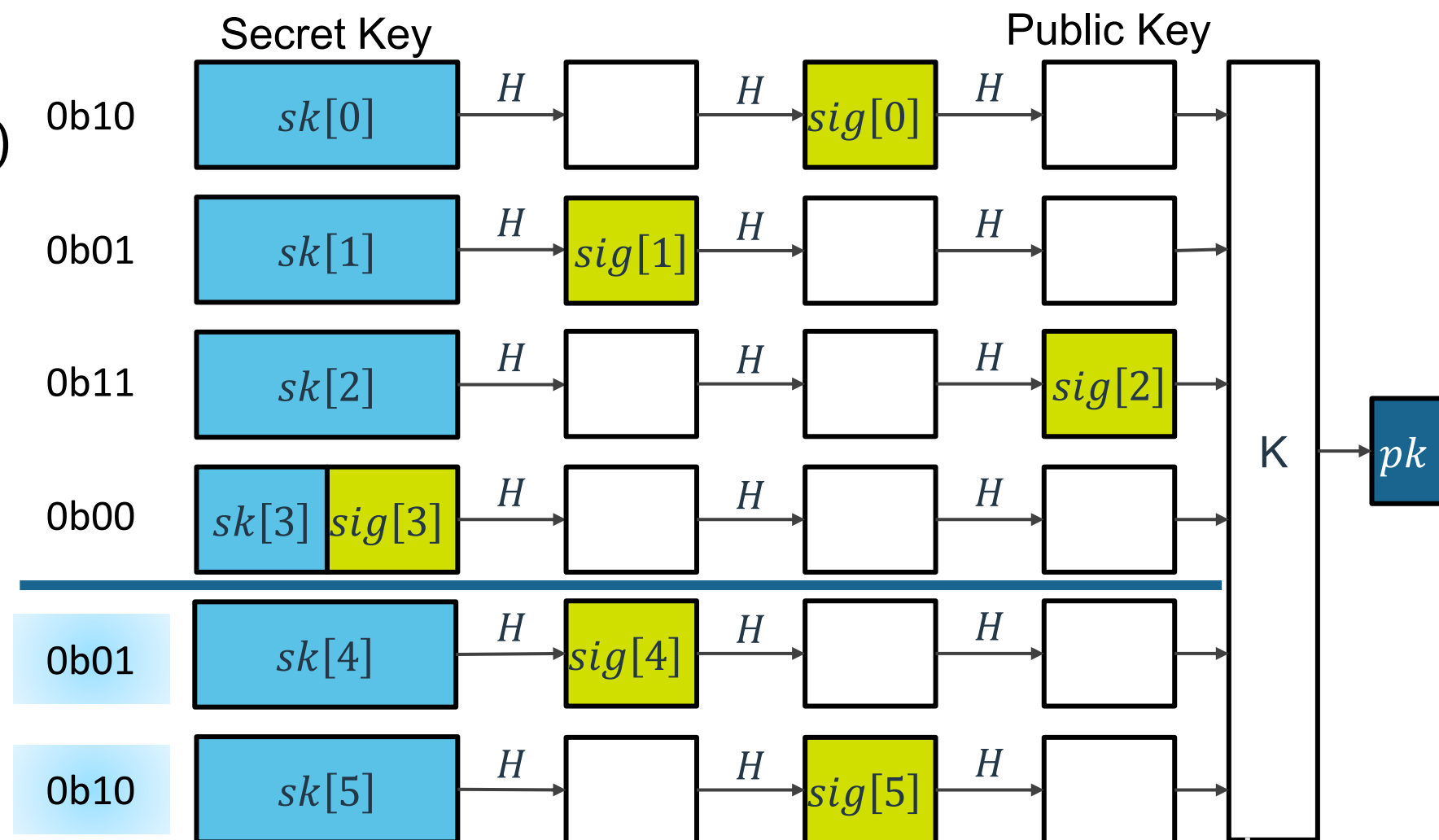
How WOTS+ Adds a Checksum

Checksum:

$$c = C(m) = \sum_{i=0}^{l_1} (w - 1 - m_i)$$

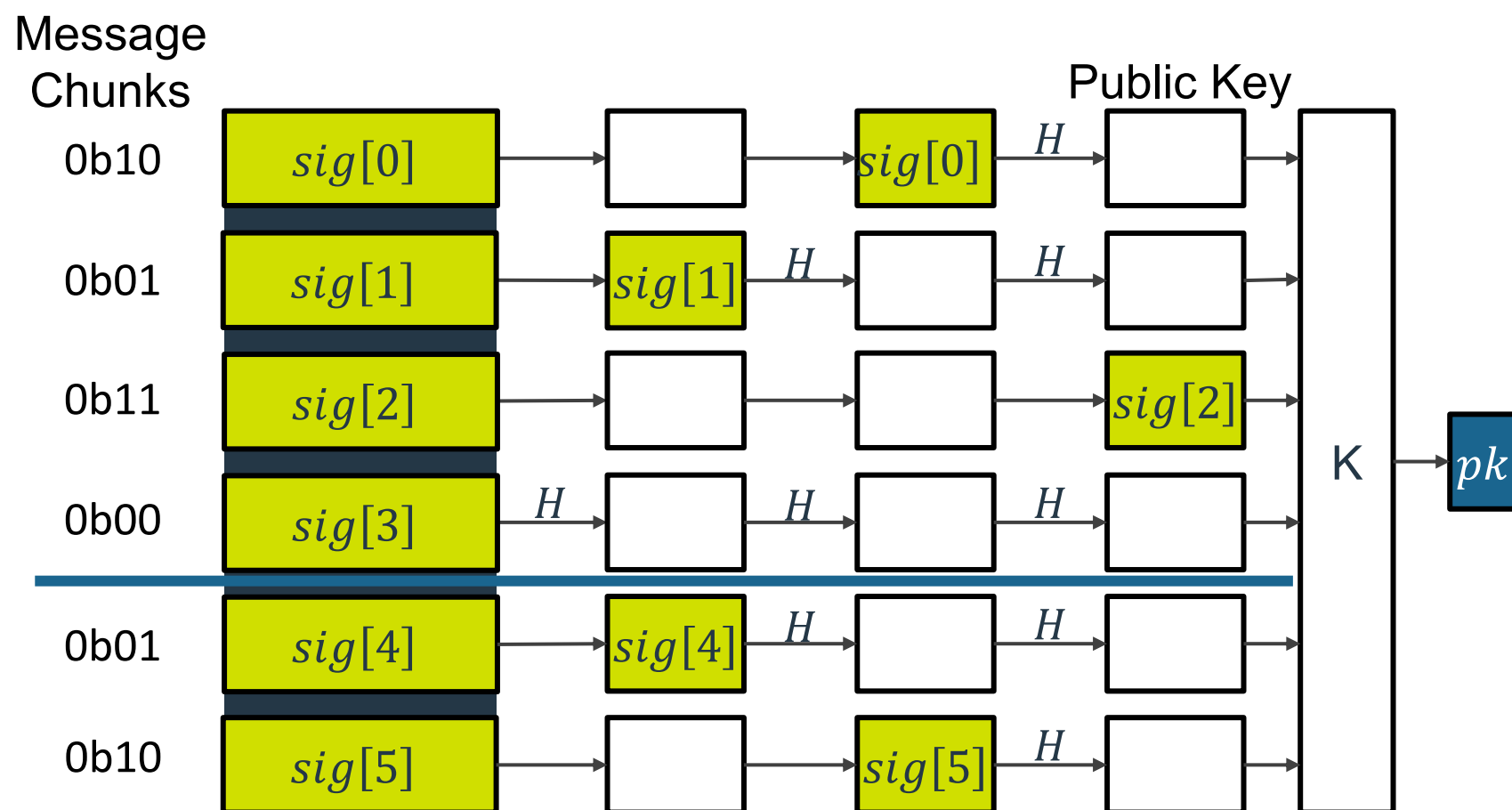
$$c = 1 + 2 + 0 + 3$$

$$c = 6 = 0b0110$$



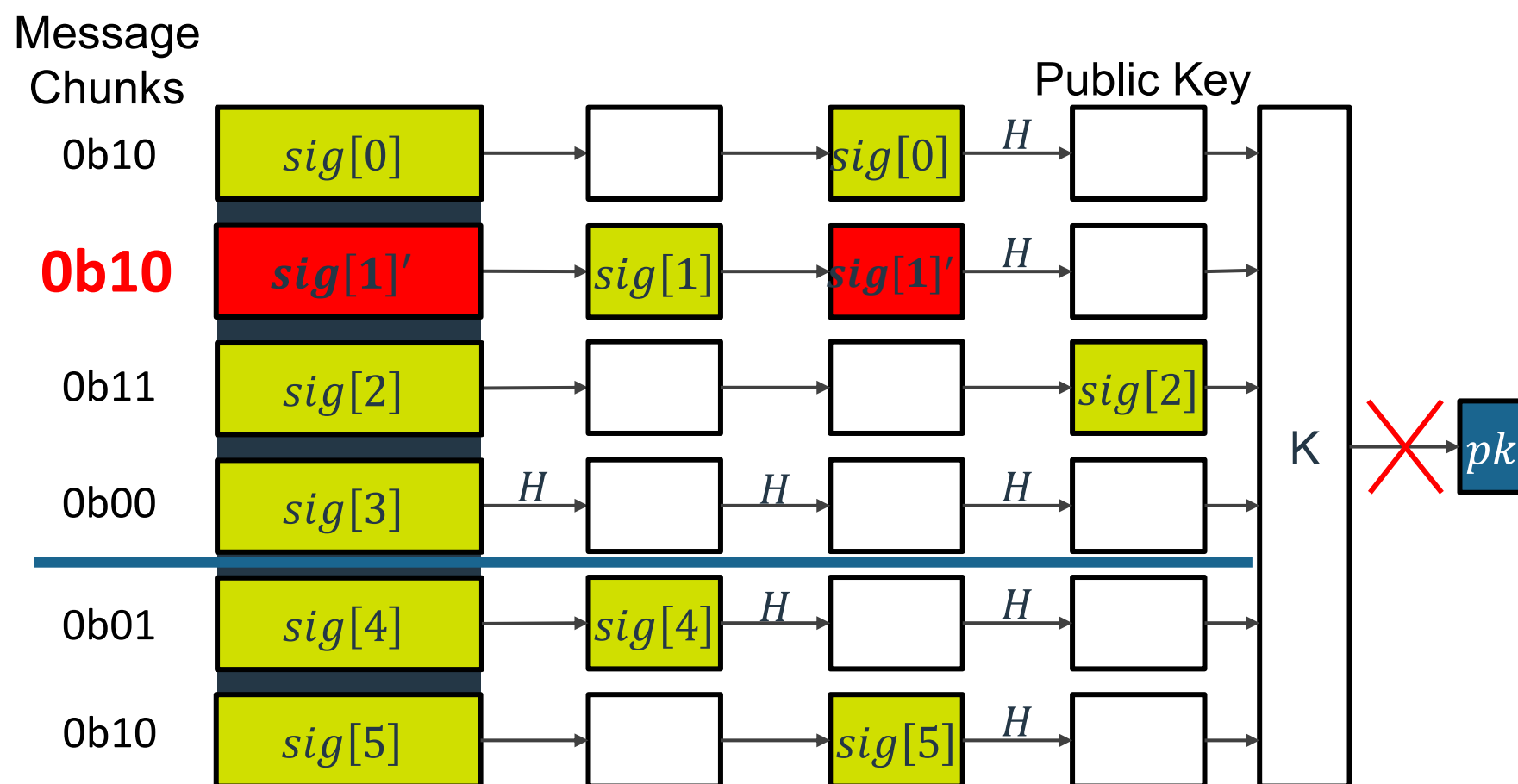
How WOTS+ Protects Against Forgery

- $pk[i] == hash_chain(sig[i], steps = w - 1 - m[i])$



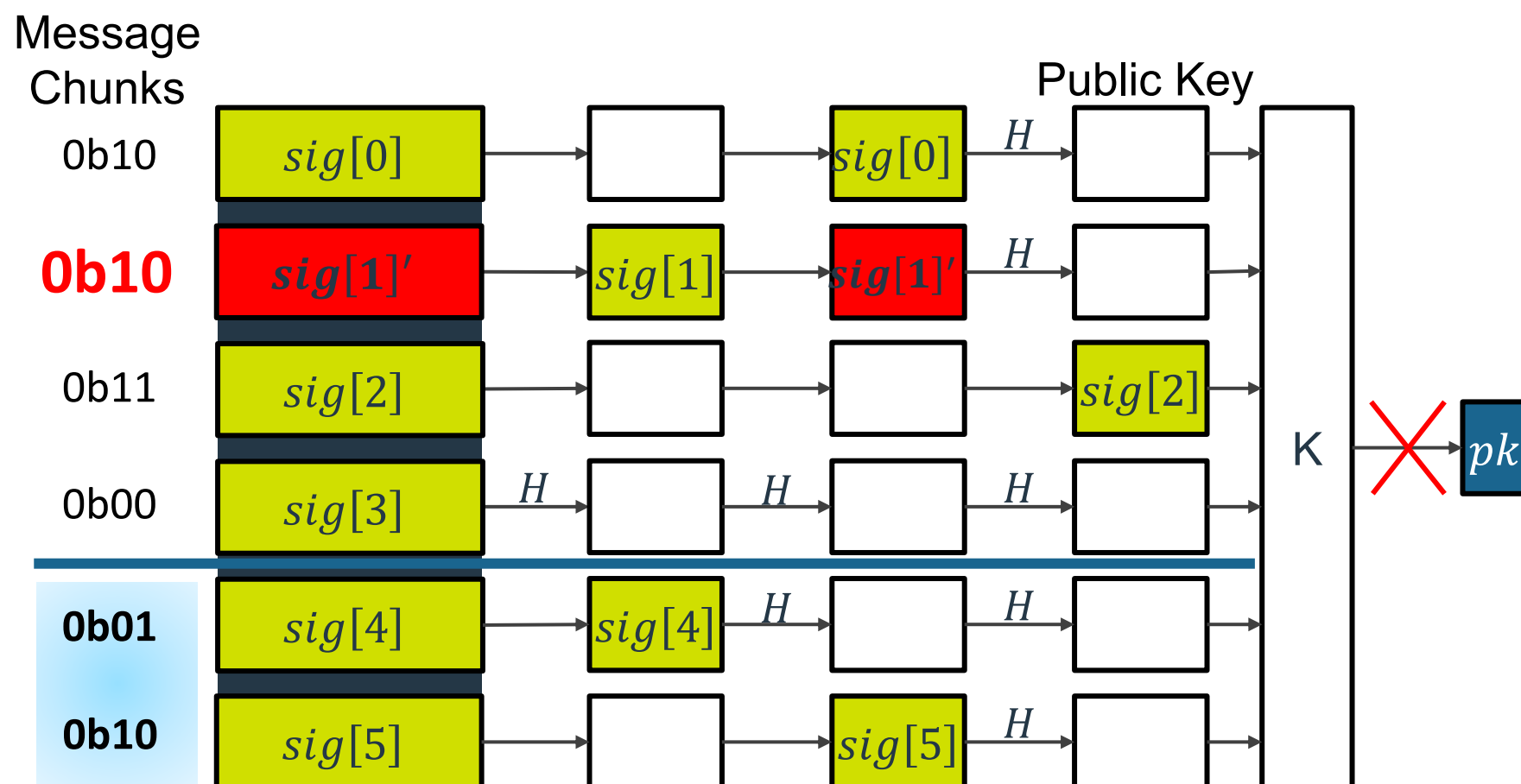
How WOTS+ Protects Against Forgery

- $c = 1 + \mathbf{1} + 0 + 3 = \mathbf{5}$



How WOTS+ Protects Against Forgery

- $c = 1 + \mathbf{1} + 0 + 3 = \mathbf{5}$



Faulting WOTS+ to forge Hash Based Signature

- This research builds on [Faulting Winternitz One-Time Signatures to forge LMS, XMSS, or SPHINCS+ signatures](#) (PQCrypto 2023)

Faulting WOTS+ to forge Hash Based Signature

- This research builds on [Faulting Winternitz One-Time Signatures to forge LMS, XMSS, or SPHINCS+ signatures](#) (PQCrypto 2023)
- The paper demonstrates an attack on the checksum calculation during signature verification using fault injection to alter normal behavior:

$$\text{sig_checksum}[j] \rightarrow \text{hash_chain}(\text{steps} = w - 1 - c'[j])$$

Faulting WOTS+ to forge Hash Based Signature

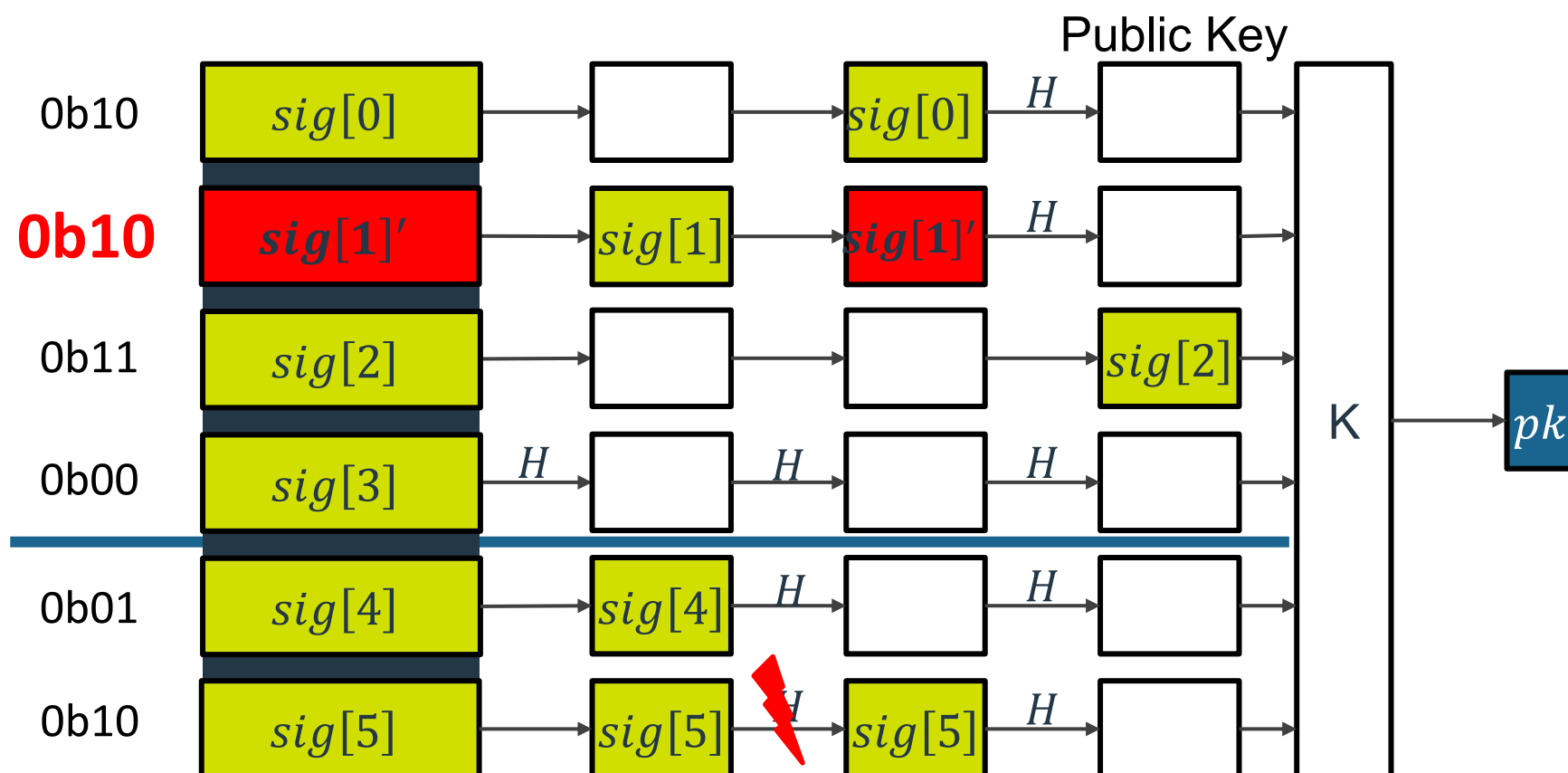
- This research builds on [Faulting Winternitz One-Time Signatures to forge LMS, XMSS, or SPHINCS+ signatures](#) (PQCrypto 2023)
- The paper demonstrates an attack on the checksum calculation during signature verification using fault injection to alter normal behavior:

$$sig_checksum[j] \rightarrow hash_chain(steps = w - 1 - c'[j])$$

- **Partial hash chain skip**
- **Full hash chain skip**

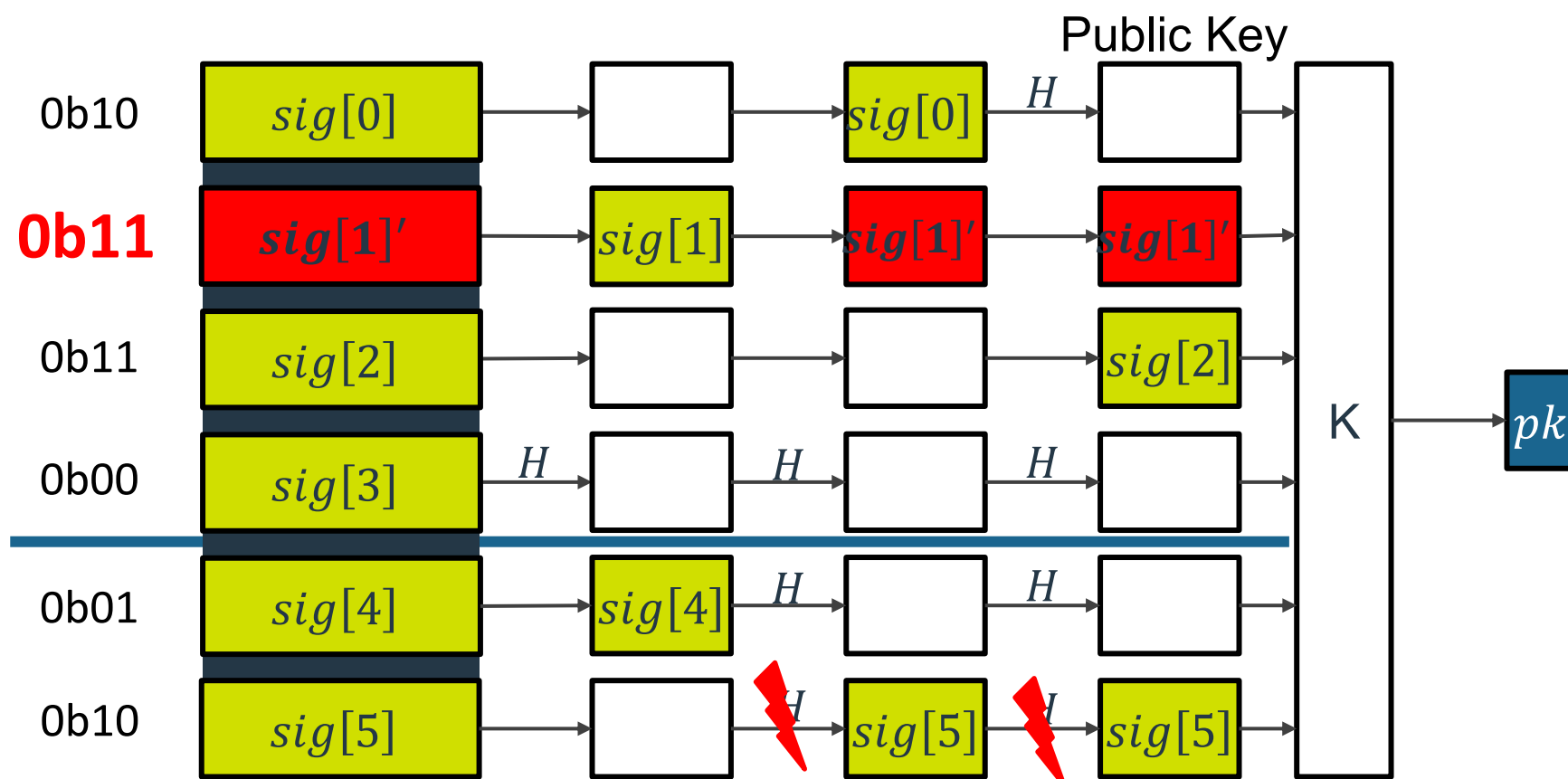
Partial hash chain skip

- $sig_checksum[j] \rightarrow hash_chain(steps = v')$ where $v' < (w - 1 - c'[j])$



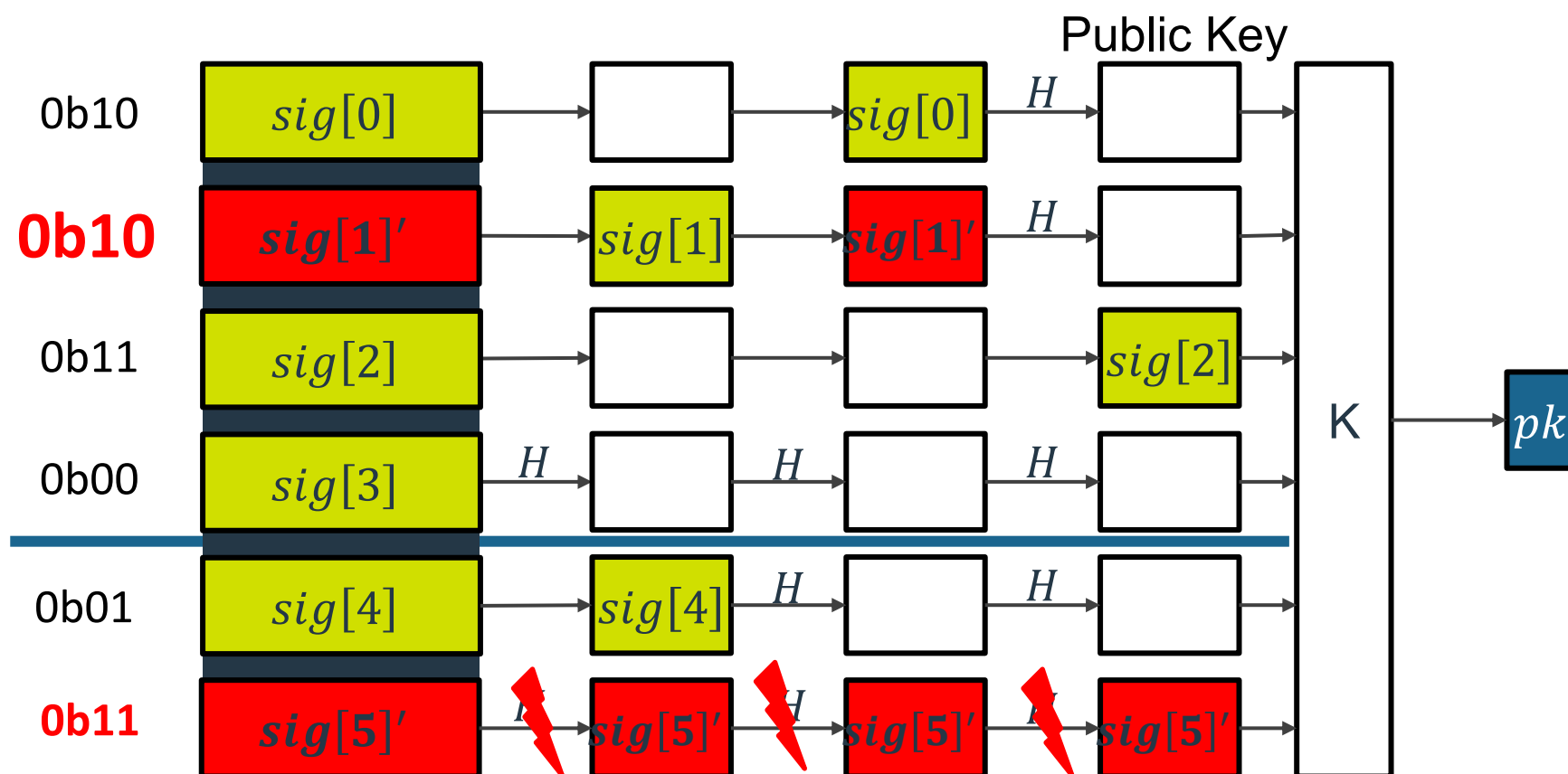
Full hash chain skip

- $sig_checksum[j] \rightarrow hash_chain(steps = 0)$



Full hash chain skip

- $sig_checksum[j] \rightarrow hash_chain(steps = 0)$



XMSS and SPHINCS+

- **XMSS (eXtended Merkle Signature Scheme)** is a stateful hash-based signature scheme built on WOTS+ and a Merkle tree.
Requires tracking a secret index to avoid key reuse.

XMSS and SPHINCS+

- **XMSS (eXtended Merkle Signature Scheme)** is a stateful hash-based signature scheme built on WOTS+ and a Merkle tree.

Requires tracking a secret index to avoid key reuse.

- **SPHINCS+** is a stateless hash-based scheme that combines **FORS** and a **hypertree of XMSS instances**.

Eliminates state management and supports flexible trade-offs in size and speed.

Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
void wots_pk_from_sig(const xmss_params *params, unsigned char *pk,
                    const unsigned char *sig, const unsigned char *msg,
                    const unsigned char *pub_seed, uint32_t addr[8]) {
    ...
    chain_lengths(params, lengths, msg);

    for(i = 0; i < params->wots_len; i++) {
        set_chain_addr(addr, i);
        gen_chain(params, pk + i*params->n, sig + i*params->n,
                lengths[i], params->wots_w - 1 - lengths[i], pub_seed0, addr);
    }
}
```


Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
void wots_pk_from_sig(const xmss_params *params, unsigned char *pk,
                    const unsigned char *sig, const unsigned char *msg,
                    const unsigned char *pub_seed, uint32_t addr[8]) {
    ...
    1 chain_lengths(params, lengths, msg); // [1]-> Attack to checksum calculation

    for(i = 0; i < params->wots_len; i++) {
        set_chain_addr(addr, i);
        gen_chain(params, pk + i*params->n, sig + i*params->n,
                  lengths[i], params->wots_w - 1 - lengths[i], pub_seed0, addr);
    }
}
```

Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
void wots_pk_from_sig(const xmss_params *params, unsigned char *pk,
                     const unsigned char *sig, const unsigned char *msg,
                     const unsigned char *pub_seed, uint32_t addr[8]) {
    ...
    1 chain_lengths(params, lengths, msg); // [1]-> Attack to checksum calculation

    for(i = 0; i < params->wots_len; i++) {
        set_chain_addr(addr, i);
        2 gen_chain(params, pk + i*params->n, sig + i*params->n, //[2]->Attack to checksum chunk
                    lengths[i], params->wots_w - 1 - lengths[i], pub_seed0, addr);
    }
}
```

Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
static void wots_checksum(...) {  
    ...  
    /* Compute checksum. */  
    for (i = 0; i < params->wots_len1; i++) {  
        csum += params->wots_w - 1 - msg_base_w[i];  
    }  
    /* Convert checksum to base_w. */  
    csum = csum << (8 - ((params->wots_len2 * params->wots_log_w) % 8));  
    ull_to_bytes(csum_bytes, sizeof(csum_bytes), csum);  
    base_w(params, csum_base_w, params->wots_len2, csum_bytes);  
}  
  
static void chain_lengths(...) {  
    base_w(params, lengths, params->wots_len1, msg);  
    wots_checksum(params, lengths + params->wots_len1, lengths);  
}
```


Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
static void wots_checksum(...) {  
    ...  
    /* Compute checksum. */  
    for (i = 0; i < params->wots_len1; i++) {  
        csum += params->wots_w - 1 - msg_base_w[i];  
    }  
    /* Convert checksum to base_w. */  
    csum = csum << (8 - ((params->wots_len2 * params->wots_log_w) % 8));  
    ull_to_bytes(csum_bytes, sizeof(csum_bytes), csum);  
    base_w(params, csum_base_w, params->wots_len2, csum_bytes);  
}  
  
static void chain_lengths(...) {  
    base_w(params, lengths, params->wots_len1, msg);  
    wots_checksum(params, lengths + params->wots_len1, lengths);  
}
```

Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
static void wots_checksum(...) {  
    ...  
    /* Compute checksum. */  
    for (i = 0; i < params->wots_len1; i++) {  
        csum += params->wots_w - 1 - msg_base_w[i];  
    }  
    /* Convert checksum to base_w. */  
    csum = csum << (8 - ((params->wots_len2 * params->wots_log_w) % 8));  
    ull_to_bytes(csum_bytes, sizeof(csum_bytes), csum);  
    base_w(params, csum_base_w, params->wots_len2, csum_bytes);  
}  
  
static void chain_lengths(...) {  
    base_w(params, lengths, params->wots_len1, msg);  
    wots_checksum(params, lengths + params->wots_len1, lengths);  
}
```

Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
static void wots_checksum(...) {  
    ...  
    /* Compute checksum. */  
    for (i = 0; i < params->wots_len1; i++) {  
        csum += params->wots_w - 1 - msg_base_w[i];  
    }  
    /* Convert checksum to base_w. */  
    csum = csum << (8 - ((params->wots_len2 * params->wots_log_w) % 8));  
    ull_to_bytes(csum_bytes, sizeof(csum_bytes), csum);  
    base_w(params, csum_base_w, params->wots_len2, csum_bytes);  
}  
  
static void chain_lengths(...) {  
    base_w(params, lengths, params->wots_len1, msg);  
    wots_checksum(params, lengths + params->wots_len1, lengths);  
}
```


Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
static void gen_chain(...)
{
    uint32_t i;

    /* Initialize out with the value at position 'start'. */
    memcpy(out, in, params->n);

    /* Iterate 'steps' calls to the hash function. */
    for (i = start; i < (start+steps) && i < params->wots_w; i++) {
        set_hash_addr(addr, i);
        thash_f(params, out, out, pub_seed, addr);
    }
}
```

Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
static void gen_chain(...)
{
    uint32_t i;

    /* Initialize out with the value at position 'start'. */
    memcpy(out, in, params->n);

    /* Iterate 'steps' calls to the hash function. */
    for (i = start; i < (start+steps) && i < params->wots_w; i++) {
        set_hash_addr(addr, i);
        thash_f(params, out, out, pub_seed, addr);
    }
}
```

Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
static void gen_chain(...)
{
    uint32_t i;

    /* Initialize out with the value at position 'start'. */
    memcpy(out, in, params->n);

    /* Iterate 'steps' calls to the hash function. */
    for (i = start; i < (start+steps) && i < params->wots_w; i++) {
        set_hash_addr(addr, i);
        thash_f(params, out, out, pub_seed, addr);
    }
}
```


Code Review of the WOTS+ Component in the Reference XMSS Implementation

```
static void gen_chain(...)
{
    uint32_t i;

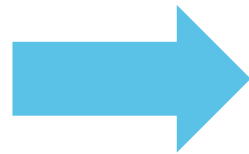
    /* Initialize out with the value at position 'start'. */
    memcpy(out, in, params->n);

    /* Iterate 'steps' calls to the hash function. */
    for (i = start; i < (start+steps) && i < params->wots_w; i++) {
        set_hash_addr(addr, i);
        thash_f(params, out, out, pub_seed, addr);
    }
}
```

Code Review of the WOTS+ Component in the Reference SPHINCS+ Implementation

- **SPHINCS+** reuses the same **WOTS+** code as **XMSS** for signature verification.
- As a result, all fault injection **attacks** demonstrated against XMSS also apply directly to SPHINCS+.

How to Target?



Chose the fault
injection point

How to Target?



Chose the fault
injection point

**Brute-force
search** for a
hash digest
m'

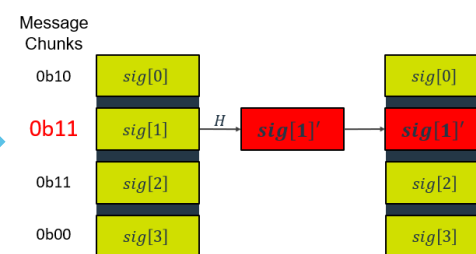
How to Target?



Chose the fault injection point



Brute-force search for a hash digest m'



Forge a signature for m'

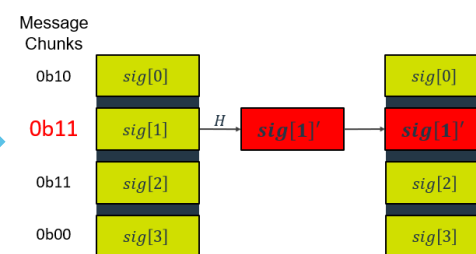
How to Target?



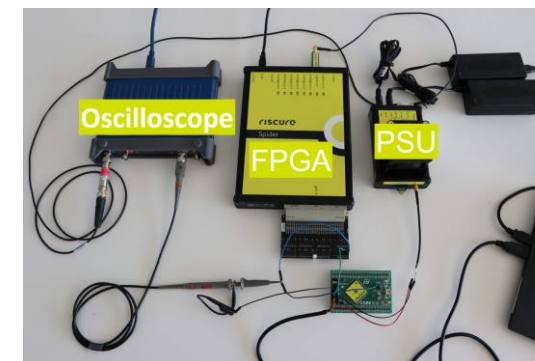
Chose the fault injection point



Brute-force search for a hash digest m'



Forge a signature for m'

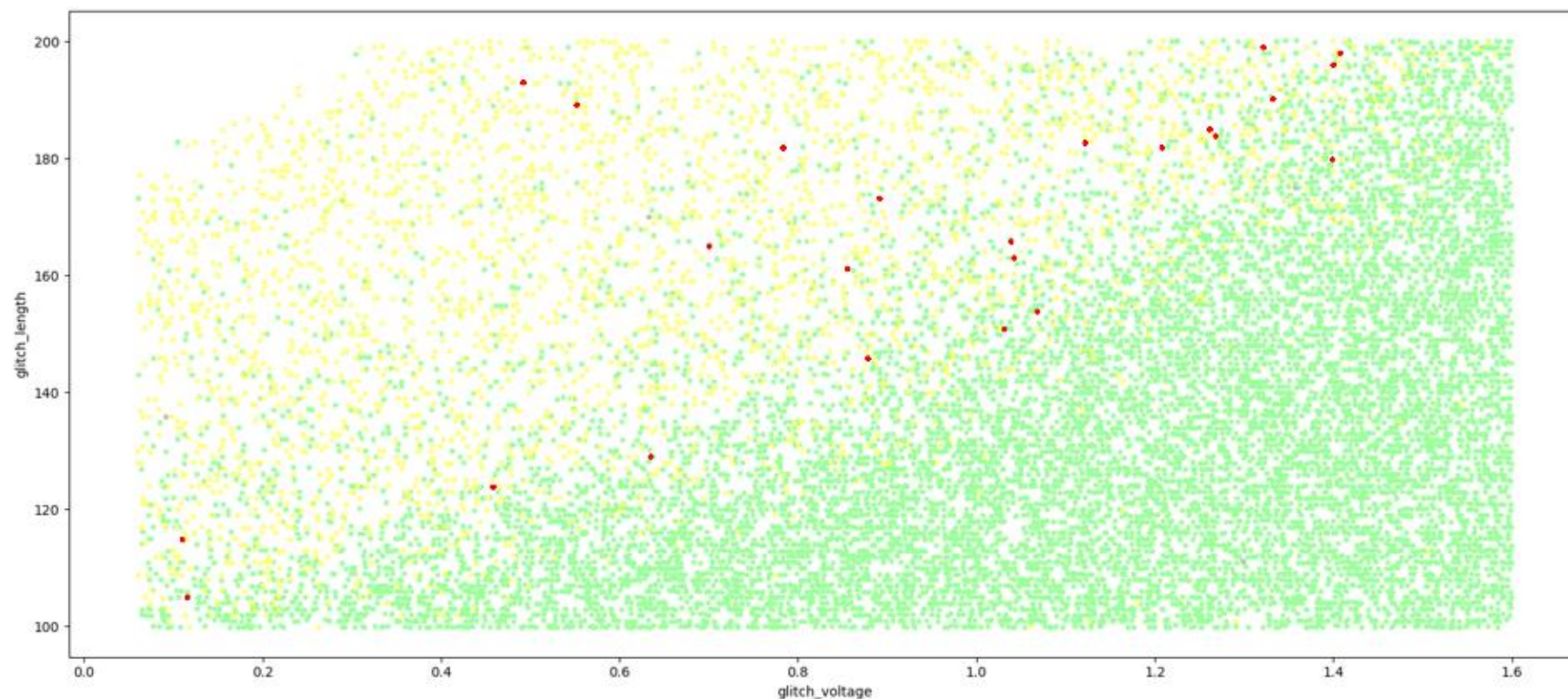


Inject the fault at the target **during verification**

Fault Injection Results of XMSS

Fault Injection Results – Skipping Hash Chains

- Green: Normal Execution
- Yellow: Mute/Reset
- Red: Success



Fault Injection Plot of Sampling of C Scenario

Summary – XMSS & SPHINCS+

- We identified multiple fault injection targets in **WOTS+**, which is used in both **XMSS** and **SPHINCS+**.

Summary – XMSS & SPHINCS+

- We identified multiple fault injection targets in **WOTS+**, which is used in both **XMSS** and **SPHINCS+**.
- The **checksum calculation** is a broad and critical target. While skipping hash chains is easy, attacking the checksum calculation requires **precise local fault injection** (e.g., laser FI, EMFI).

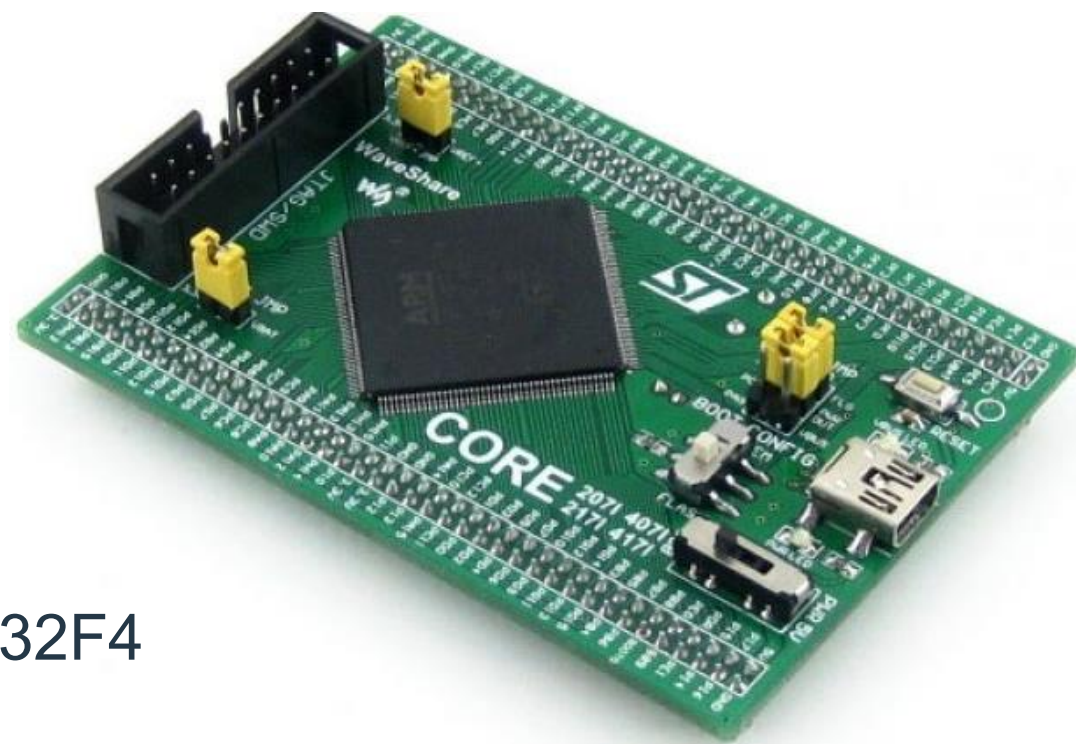
Summary – XMSS & SPHINCS+

- We identified multiple fault injection targets in **WOTS+**, which is used in both **XMSS** and **SPHINCS+**.
- The **checksum calculation** is a broad and critical target. While skipping hash chains is easy, attacking the checksum calculation requires **precise local fault injection** (e.g., laser FI, EMFI).
- Since **SPHINCS+** reuses the same **WOTS+** code, all attack techniques against XMSS verification **apply to SPHINCS+**.

Bonus: Fault Injection on Fault Resistance XMSS Library

Introduction

- Fox Crypto released XMSS v1.0 **before** fault injection attacks on WOTS+ were published.
- The library includes **fault injection resistance for verification**, as stated in their documentation and presentations*.
- **Our goal:** Apply the attack on WOTS+
- We implemented Fox Crypto's XMSS library on our STM32F4 target
- We used the same Voltage Fault Injection Setup from our earlier XMSS research.



Source: [Production ready XMSS](#)

Vulnerability We Found in Fox Crypto XMSS Implementation

```
static void chain(...) {  
    ...  
    input_prf->M.ADRS.typed.OTS_Hash_Address.hash_address = start_index; // [1]  
  
    assert(start_index + num_steps < W); // [2]  
  
    native_256_copy(output, input);  
  
    for (uint_fast8_t i = 0; i < num_steps; i++) { // [3]  
        ...  
        input_prf->M.ADRS.typed.OTS_Hash_Address.keyAndMask = 1;  
        xmss_PRF(HASH_ABSTRACTION(hashes) &input_f.M, input_prf);  
        for (uint_fast8_t j = 0; j < XMSS_VALUE_256_WORDS; j++) {  
            input_f.M.data[j] ^= output->data[j];  
        }  
        xmss_F(HASH_ABSTRACTION(hashes) output, &input_f);  
        input_prf->M.ADRS.typed.OTS_Hash_Address.hash_address += 1; // [4]  
    }  
}
```


Vulnerability We Found in Fox Crypto XMSS Implementation

```
static void chain(...) {  
    ...  
    input_prf->M.ADRS.typed.OTS_Hash_Address.hash_address = start_index; // [1]  
  
    assert(start_index + num_steps < W); // [2]  
  
    native_256_copy(output, input);  
  
    for (uint_fast8_t i = 0; i < num_steps; i++) { // [3]  
        ...  
        input_prf->M.ADRS.typed.OTS_Hash_Address.keyAndMask = 1;  
        xmss_PRF(HASH_ABSTRACTION(hashes) &input_f.M, input_prf);  
        for (uint_fast8_t j = 0; j < XMSS_VALUE_256_WORDS; j++) {  
            input_f.M.data[j] ^= output->data[j];  
        }  
        xmss_F(HASH_ABSTRACTION(hashes) output, &input_f);  
        input_prf->M.ADRS.typed.OTS_Hash_Address.hash_address += 1; // [4]  
    }  
}
```


Vulnerability We Found in Fox Crypto XMSS Implementation

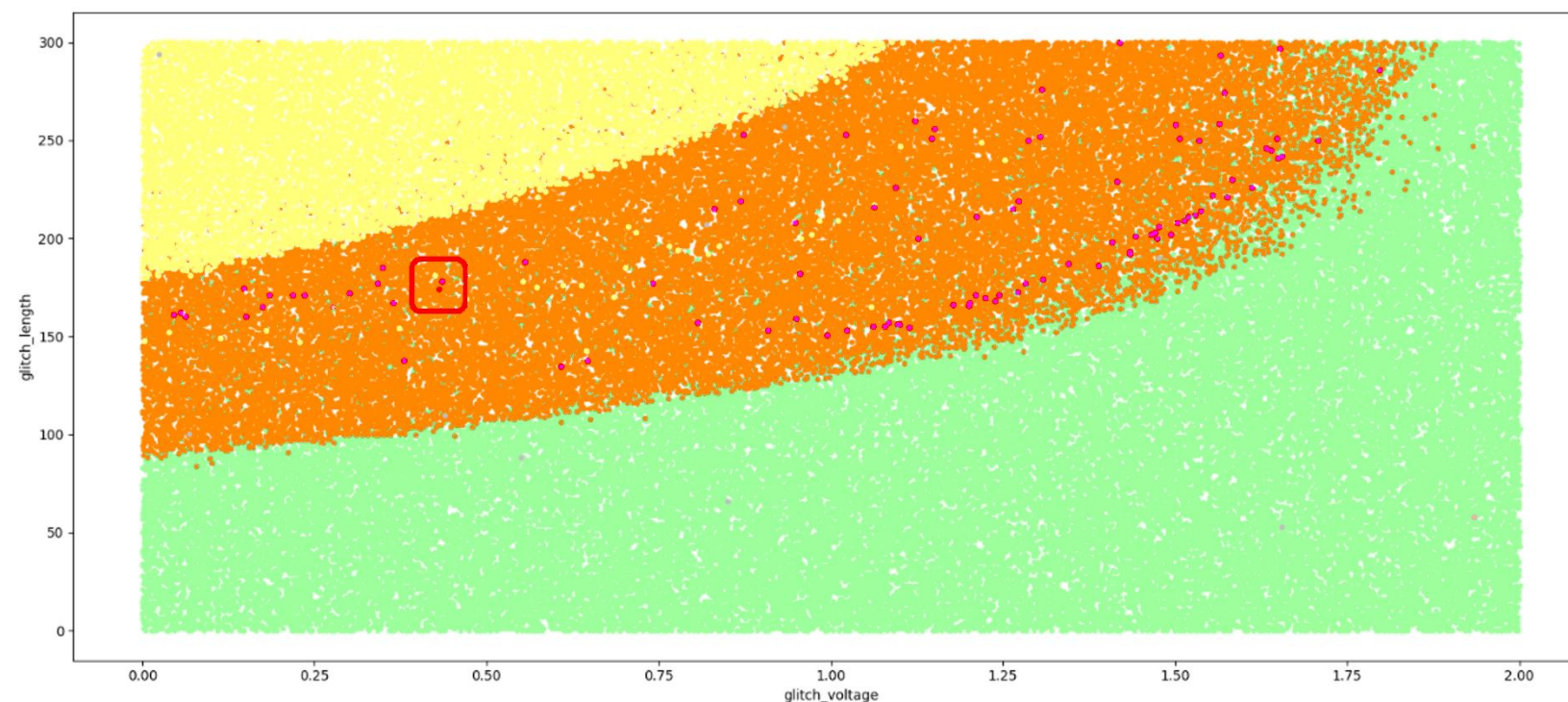
```
static void chain(...) {  
    ...  
    input_prf->M.ADRS.typed.OTS_Hash_Address.hash_address = start_index; // [1]  
  
    assert(start_index + num_steps < W); // [2]  
  
    native_256_copy(output, input);  
  
    for (uint_fast8_t i = 0; i < num_steps; i++) { // [3]  
        ...  
        input_prf->M.ADRS.typed.OTS_Hash_Address.keyAndMask = 1;  
        xmss_PRF(HASH_ABSTRACTION(hashes) &input_f.M, input_prf);  
        for (uint_fast8_t j = 0; j < XMSS_VALUE_256_WORDS; j++) {  
            input_f.M.data[j] ^= output->data[j];  
        }  
        xmss_F(HASH_ABSTRACTION(hashes) output, &input_f);  
        input_prf->M.ADRS.typed.OTS_Hash_Address.hash_address += 1; // [4]  
    }  
}
```

Vulnerability We Found in Fox Crypto XMSS Implementation

```
static void chain(...) {  
    ...  
    input_prf->M.ADRS.typed.OTS_Hash_Address.hash_address = start_index; // [1]  
  
    assert(start_index + num_steps < W); // [2]  
  
    native_256_copy(output, input);  
  
     for (uint_fast8_t i = 0; i < num_steps; i++) { // [3]  
        ...  
        input_prf->M.ADRS.typed.OTS_Hash_Address.keyAndMask = 1;  
        xmss_PRF(HASH_ABSTRACTION(hashes) &input_f.M, input_prf);  
        for (uint_fast8_t j = 0; j < XMSS_VALUE_256_WORDS; j++) {  
            input_f.M.data[j] ^= output->data[j];  
        }  
        xmss_F(HASH_ABSTRACTION(hashes) output, &input_f);  
        input_prf->M.ADRS.typed.OTS_Hash_Address.hash_address += 1; // [4]  
    }  
}
```


Fault Injection Results of Fox Crypto XMSS Implementation

- We used same Voltage Fault Injection Setup
- Green: Normal Behaviour
- Orange and Purple: Hash chain skip is performed, but the library **returns an error** due to the **countermeasure checks**
- Yellow: Mute/Reset
- Red: Success



Fault Injection Plot of Full Hash Chain Skip

Vulnerability Disclosure Timeline

- **Discovery & Validation:**

- [26.04.2024] – Vulnerability identified and validated internally.

- **Initial Contact with Fox Crypto:**

- [02.05.2024] – Notified Fox Crypto about the identified vulnerability.
- [16.05.2024] – We presented the vulnerability and FI results to Fox Crypto.

- **Public Disclosure:**

- [17.05.2024] – Fox Crypto created a public security issue on GitHub regarding the vulnerability.

- **Fox Crypto Fix Released:**

- [08.10.2024] – Fox Crypto released version 2.0 of the XMSS library, confirming the fix for the vulnerability.
- We appreciate Fox Crypto's timely response and transparency in addressing the issue.

Key Takeaways and Conclusions

Key Takeaways and Conclusions

- Even “quantum-safe” code can be **glitched**. PQC is **not immune** to FI.

Key Takeaways and Conclusions

- Even “quantum-safe” code can be **glitched**. PQC is **not immune** to FI.
- **Signature forgery** is possible **without breaking** crypto, just by skipping checks.

Key Takeaways and Conclusions

- Even “quantum-safe” code can be **glitched**. PQC is **not immune** to FI.
- **Signature forgery** is possible **without breaking** crypto, just by skipping checks.
- New fault targets continue to emerge. Attackers adapt quickly, so defenses must evolve just as fast.

Key Takeaways and Conclusions

- Even “quantum-safe” code can be **glitched**. PQC is **not immune** to FI.
- **Signature forgery** is possible **without breaking** crypto, just by skipping checks.
- New fault targets continue to emerge. Attackers adapt quickly, so defenses must evolve just as fast.
- Implementation security is the **next battleground** for PQC.



AUGUST 6-7, 2025
MANDALAY BAY / LAS VEGAS



Thank you