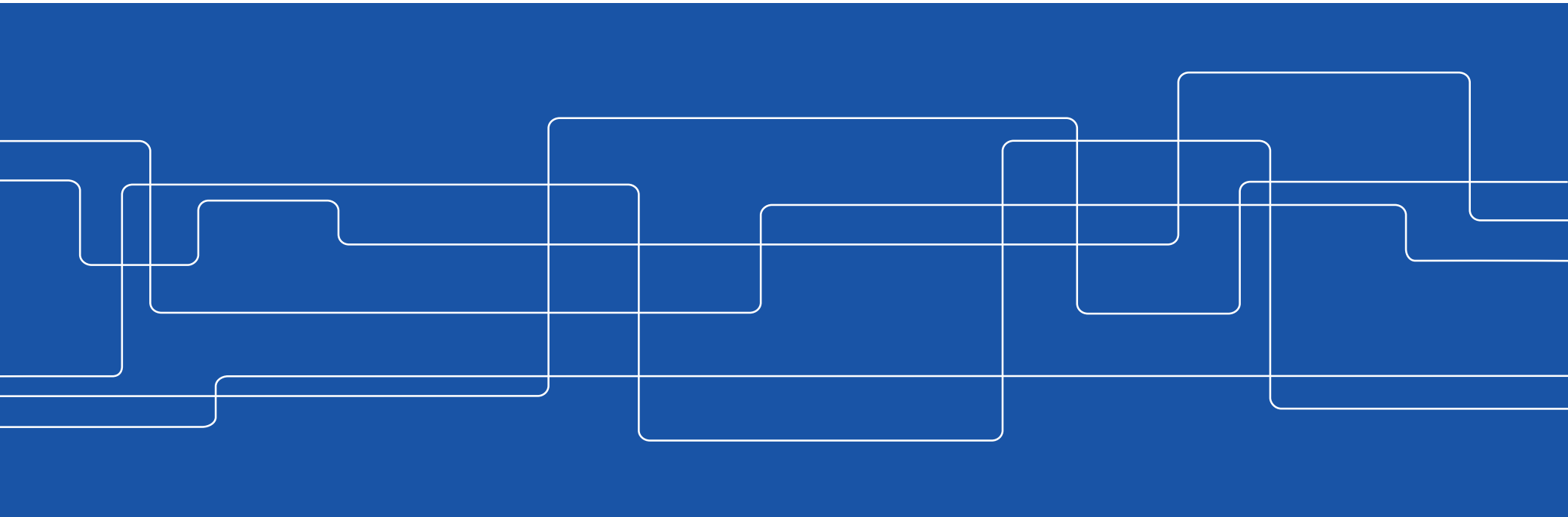
Side-Channel and Fault Attacks on ML-KEM and ML-DSA

Elena Dubrova
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden



Outline

- Background
  - Public-key, secret-key and post-quantum cryptography
  - Learning With Errors (LWE) problem
  - ML-KEM (Kyber) and ML-DSA (Dilithium) algorithms
  - Side-channel and fault attacks
- Attacks on software implementations of ML-KEM and ML-DSA algorithms
- Summary

Discrete Logarithm

Integer Factorization

Public-Key Cryptographic Algorithms

Depend on hardness of

Mathematical Problems

Can solve

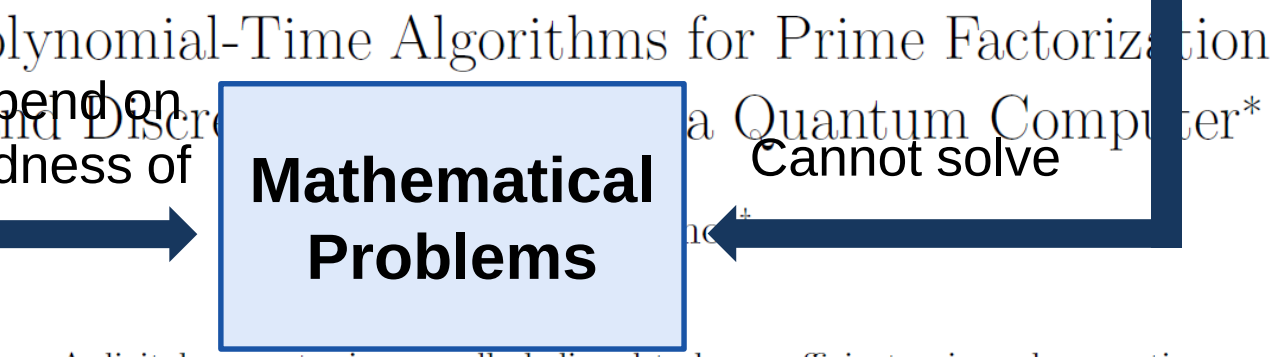
Quantum Computers

Post-Quantum Cryptographic Algorithms

Depend on hardness of

Mathematical Problems

Cannot solve


Polynomial-Time Algorithms for Prime Factorization
and Discr a Quantum Comput er*

A digital computer is generally believed to be an efficient universal computing device; tha physical computing device with an increa nomial factor. This may not be true wh leration. This paper considers

Lattice-based

Code-based

When will a large-scale quantum computer be built?

”I estimate a 1/7 chance of breaking RSA-2048 by 2026 and a 1/2 chance by 2031.”

Michele Mosca [NIST, April 2015]
https://eprint.iar.org/2015/1075

Public-Key Cryptography (Assymetric)

RSA

Encryption — Data confidentially

DSA

Digital Signature — Data integrity & authenticity Non-repudiation

Diffie-Hellman

Key Exchange — Shared key establishment

Secret-Key Cryptography (Symmetric)

AES

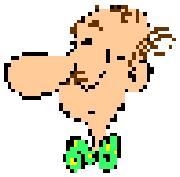
Encryption — Data confidentially

HMAC SHA-256

Message Authentication — Data integrity & authenticity

Authenticated key exchange & symmetric encryption

Alice



Bob

Alice's private key sk_A
Bob's public key pk_B

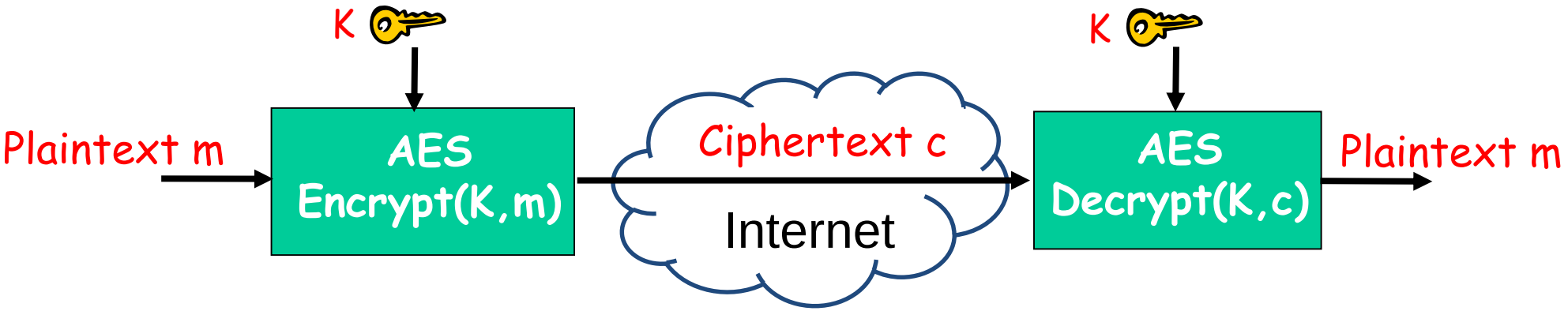
Authentication using digital signatures

Bob's private key sk_B
Alice's public key pk_A

Shared key K

Key establishment using Diffie-Hellman

Shared key K



Post-Quantum Cryptography

Both algorithms rely on hardness of Learning With Errors (LWE) problem

Data confidentially

Encryption

CRYSTALS-Kyber, (NIST ML-KEM, 2022)

Data integrity & authenticity Non-repudiation

Digital Signature

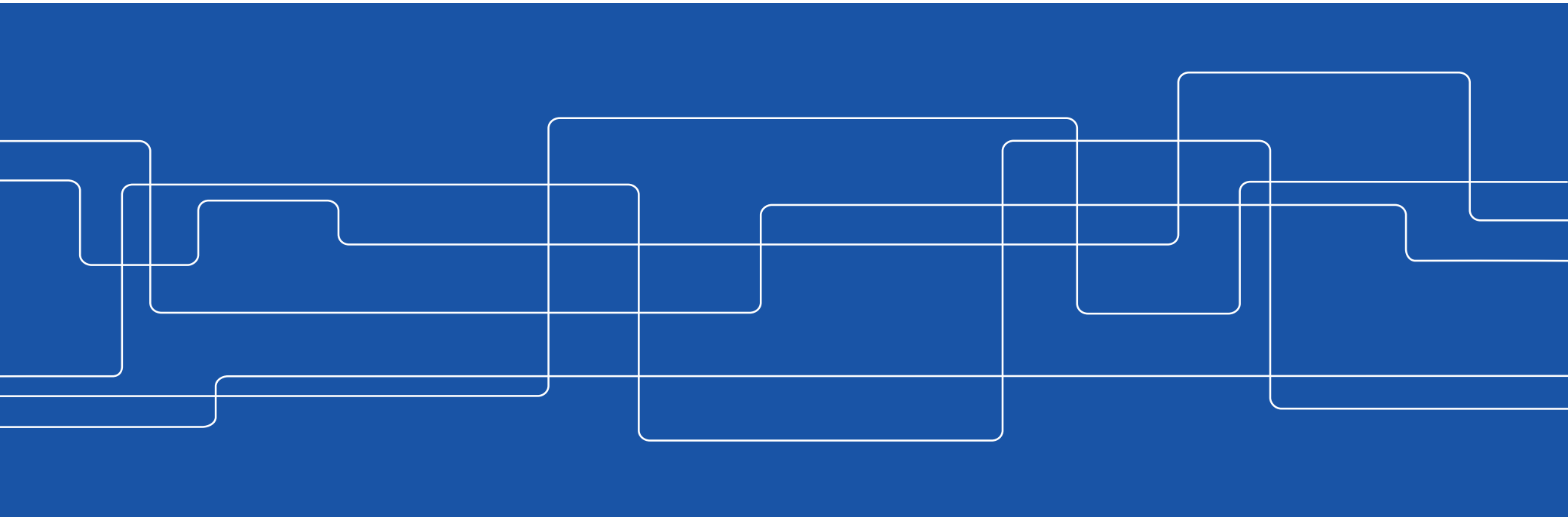
CRYSTALS-Dilithium (NIST ML-DSA, 2022)

Shared key establishment

Key Encapsulation

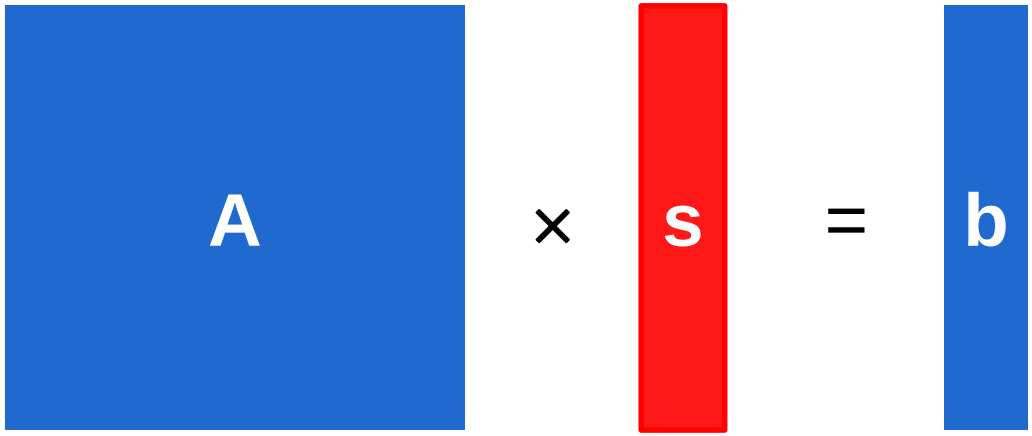
CRYSTALS-Kyber, (NIST ML-KEM, 2022)

Learning With Errors



Learning With Errors (LWE) problem

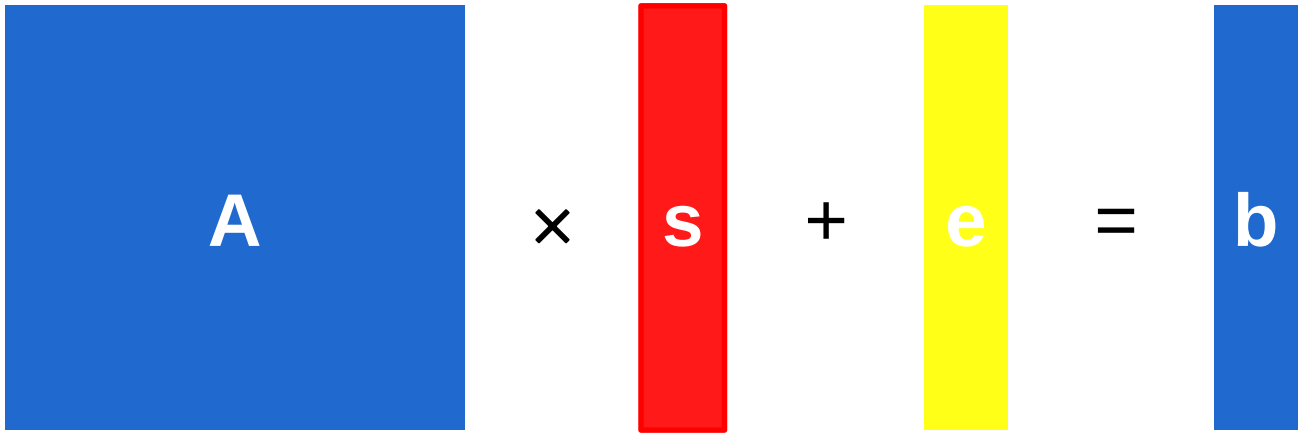
A system of linear equations can be easily solved ...



Given blue, find red

Learning With Errors (LWE) problem, cont.

... but not if errors (small noise) are added

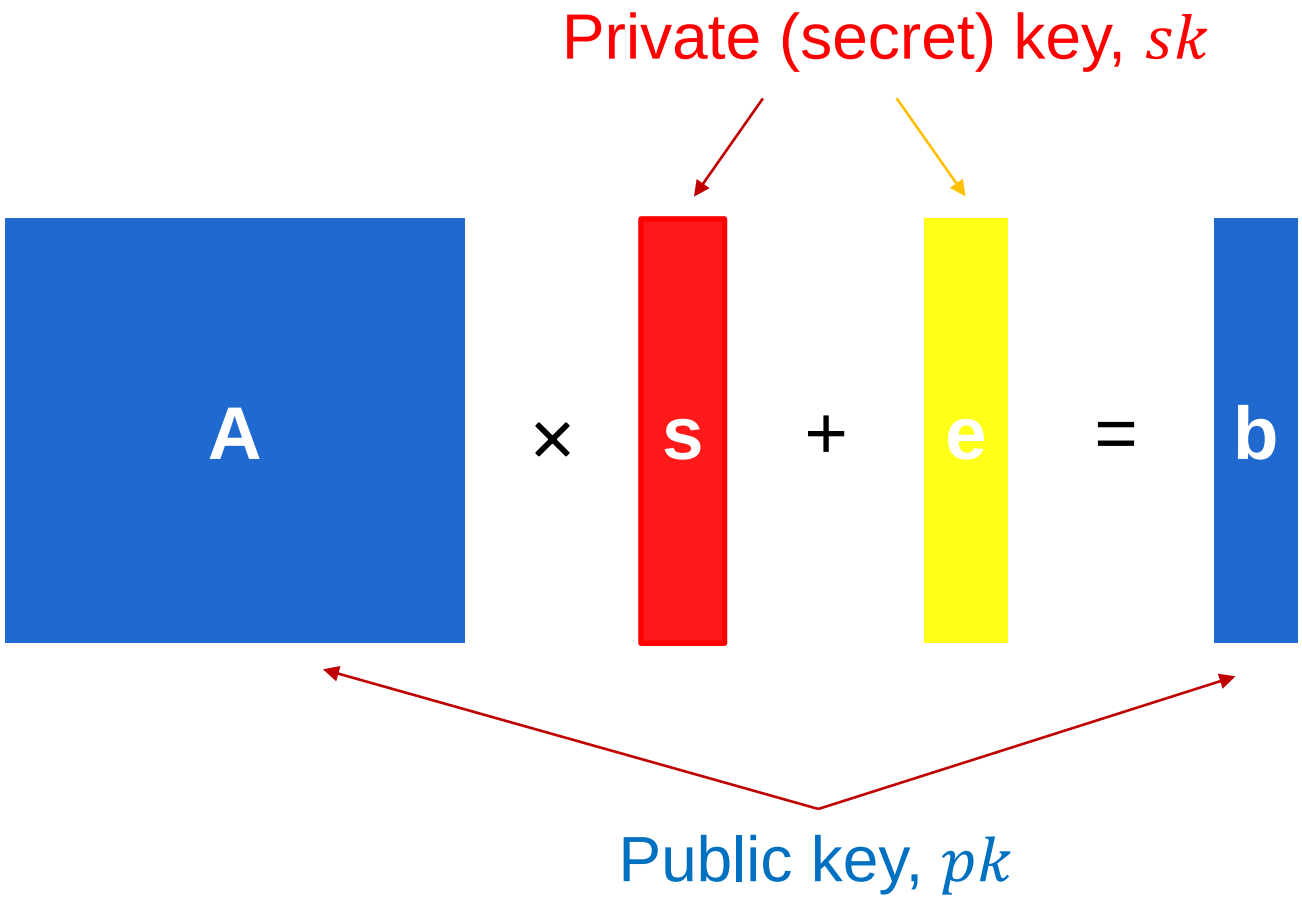


Given blue, find red (search LWE problem)

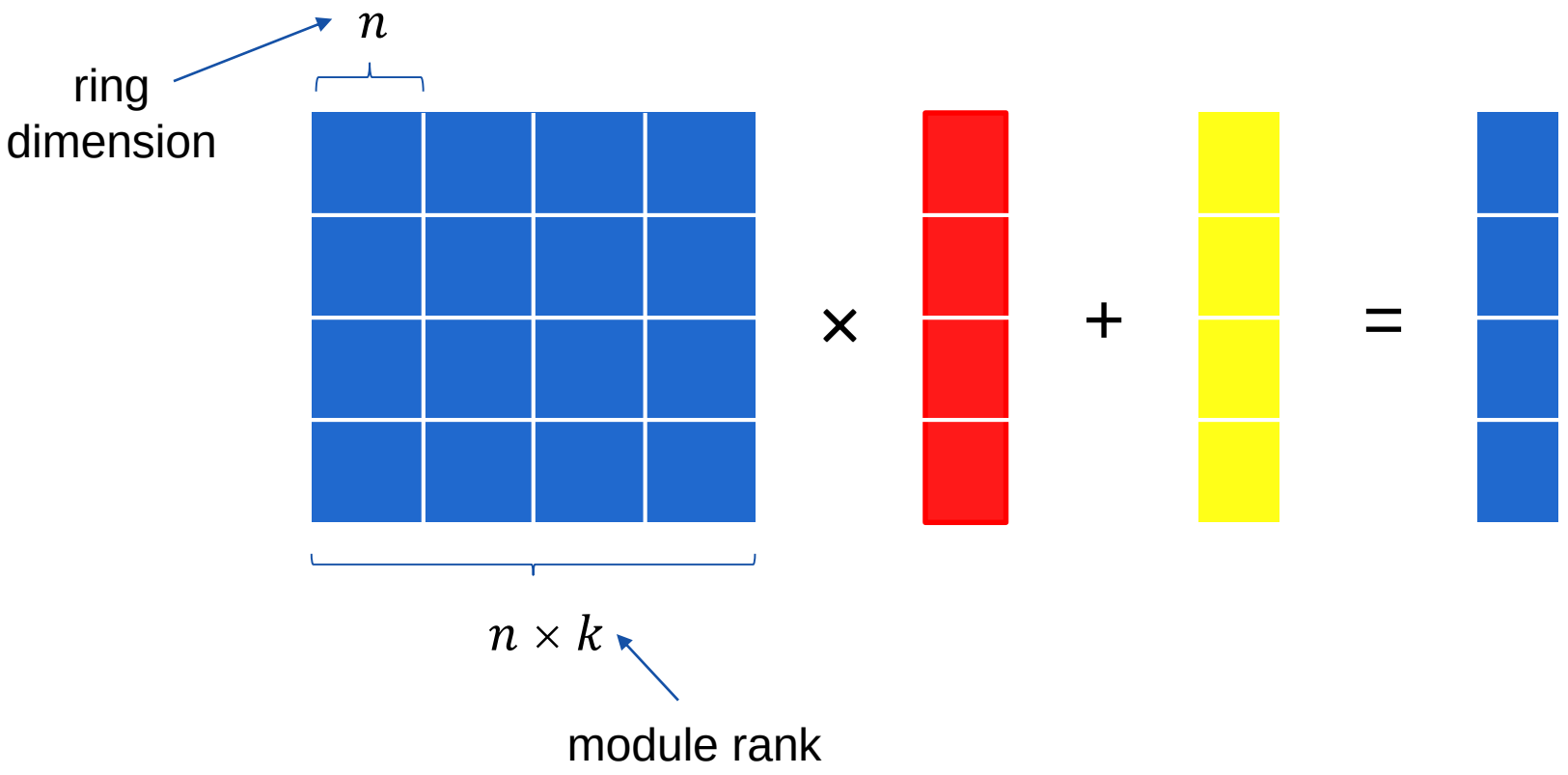
Given A, distinguish b from unifrmy random (decision LWE problem)

Hard even if A is over ring $Z_q[X]/f(X)$ for certain $f(X)$

Learning With Errors (LWE) problem, cont.

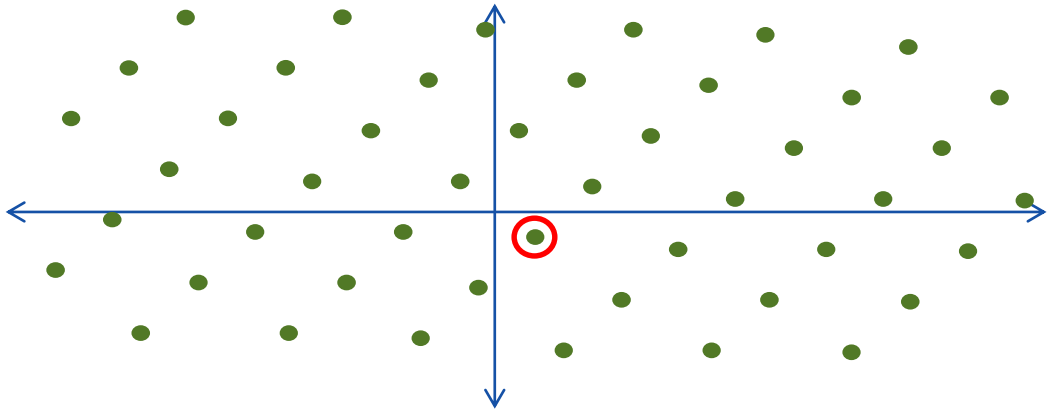


Module LWE



Why ``lattice-based´´?

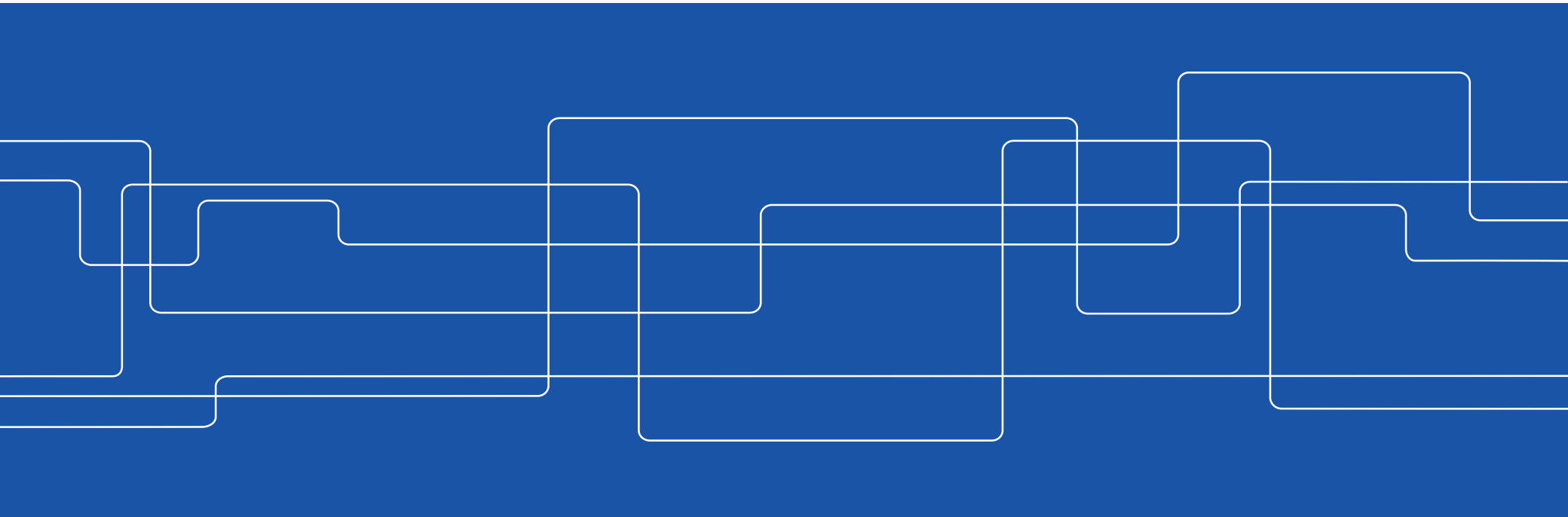
- Module-LWE problem can be interpreted as a version of the Closest Vector Problem (CVP) in a structured *q*-ary lattice
- This CVP instance can be solved by finding an unusually short vector in a related lattice
  - a version of the Shortest Vector Problem (SVP)



All solutions to
b = As + e mod *q*
form a “shifted” lattice

The goal is to find the point closest to the origin

ML-KEM (CRYSTALS-Kyber)



Module Lattice Key Encapsuation Mechanism

- Security is based on the hardness of LWE in module lattices
- PKE algorithms:
  - Key generation, (pk, sk) = PKE.KeyGen()
  - Encryption, c = Encrypt(pk, m, r)
  - Decryption, m = Decrypt(sk, c)
- KEM algorithms:
  - Key generation, (pk, sk) = KEM.KeyGen()
  - Encapsulation, (c, K) = Encaps(pk)
  - Decapsualtion, K = Decaps(c, sk)

▷ pk is public key

▷ sk is secret (private) key

▷ r is random coin

▷ m is message

▷ c is ciphertext, $c = (\boldsymbol{u}, v)$

▷ K is shared key

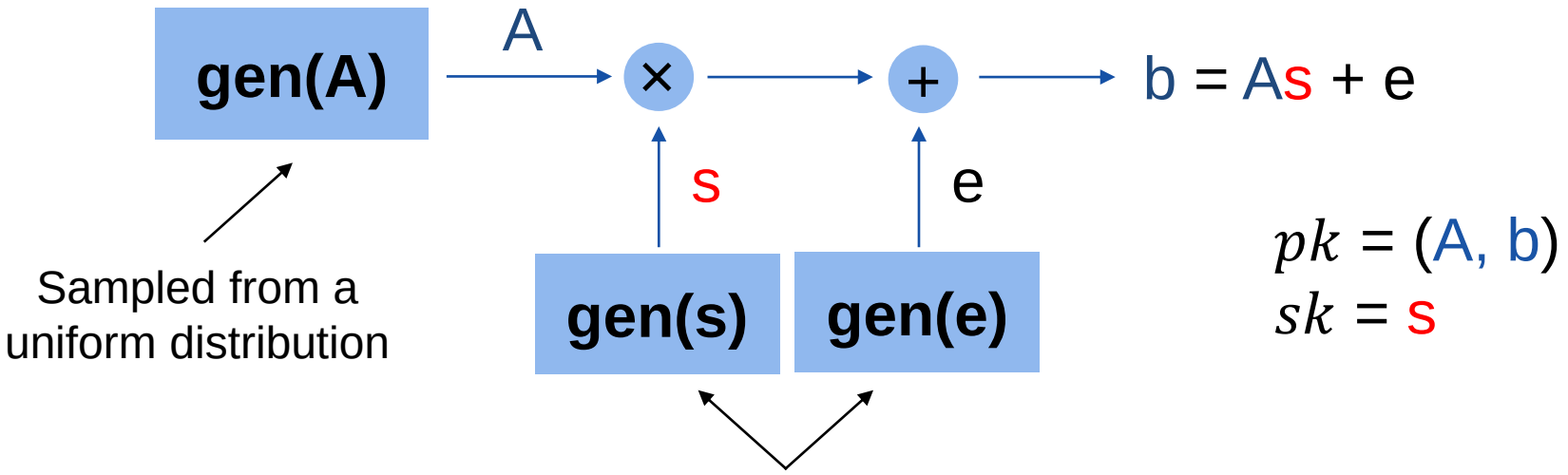
ML-KEM parameters

	n	k	q	η_1	η_2	(d_u, d_v)
KYBER512	256	2	3329	3	2	(10, 4)
KYBER768	256	3	3329	2	2	(10, 4)
KYBER1024	256	4	3329	2	2	(11, 5)

- $\mathbb{Z}_q$ is the ring of integers modulo a prime $q = 2^{13} – 2^9 + 1$
- R_q is the polynomial ring $\mathbb{Z}_q[X]/(Xn + 1)$, where n is the ring dimension
- ML-KEM works with vectors of ring elements in R_q^k, where *k* is the rank of the module defining the security level
- Inputs and outputs to all API functions of ML-KEM are byte arrays

Key generation algorithm

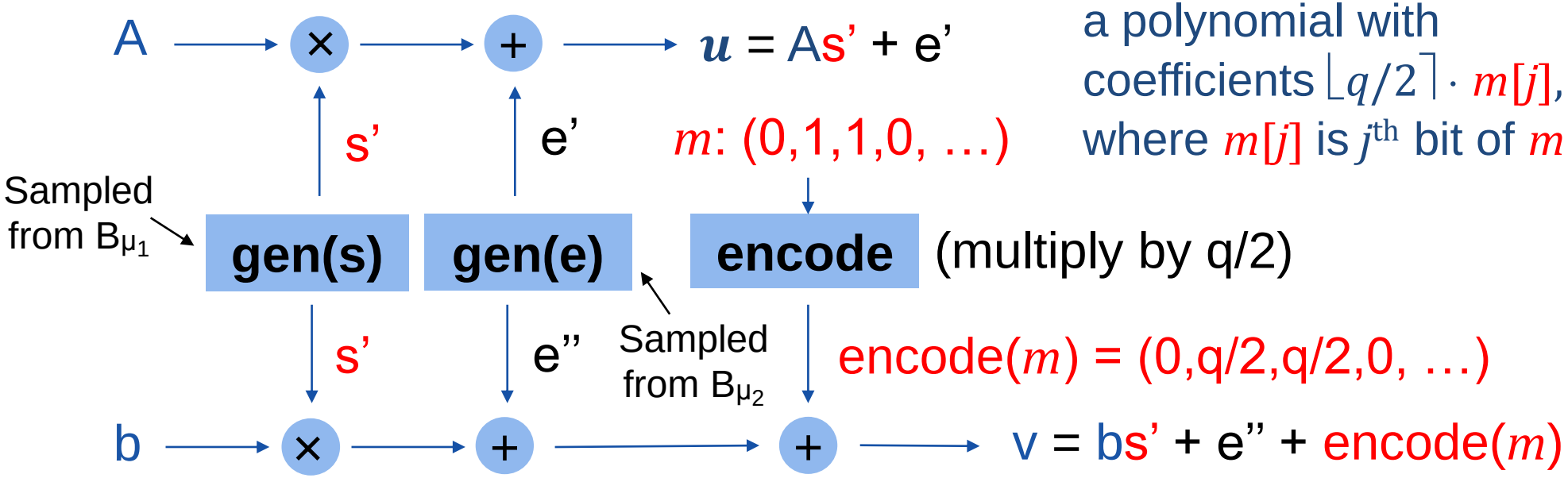
Output: public key pk, secret key sk



Encryption algorithm

Input: public key pk = (A, b), message m

Output: ciphertext c = (u, v)

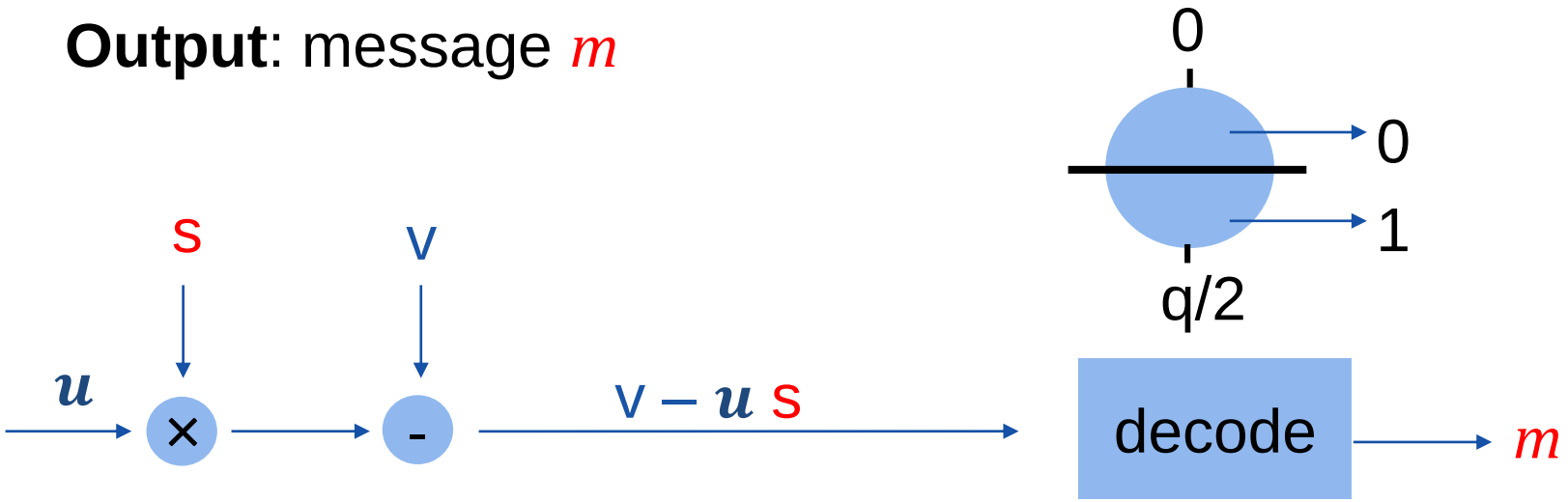


Encode converts a binary message m into a polynomial with coefficients $\lfloor q/2 \rceil \cdot m[j]$, where $m[j]$ is j^{th} bit of m

Decryption algorithm

Input: ciphertext c = ($\boldsymbol{u}$, v), secret key sk = s

Output: message m



= encode(m) + (es’ + e’’ + e’s)

small

b = As + e

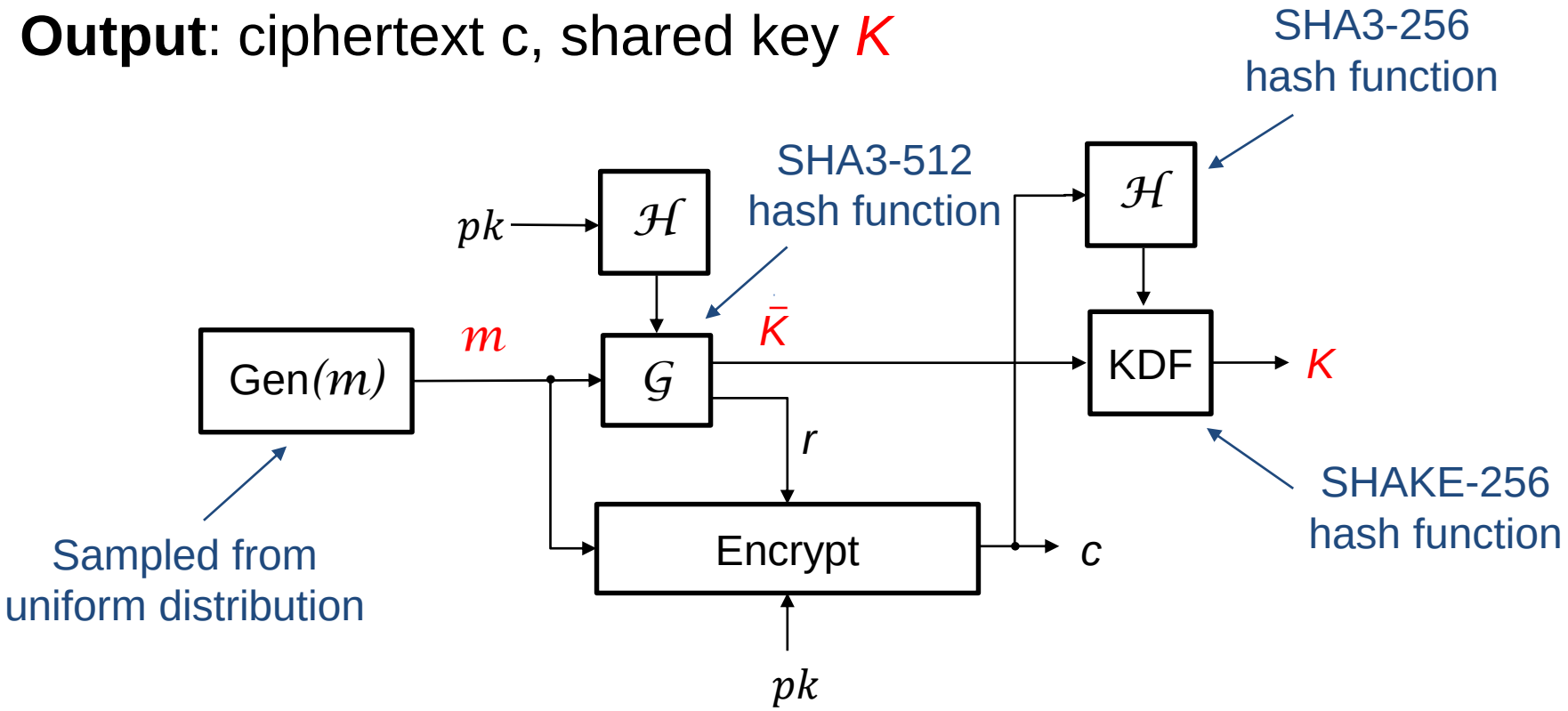
v = bs’ + e’’ + encode(m)

$\boldsymbol{u}$ = As’ + e’

Encapsulation algorithm

Input: public key pk

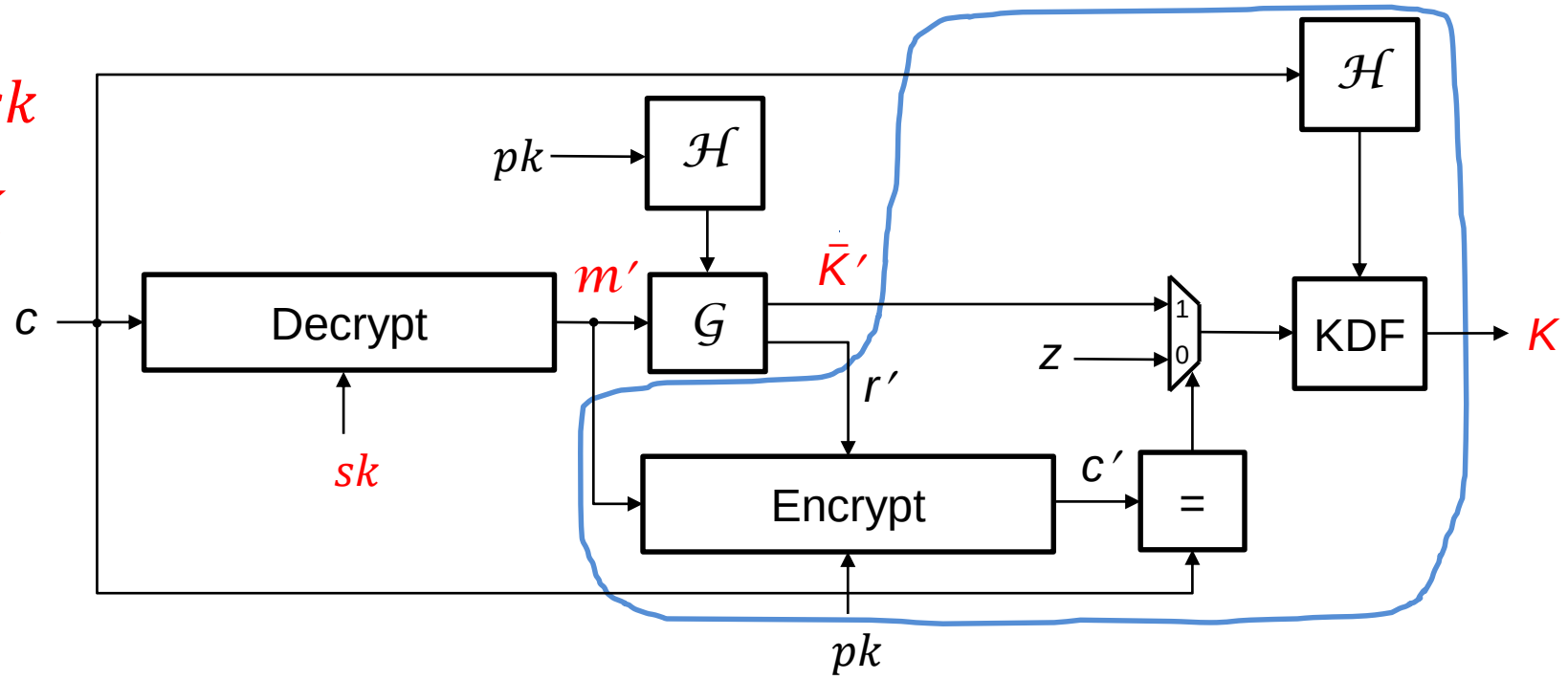
Output: ciphertext c, shared key *K*



Decapsulation algorithm

Input: c, sk

Output: K



A version of Fujisaki-Okamoto (FO) transform is used to create an IND-CCA2 secure KEM from an IND-CPA secure PKE

IND-CPA = Indistinguishability against chosen-plaintext attacks

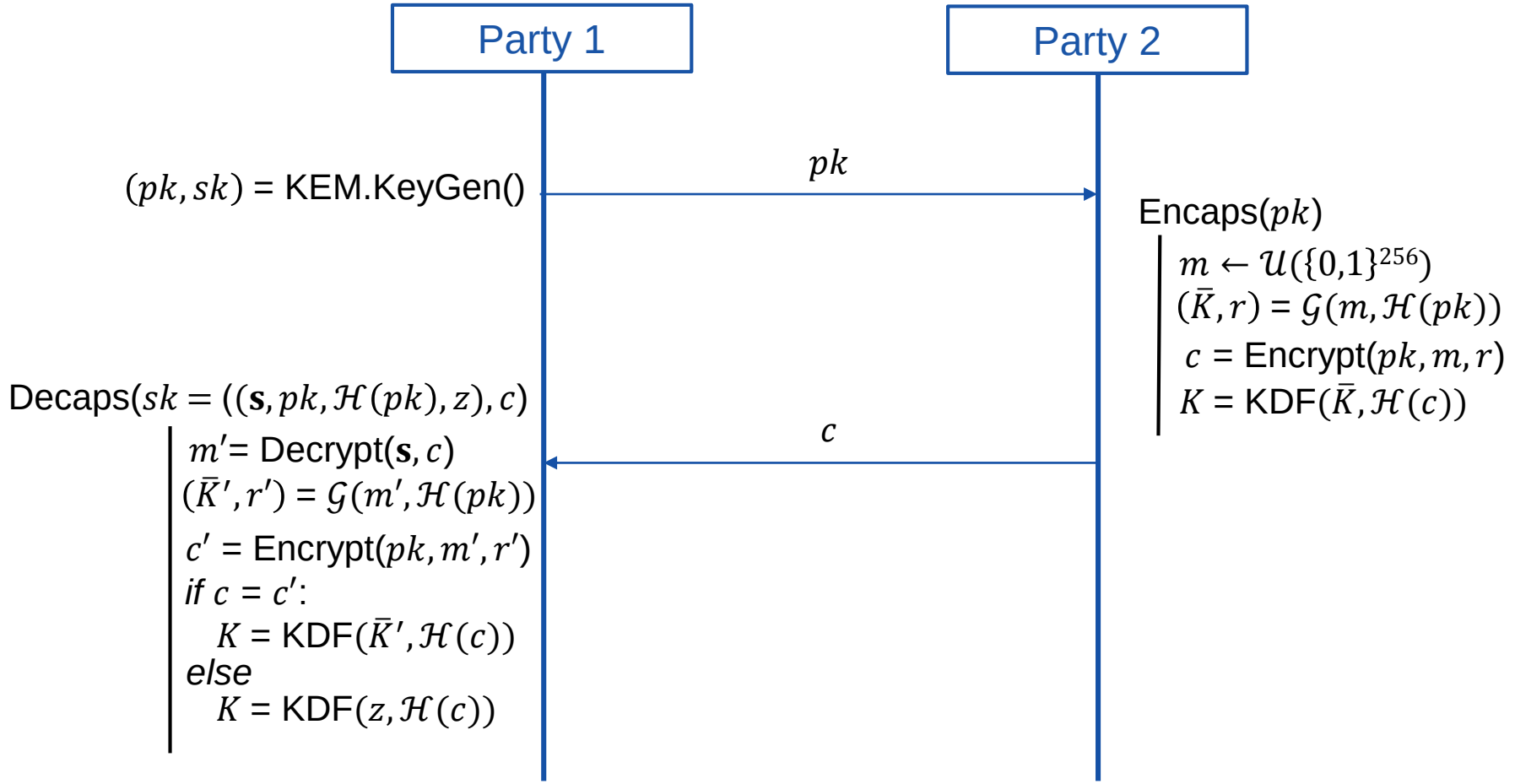
IND-CCA2 = Indistinguishability against adaptive chosen-ciphertext attacks

IND-CPA and IND-CCA2

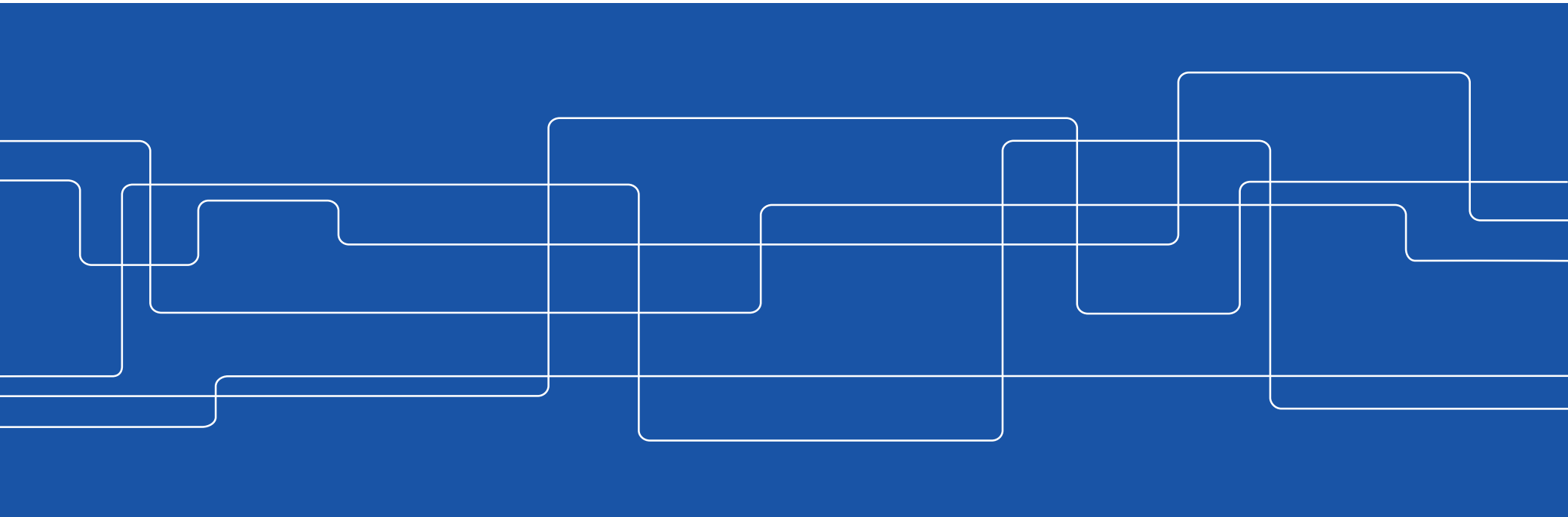
- IND-CPA means that one cannot distinguish two ciphertexts based on the messages they encrypt
  - Basic requirement for most provably secure PKE schemes

- IND-CCA2 means that one cannot improve the guess by allowing the use of a decryption oracle that can decrypt any ciphertexts except the given ones

Shared key establishment protocol



ML-DSA (CRYSTALS-Dilithium)



Module Lattice Digital Signature Scheme

Security is based on the hardness of LWE in module lattices and a version of the module short integer solution (M-SIS) problem

pk is public key

sk is secret key

σ is signature

m is message

- Key generation, (pk, sk) = KeyGen()
- Signing, σ = Sign(sk, m)
- Verification, Verify(pk, m, σ)

- Inputs and outputs to all API functions are byte arrays (as in Kyber)

  $\Rightarrow$ unpacking of the byte arrays into the polynomial coefficients and vice versa must be performed

ML-DSA parameters

Version	n	q	(k, ℓ)	η	d	γ_1	γ_2	β	ω
Dilithium-2	256	8380417	(4, 4)	2	13	2^{17}	$(q-1)/88$	78	80
Dilithium-3	256	8380417	(6, 5)	4	13	2^{19}	$(q-1)/32$	196	55
Dilithium-5	256	8380417	(8, 7)	2	13	2^{19}	$(q-1)/32$	120	75

prime $q = 2^{33} - 2^{13} + 1$

ML-DSA key generation algorithm (simplified)

Output: public key pk, secret key sk

$\text{KeyGen}()$

...

3: $\mathbf{A} \in R_q^{k \times \ell} = \text{ExpandA}(\rho)$

4: $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k = \text{ExpandS}(\rho')$

5: $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$

6: $(\mathbf{t}_1, \mathbf{t}_0) = \text{Power2Round}(\mathbf{t}, d)$

...

8: $pk = \text{pkEncode}(\rho, \mathbf{t}_1)$

9: $sk = \text{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

10: **return** (pk, sk)

Drops *d* low-order bits of each coefficient of **t**

Packing into byte arrays

b = As + e in ML-KEM

pk = (A, b)

sk = s

ML-DSA signing algorithm (simplified)

Input: secret key sk, message m

Output: signature σ

$\mathrm{Sign}(sk, m)$

1: $(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) = \mathsf{skDecode}(sk)$

2: $\mathbf{A} \in R_q^{k \times \ell} = \mathsf{ExpandA}(\rho)$

...

6: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**

...

12: $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$

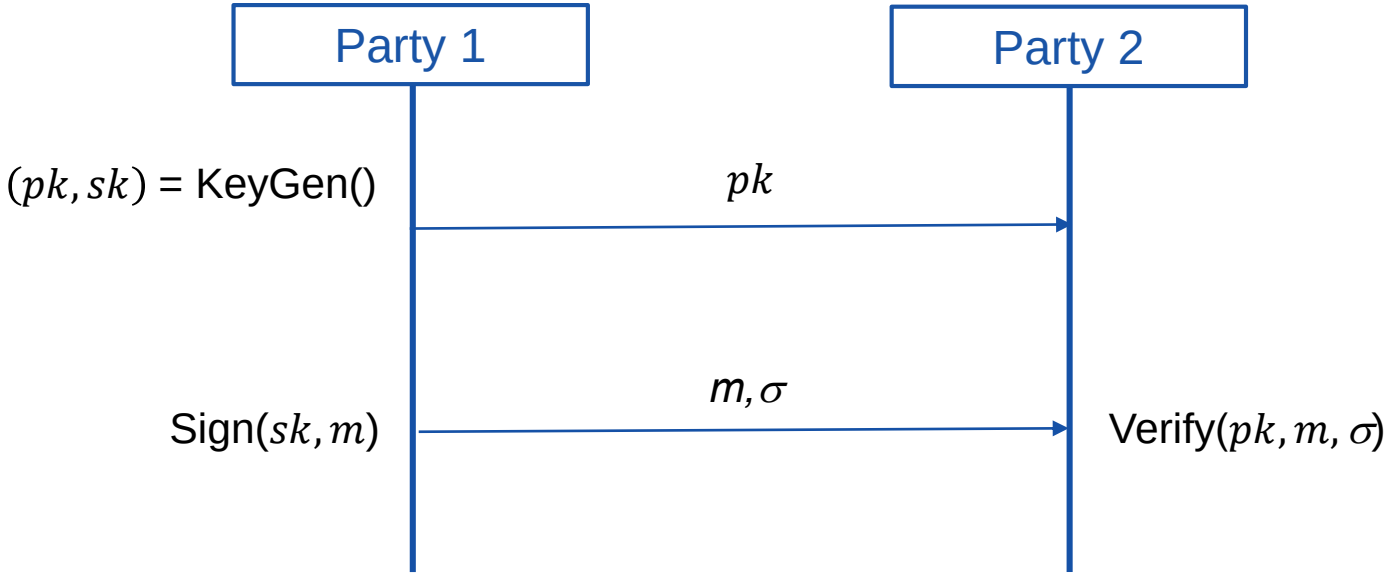
...

19: **return** $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

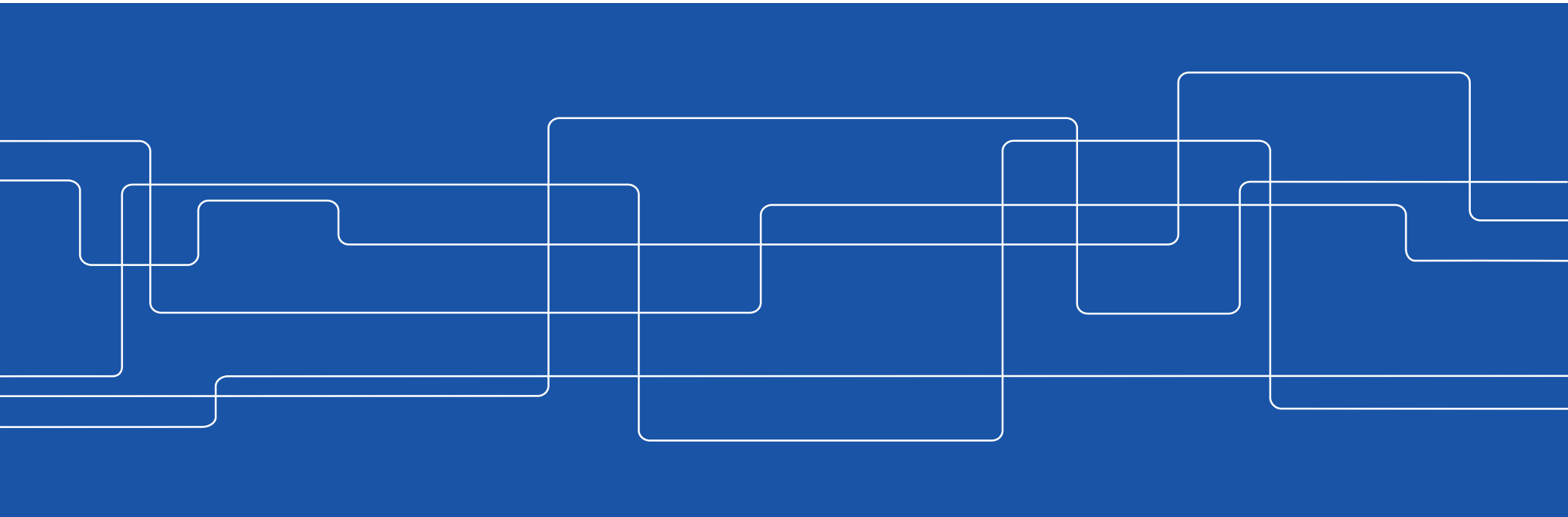
Unpacking of byte arrays

Many side-channel attacks target this point

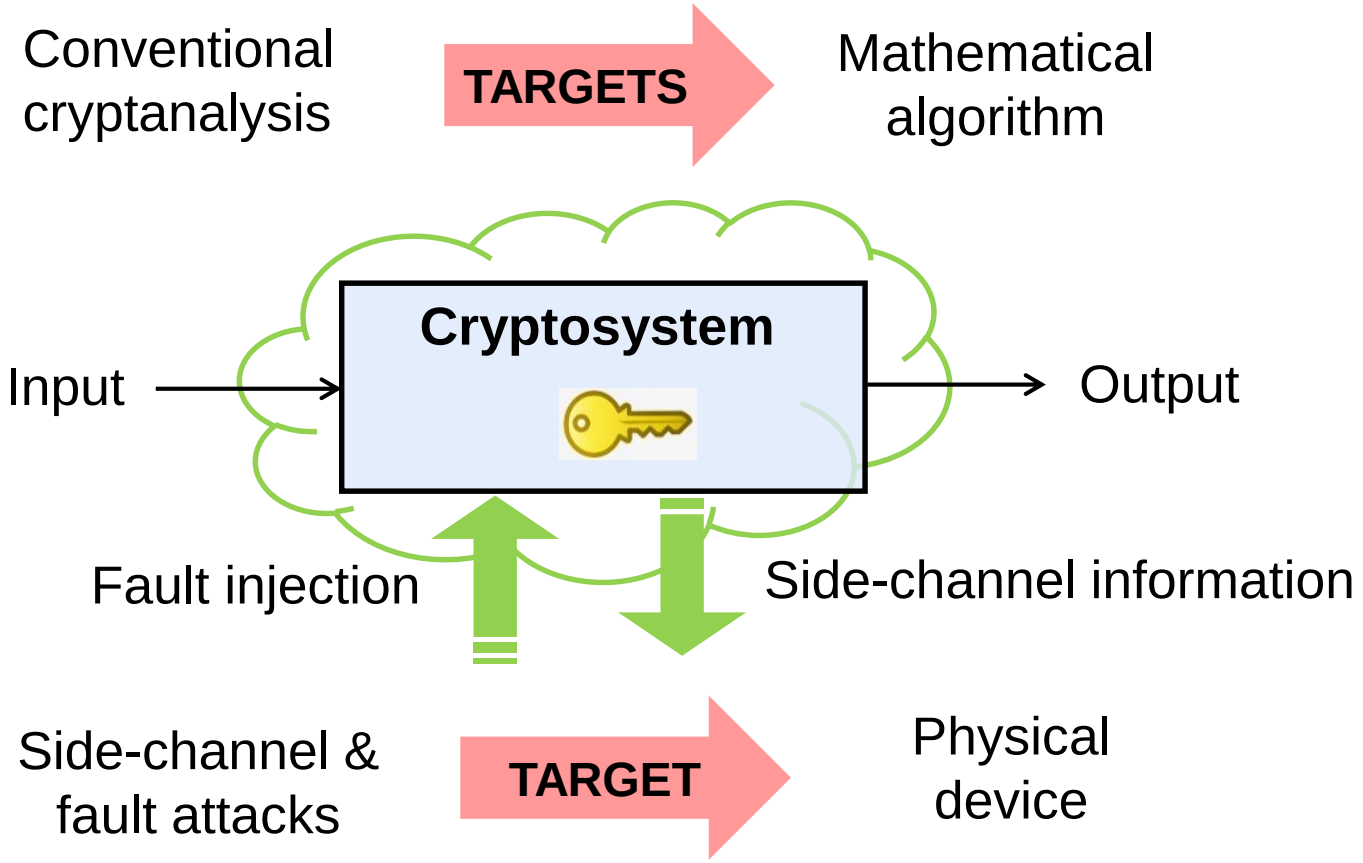
Authentication protocol



Side-channel and fault attacks

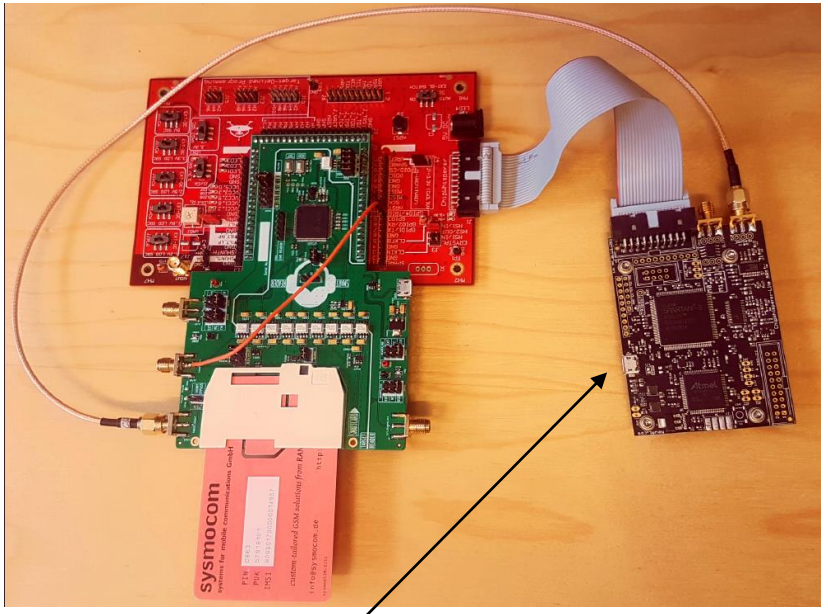


What is a physical attack?



Side-Channel Analysis (SCA)

- Algorithms are implemented in MCUs, CPUs, FPGAs, ASICs,…
- Different operations may consume different amount of power/time
- The same operation executed on different data may consume different amount of power/time
- It may be possible to recognize which operations and data are processed from power/time
  - ML techniques are useful



ChipWhisperer-Lite

photo credit: Martin Brisfors

Fault Injection (FI)

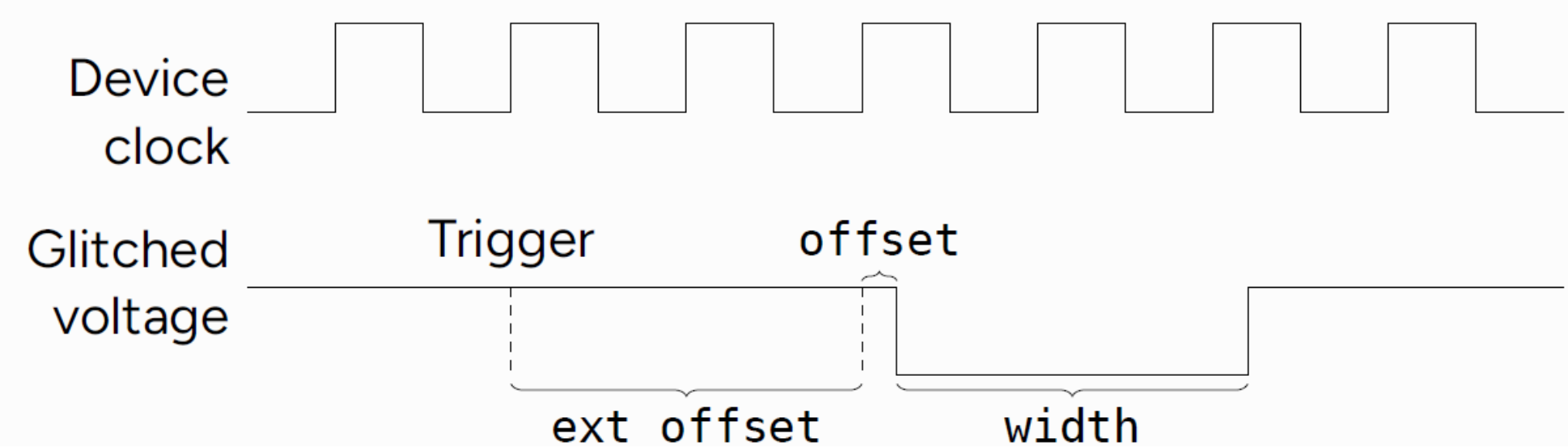
- Clock glitching:
  - Inject/withhold rising edge in clock signal
- Voltage glitching:
  - Short power supply

ChipWhisperer-Husky

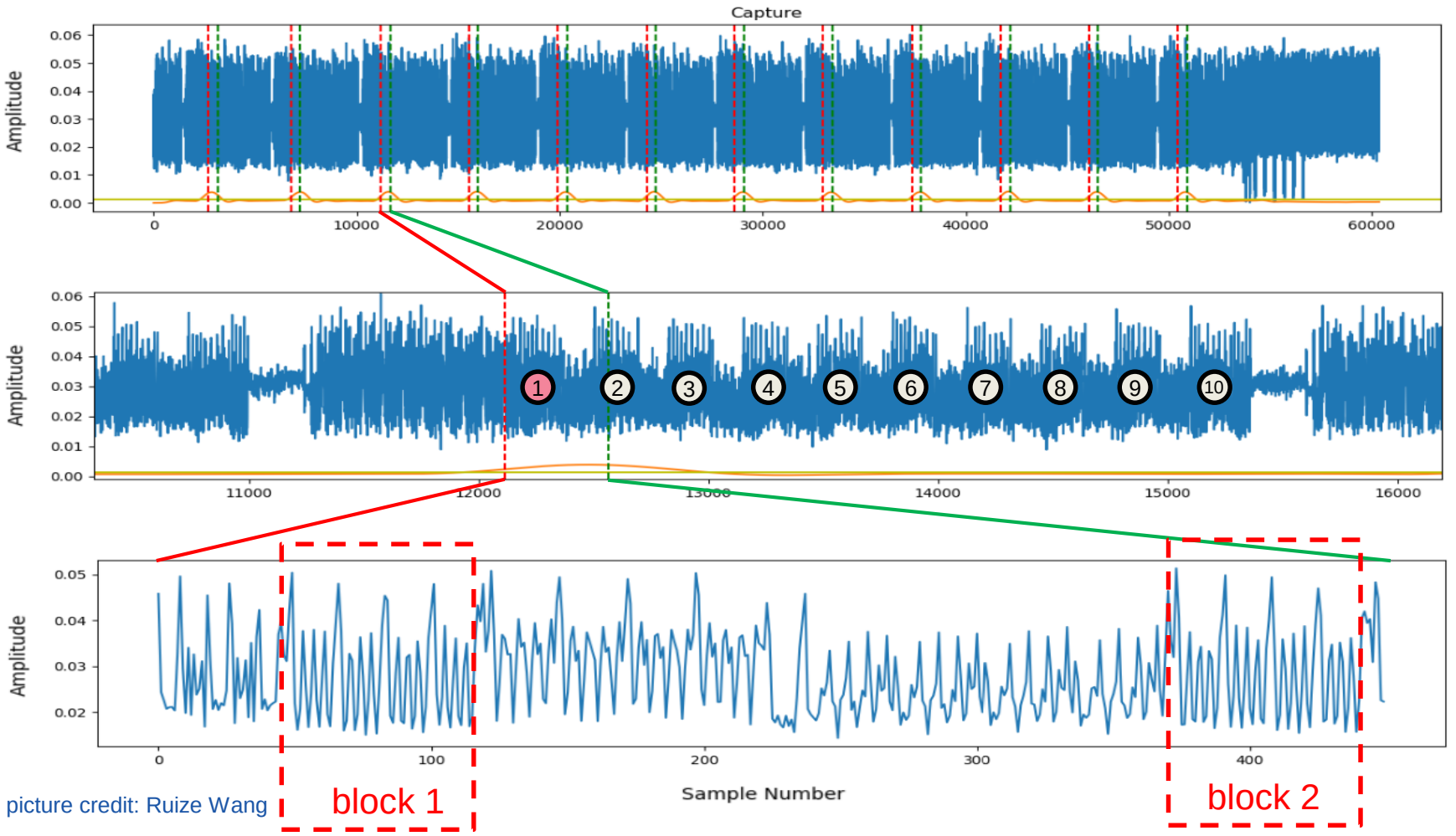
- Inexpensive ($550), easy to use
- Requires precise timing

photo credit: Sönke Jenral

Voltage glitching parameters



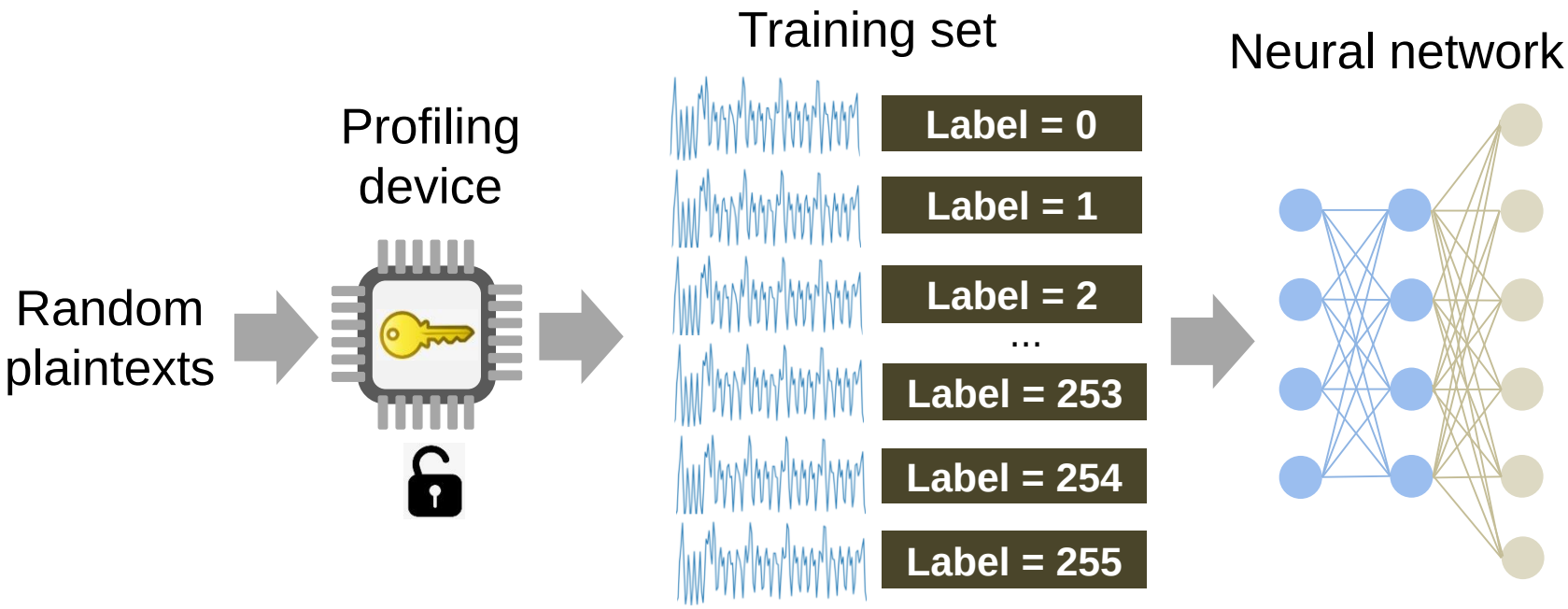
SCA example of AES-128 on 32-bit MCU



picture credit: Ruize Wang

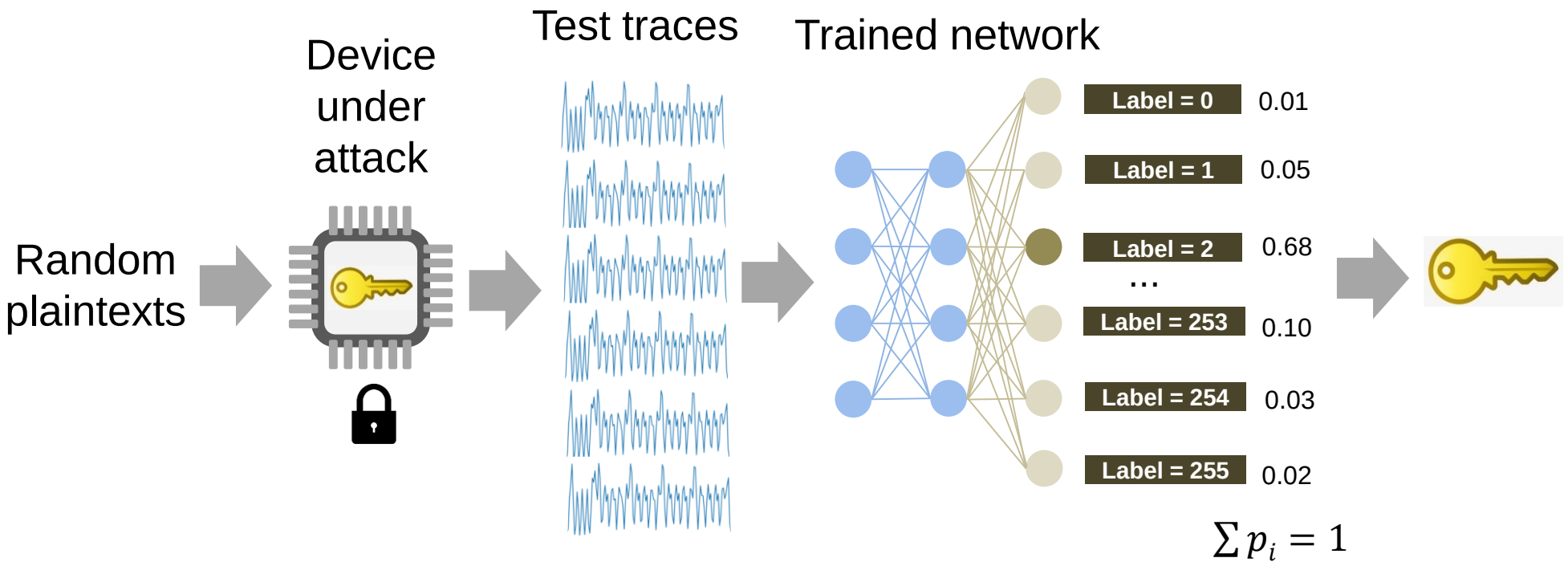
How Deep Learning (DL) helps?

Profiling stage: Train a neural network using traces from profiling devices

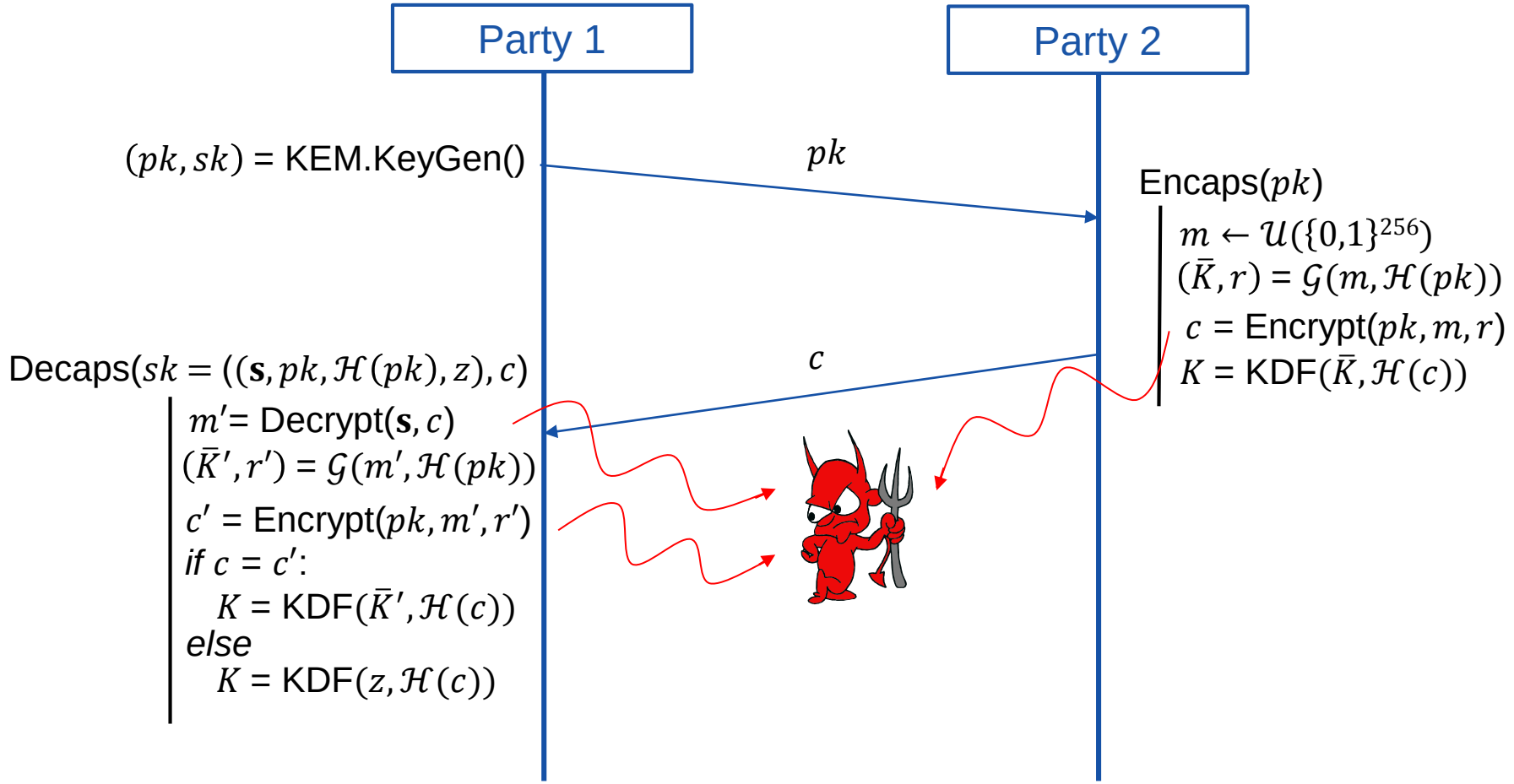


How Deep Learning (DL) helps?

Attack stage: Use the trained network to classify traces from the device under attack



Attack scenario for ML-KEM



Party 1

Party 2

(pk_p, sk_p) = KeyGen()
$m_p \leftarrow \mathcal{U}(\{0,1\}^{256})$
c_p= Encrypt(pk_p, m_p, r_p)
$T_p \rightsquigarrow$ Decaps(sk_p, c_p)
$\mathcal{M}$ = TrainModel(T_p, m_p)

If m is used as a label, profiling traces can be captured from the device under attack since pk is used to compute c

(pk, sk) = KeyGen()

pk

pk

(c, K) = Encaps(pk)

c

c

K = Decaps(sk, c)

T

$m' = \mathcal{M}(T)$
$(\bar{K}', r') = \mathcal{G}(m', \mathcal{H}(pk))$
K = KDF$(\bar{K}', \mathcal{H}(c))$

$c_1, c_2, \ldots$

K_1 = Decaps(sk, c_1)
K_2 = Decaps(sk, c_2)
…

$T_1, T_2, \ldots$

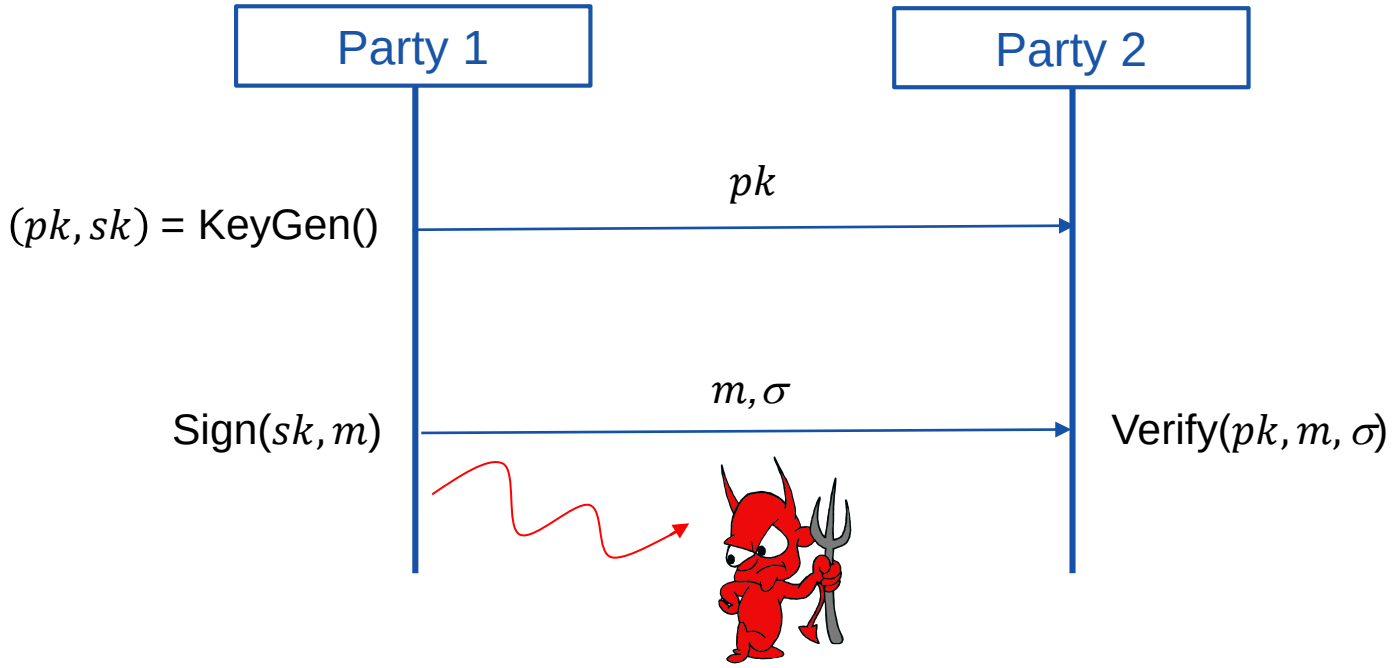
$m_1 = \mathcal{M}(T_1)$
$m_2 = \mathcal{M}(T_2)$
…
sk = RecoverKey$(m_1, m_2, \ldots)$

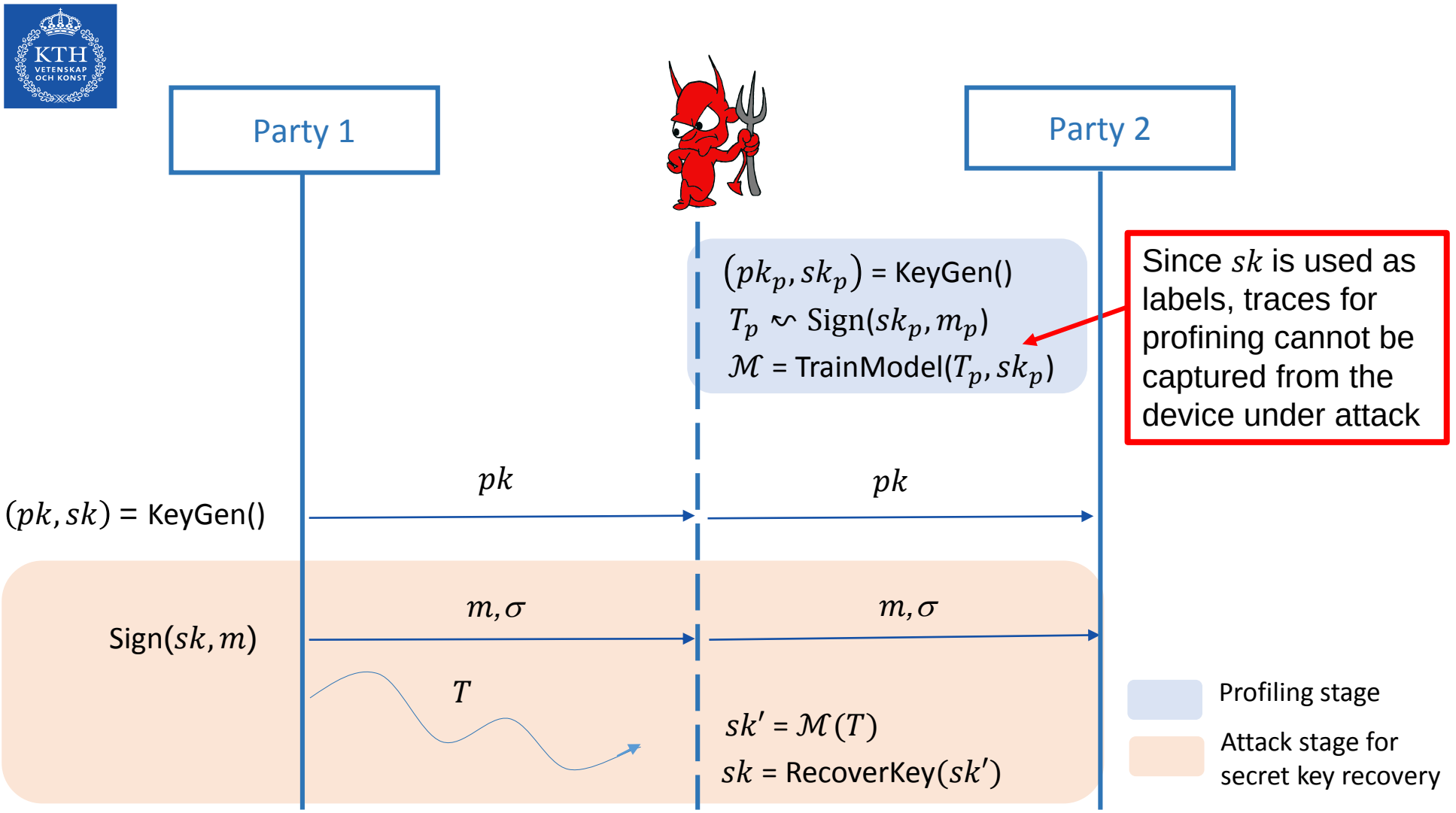
Profiling stage

Attack stage for shared key recovery

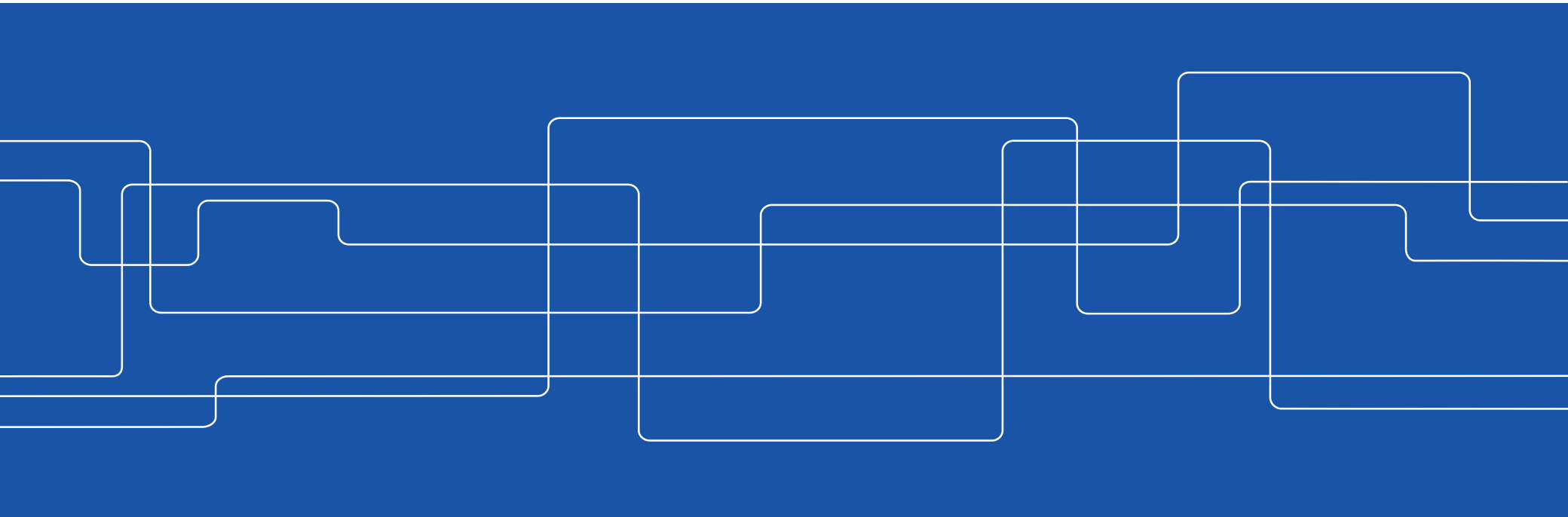
Attack stage for secret key recovery

Attack scenario for ML-DSA

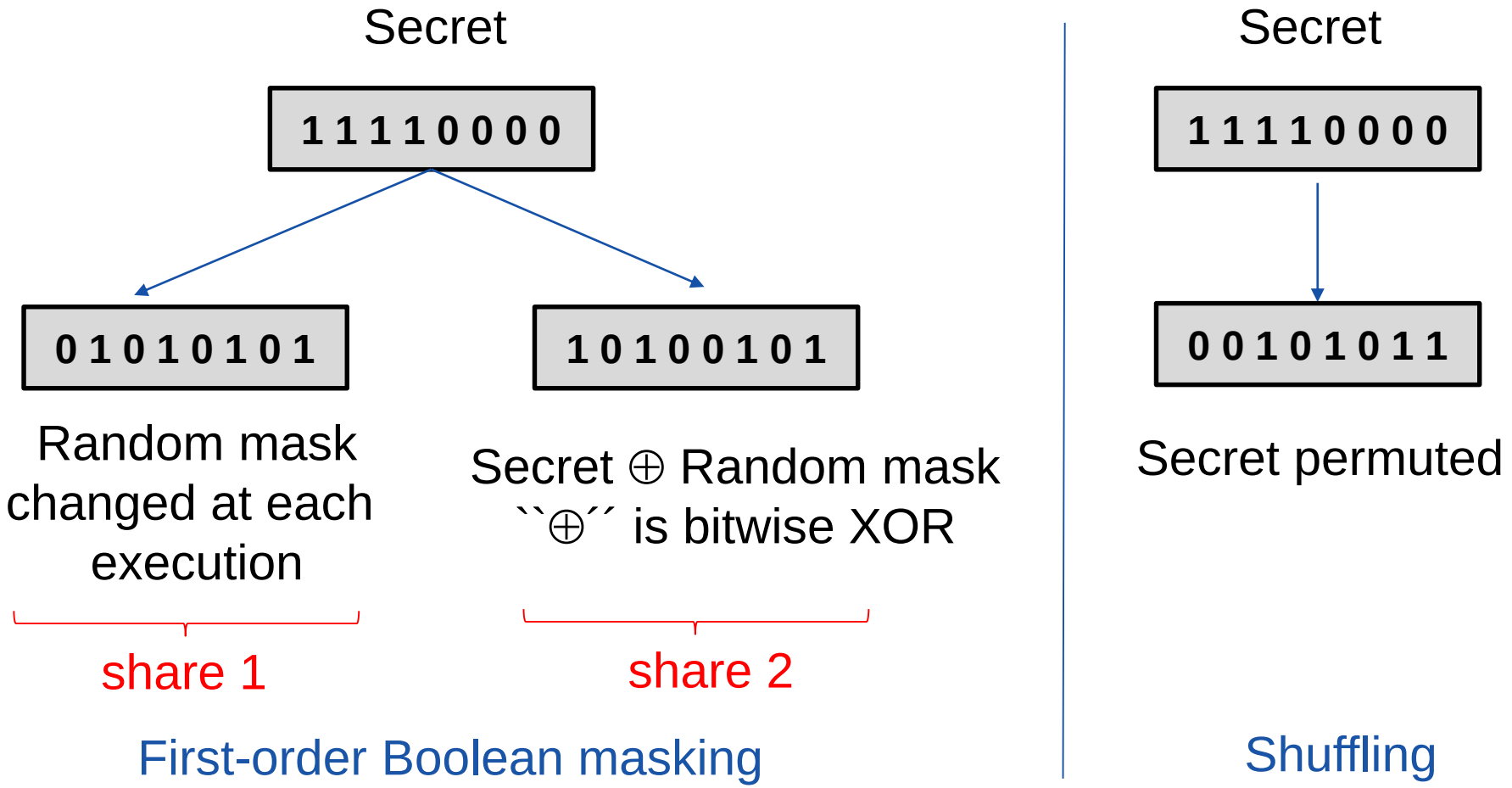



Party 1
Party 2
(pk_p, sk_p) = KeyGen()
$T_p \rightsquigarrow \text{Sign}(sk_p, m_p)$
$\mathcal{M}$ = TrainModel(T_p, sk_p)
Since sk is used as labels, traces for profining cannot be captured from the device under attack
(pk, sk) = KeyGen()
pk
pk
Sign(sk, m)
m, σ
m, σ
T
$sk' = \mathcal{M}(T)$
sk = RecoverKey(sk')
Profiling stage
Attack stage for secret key recovery

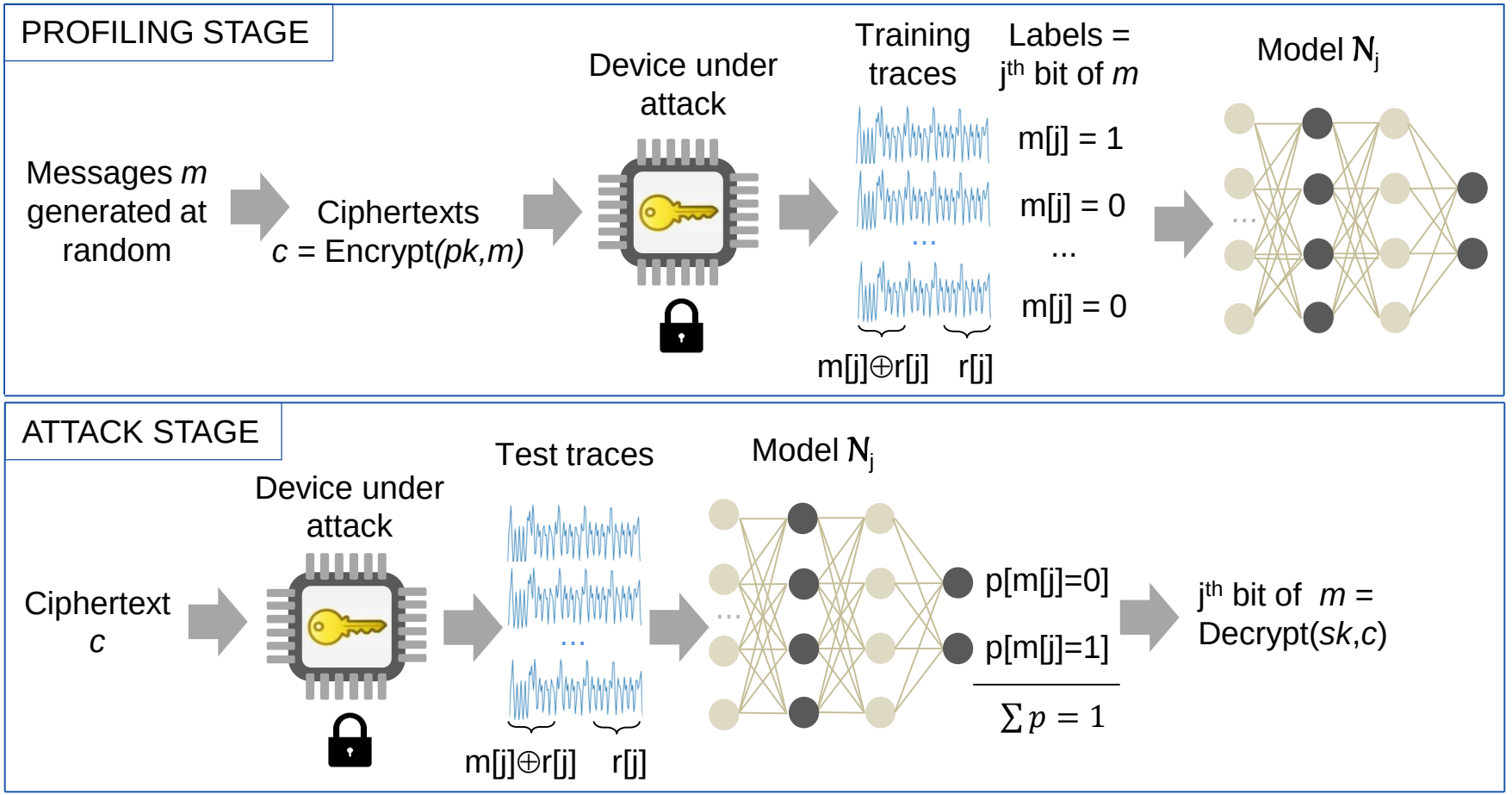
Countermeasures



Masking and shuffling countermeasures

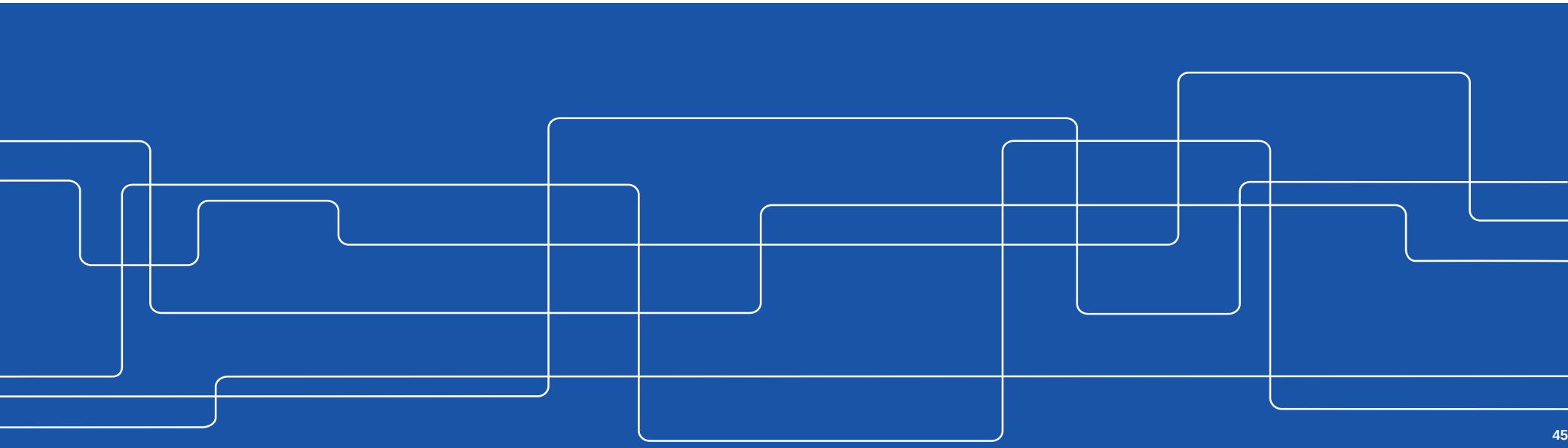


How DL helps break masking



A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM Implementation,
K. Ngo, E. Dubrova, Q. Guo, T. Johansson, TCHES'2021(4), 676-707

Attacks on ML-KEM and ML-DSA

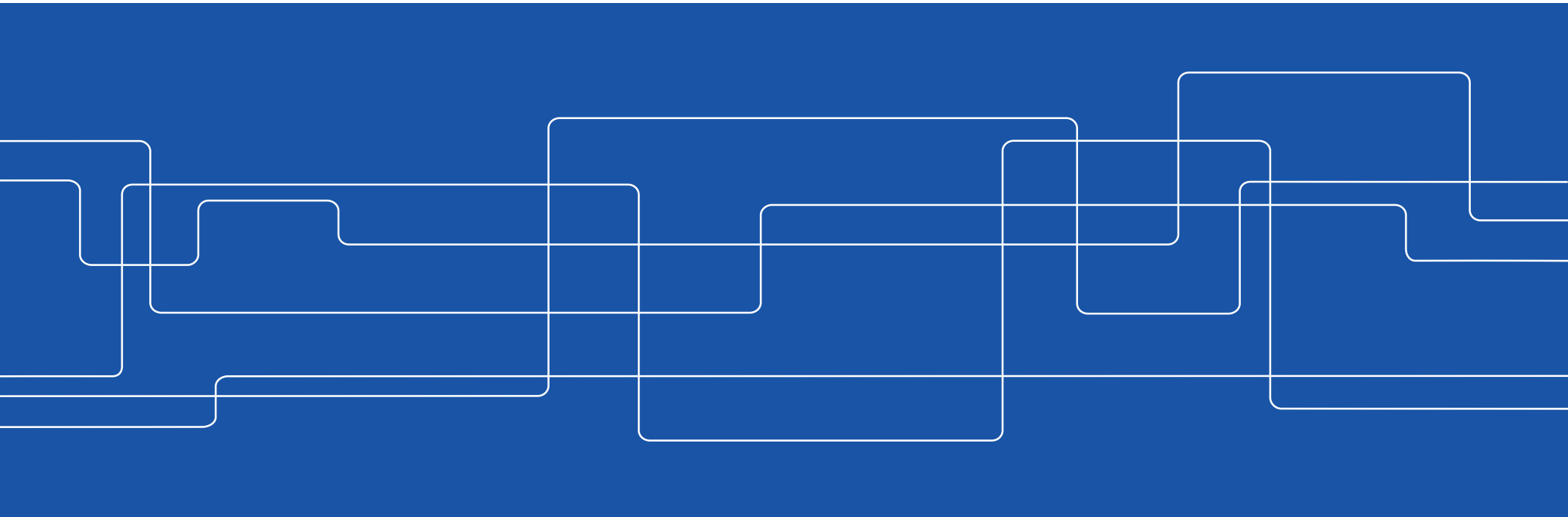


Six attacks on software implementations

- ML-KEM:
  - DL-SCA on an unmasked implementation
  - DL-SCA on a first-order masked & shuffled implementation
  - FI attack on a first-order masked & shuffled implementation
  - DL-SCA on a higher-order masked implementation
- ML-DSA
  - DL-SCA on an unmasked implementation
  - FI attack on an unmasked implementation

Secret key recovery attack on unmasked Kyber using *k* chosen ciphertexts

A Side-Channel Secret Key Recovery Attack on CRYSTALS-Kyber Using *k* Chosen Ciphertexts, R. Wang, E. Dubrova, *C2SI'2023*



Attack details

- Kyber-768 C implementation:
  - Kannwischer, M.J., Petri, R., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, https://github.com/mupq/pqm4
  - Complied with optimization level -O3
- Attack point:
  - Decryption step of decapsulation
    - message decoding and polynomial reduction procedures
- Target board:
  - ARM Cortex-M4 in CW308TSTM32F4

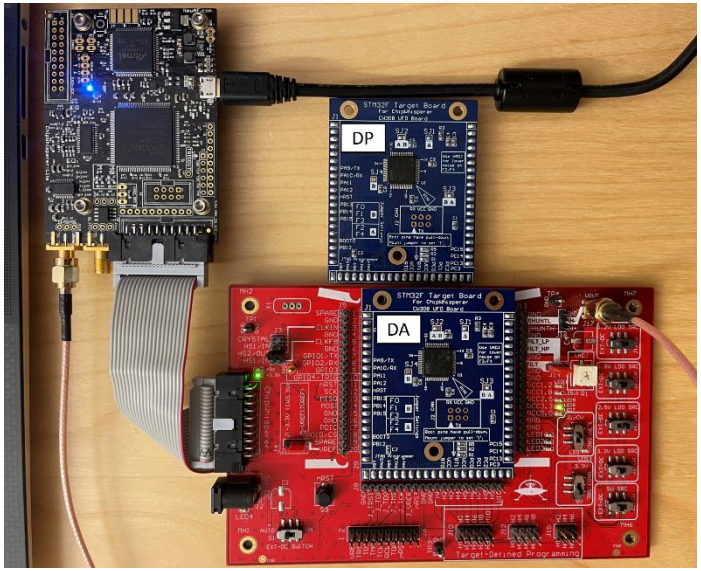


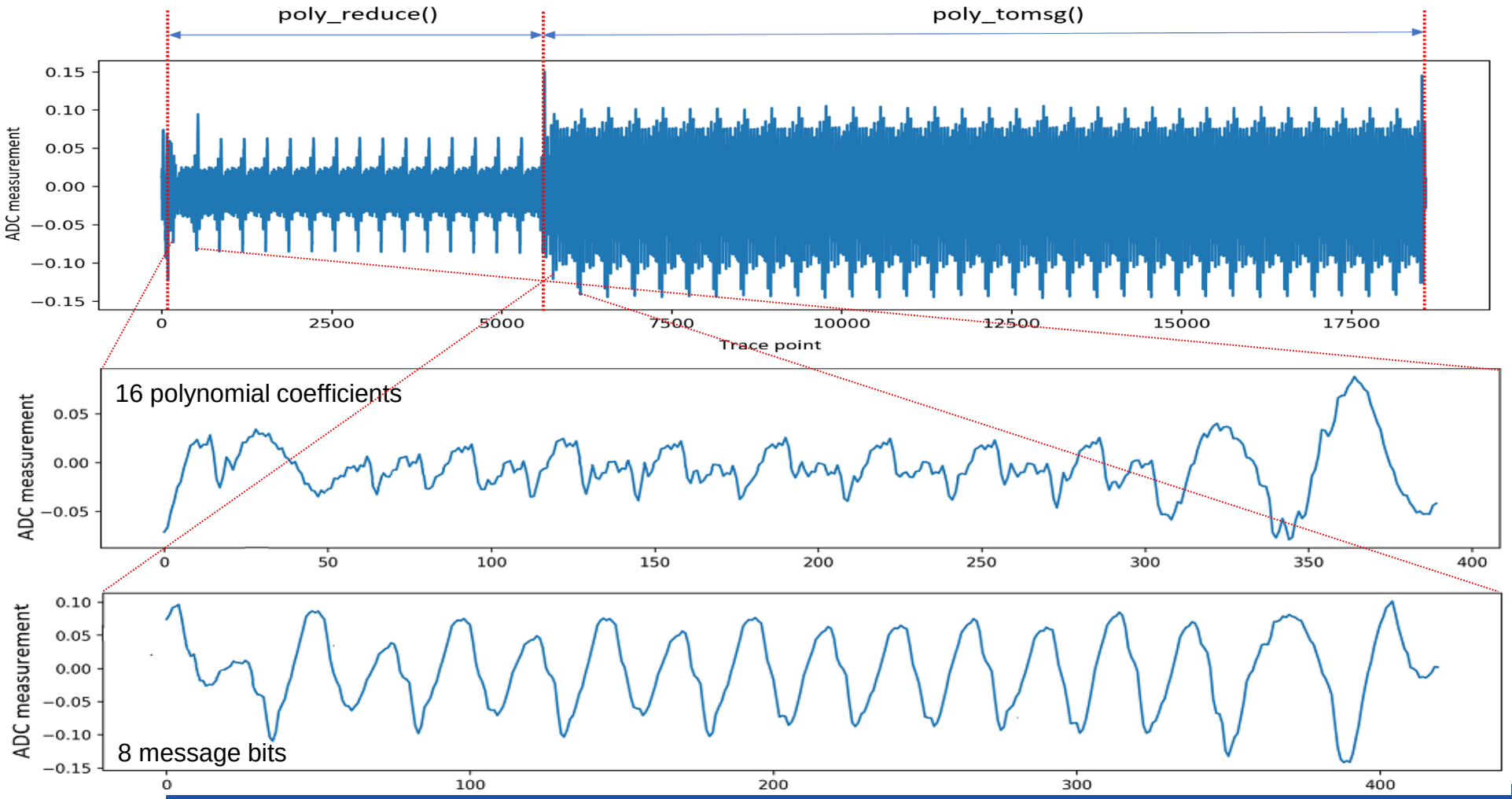
photo credit: Ruize Wang

Message decoding and polynomial reduction procedures

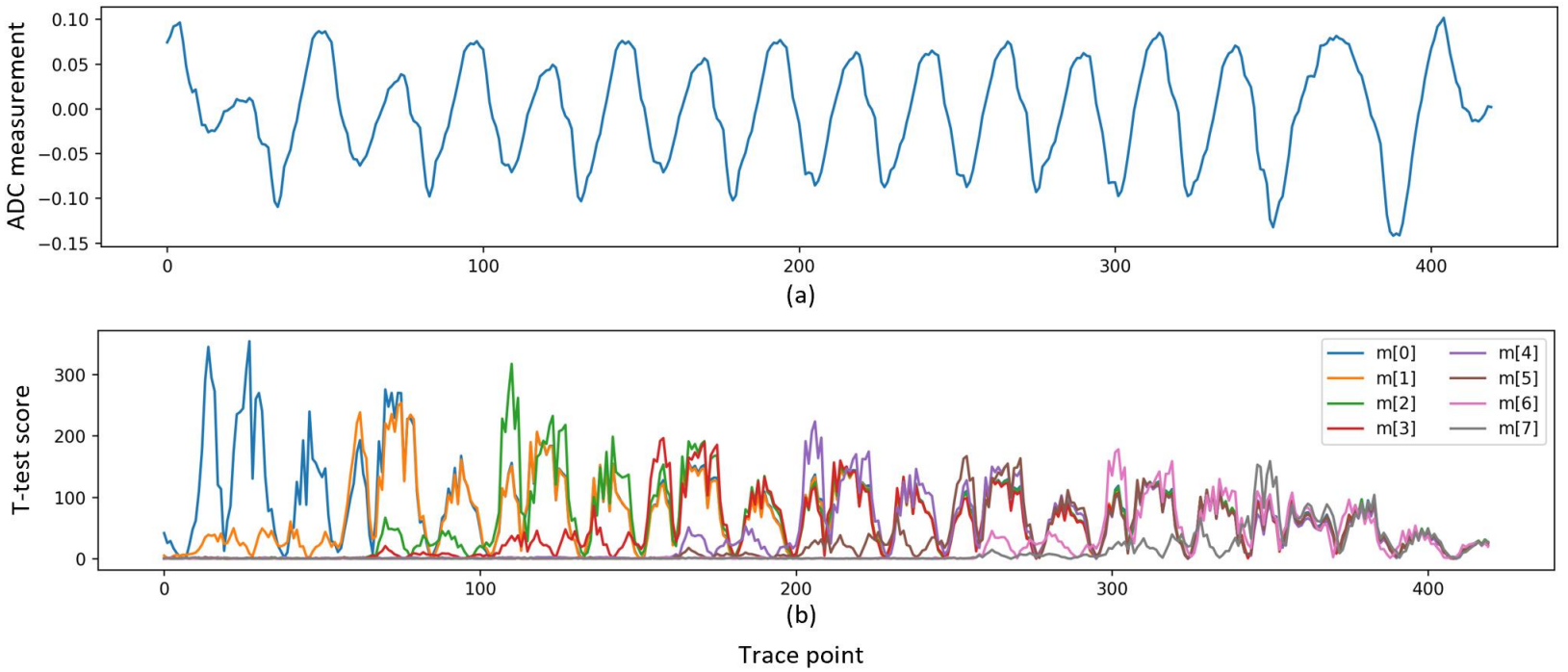
```
void poly_tomsg(char *msg, poly *a)
 1: for (i = 0; i < BYTES; i++) do
 2:    msg[i] = 0;
 3:    for (j = 0; j < 8; j++) do
 4:       t=(((a->coeffs[8*i+j]<<1) + KYBER_Q/2)/KYBER_Q)&1;
 5:       msg[i] |= t<<j;
 6:    end for
 7: end for
```

```
void poly_reduce(poly *r)
 1: asm_barrett_reduce(r->coeffs); /*In assembly*/
```

Power trace of poly_reduce() and poly_tomsg()



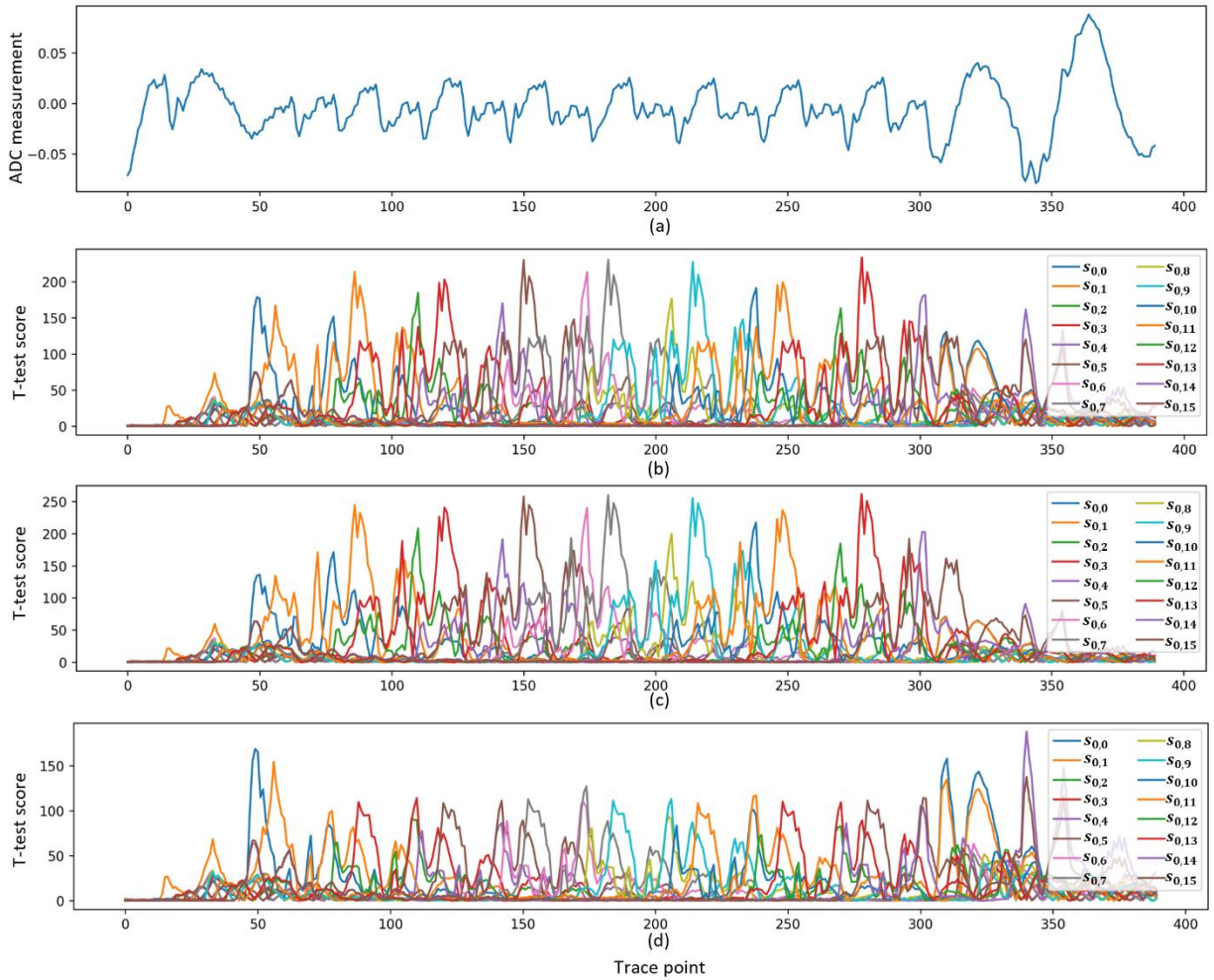
Leakage analysis of poly_ tomsg()



m[i] = 0
vs
m[i] = 1

(a) Power trace of poly_tomsg(); (b) T-test for 8 message bits (100K traces)

Leakage analysis of poly_ reduce()



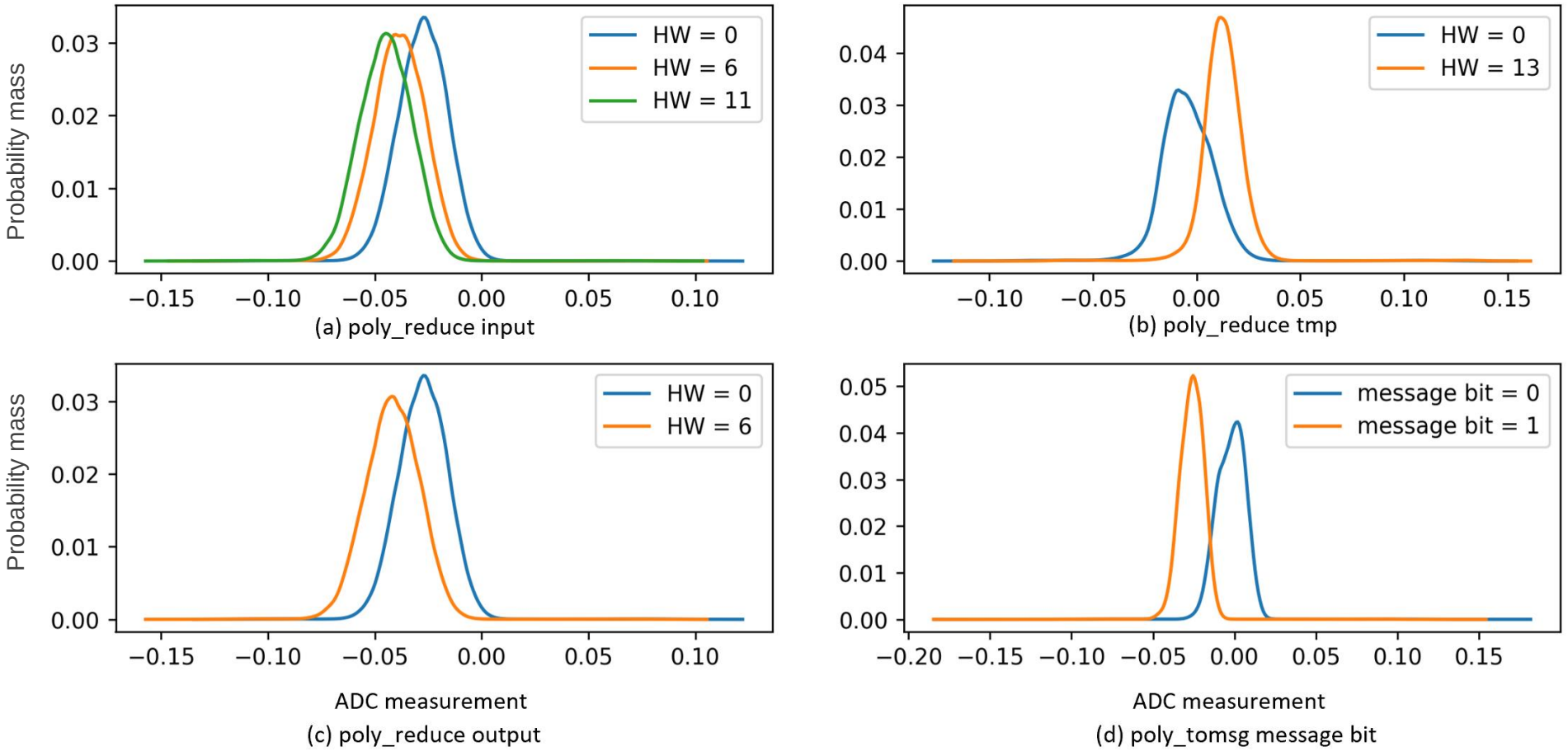
HW(input) = 0 vs
HW(input) = 11

HW(tmp) = 0 vs
HW(tmp) = 13

HW(output) = 0 vs
HW(output) = 6

(a) Power trace of poly_reduce(): (b,c,d) T-test for 16 poly. coeff. (100K traces)

Distributions of power consumption for different intermediate variables



Chosen ciphertext construction method

The secret key $\boldsymbol{s}$ consists three polynomials $\boldsymbol{s} = (s_0, s_1, s_2)$

The ciphertext $c = (\boldsymbol{u}, v)$ consists of $\boldsymbol{u} = (u_0, u_1, u_2)$ and v

To recover *n* coefficients of s_i, we set:

$$\boldsymbol{u} = \begin{cases} (k_1, 0, 0) \in R_q^{3\times 1} & \text{for } i = 0 \\ (0, k_1, 0) \in R_q^{3\times 1} & \text{for } i = 1 \\ (0, 0, k_1) \in R_q^{3\times 1} & \text{for } i = 2 \end{cases} \qquad v = k_0 \sum_{j=0}^{255} x^j \in R_q^{1\times 1}$$

Then, for $i \in \{0,1,2\}$, $m[j]$ is a function of the tuple $(k_0, k_1, si[j])$:

$$v - \boldsymbol{s}^T \boldsymbol{u} = \sum_{j=0}^{255} (k_0 - k_1 s_i[j])\, xj$$

Chosen ciphertext construction, cont.

Search through all possible $k_0, k_1 \in \mathbb{Z}_q$ to construct $c = (\boldsymbol{u}, v)$ such that:

1. The number of different HWs of poly_reduce() intermediate variables is minimized
2. Hamming distances between poly_reduce() intermediate variables are maximized
3. There are both 0 and 1 message bits in poly_tomsg()
4. Given 1-3, all five key coefficients are uniquely defined

Mapping of intermediate values into secret key coefficients for $k_0 = 0$ and $k_1 = 1369$

Procedure	Variable	-2	-1	0	1	2
`poly_reduce`	input	-591 (11)	1369 (6)	0 (0)	-1369 (11)	591 (6)
	tmp	-3329 (13)	0 (0)	0 (0)	-3329 (13)	0 (0)
	output	2738 (6)	1369 (6)	0 (0)	1960 (6)	591 (6)
`poly_tomsg`	message bit	0	1	0	1	0

Mapping of MLP's labels into secret key coefficients for $k_0 = 0$ and $k_1 = 1369$

Procedure	MLP models	-2	-1	0	1	2
poly_reduce	$\{\mathcal{M}_0^{in}, \cdots, \mathcal{M}_{15}^{in}\}$	1	2	0	1	2
	$\{\mathcal{M}_0^{tmp}, \cdots, \mathcal{M}_{15}^{tmp}\}$	1	0	0	1	0
	$\{\mathcal{M}_0^{out}, \cdots, \mathcal{M}_{15}^{out}\}$	1	1	0	1	1
poly_tomsg	$\{\mathcal{M}_0^{p2m}, \cdots, \mathcal{M}_7^{p2m}\}$	0	1	0	1	0

Empirical secret key recovery results (mean for 100 different secret keys)

$N \times k$	For profiling device			For device under attack		
	Single coefficient	Full key	Max # enum.	Single coefficient	Full key	Max # enum.
1×3	0.9990	0.47	5^8	0.9940	0.02	5^{16}
10×3	0.9997	0.81	5^3	0.9990	0.43	5^4
20×3	0.9997	0.81	5^3	0.9991	0.45	5^3
50×3	0.9997	0.81	5^2	0.9994	0.53	5^2
100×3	0.9997	0.81	5^2	0.9994	0.53	5^2

Number of chosen ciphertexts required for Kyber-768 secret key recovery

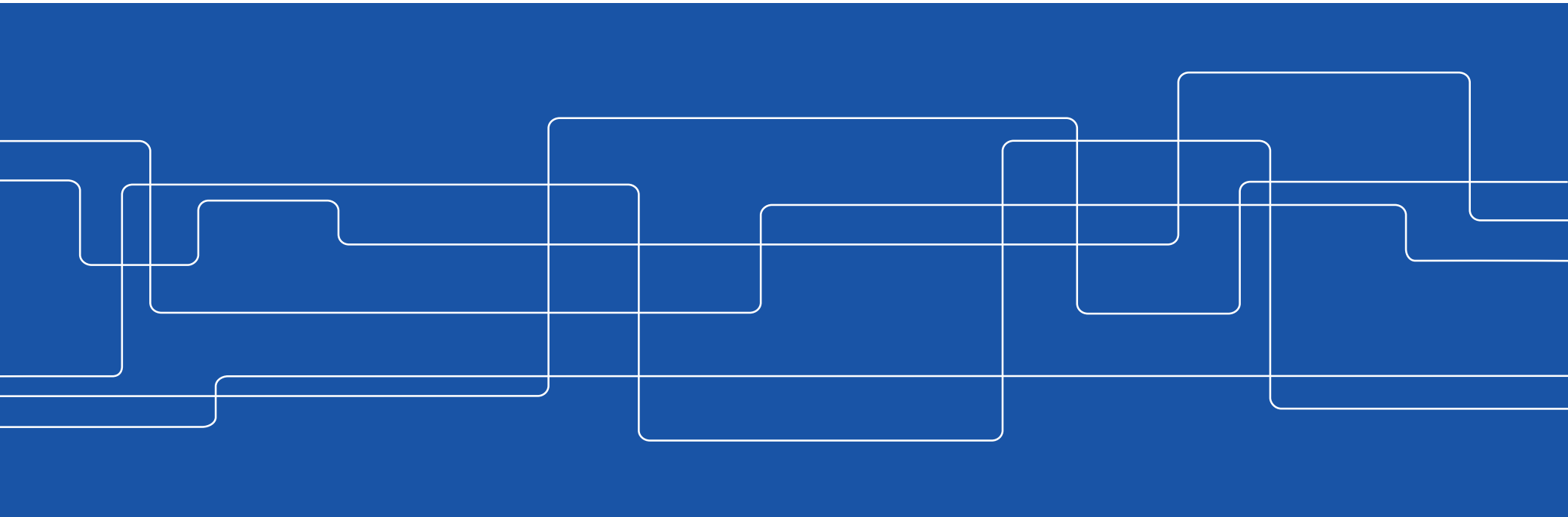
Paper	Attack target	Attack method	#CCT	Detect errors	Protected implem.	Real
Xu et al. [32]	Message encoding	Template	4×3	No	No	Yes
Ravi et al. [21]	Message decoding		3×3	No	No	Yes
Mu et al. [16]	Barrett reduction		11	No	No	Yes
Sim et al. [26]		Clustering	3×3	No	No	Yes
Hamburg et al. [10]	NTT	Template	$1 \times 3^{*}$	No	Yes	No
Backlund et al. [3]	Message decoding	MLP	$4 \times 3^{**}$	Yes	Yes	Yes
This work	Message decoding & Barrett reduction		1×3	Yes	No	Yes

*For noise tolerance level $\sigma \leq 1.2$ in the Hamming weight (HW) leakage.
**If a linear code with the code distance two is used for constructing CCT.

Secret key recovery attacks on a masked and shuffled ML-KEM

Secret Key Recovery Attack on Masked and Shuffled Implementations of CRYSTALS-Kyber and Saber, L. Backlund, K. Ngo, J. Gärtner, E. Dubrova, in *ACNS Workshops, 2023*

Breaking SCA-Protected CRYSTALS-Kyber with a Single Trace, S. Jendral, K. Ngo, R. Wang, E. Dubrova, HOST'2024



Attack details

- Kyber-768 C implementation (with shuffling added):
  - Heinz, D.,Kannwischer, M.J., Land, G., Pöppelmann, T., Schwabe, P., Sprenkels: First-order masked Kyber on ARM Cortex-M4. Cryptology ePrint Archive, Report 2022/058
  - Complied with optimization level -O3
- Attack point:
  - Decryption step of decapsulation
    - message decoding
    - Fisher-Yates index generation and usage
- Target board:
  - ARM Cortex-M4 in CW308TSTM32F4

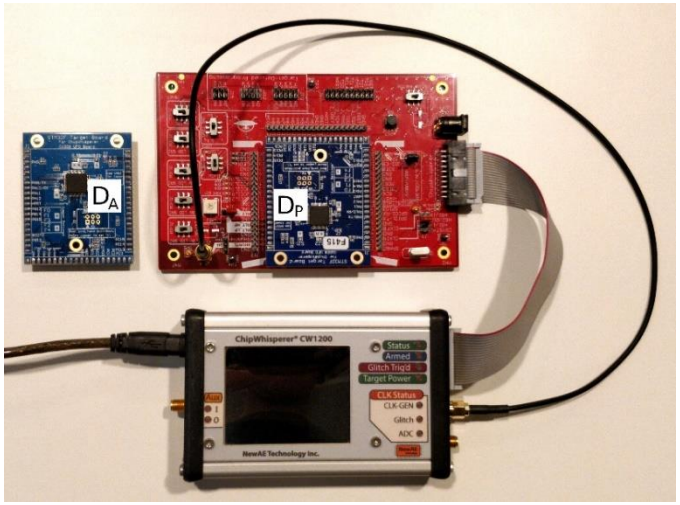
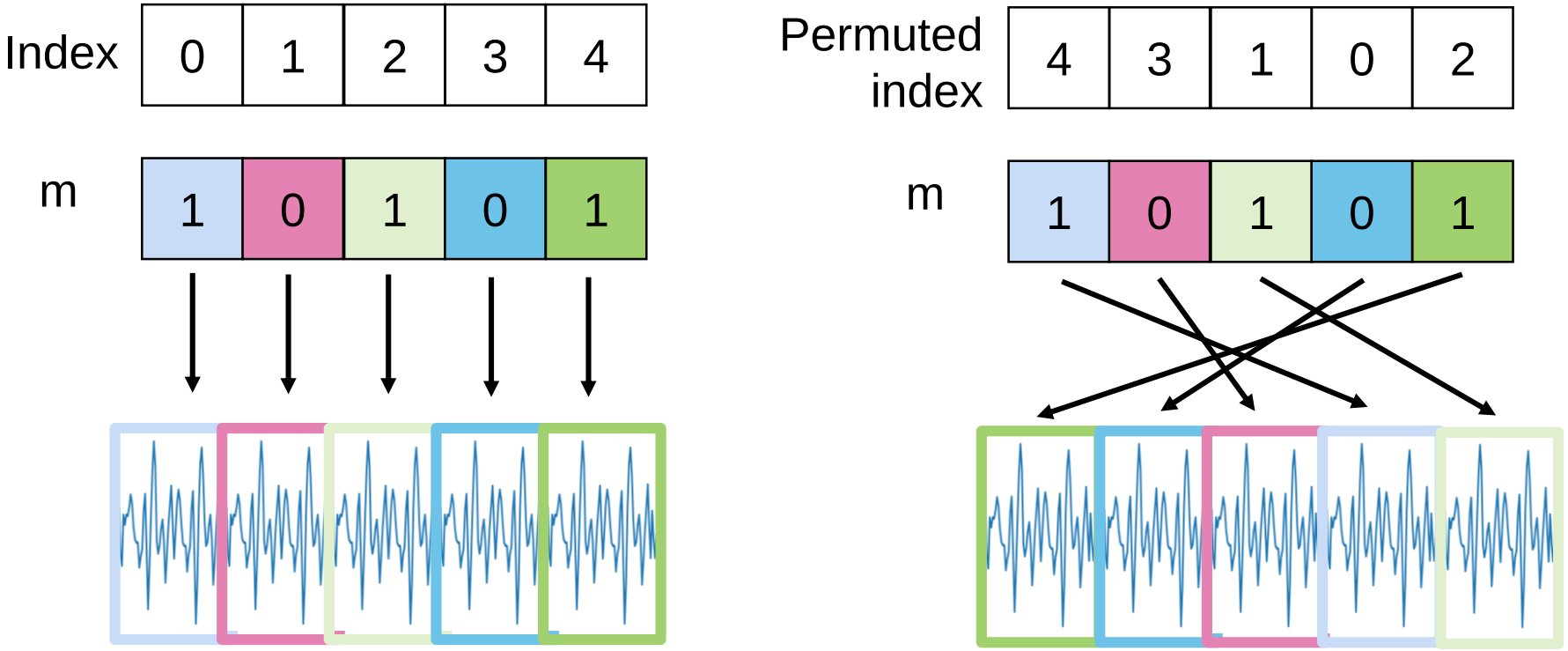


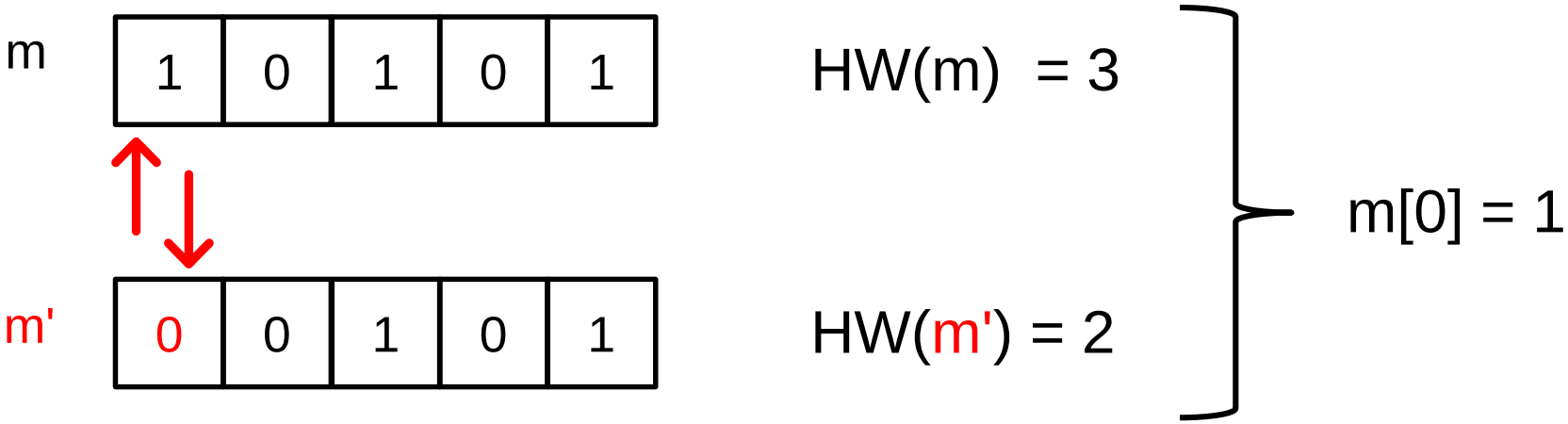
photo credit: Linus Backlund

Message recovery for a shuffled implementation



Individual message bits can be recovered, but their order is unknown

Previous approaches based on bit flipping



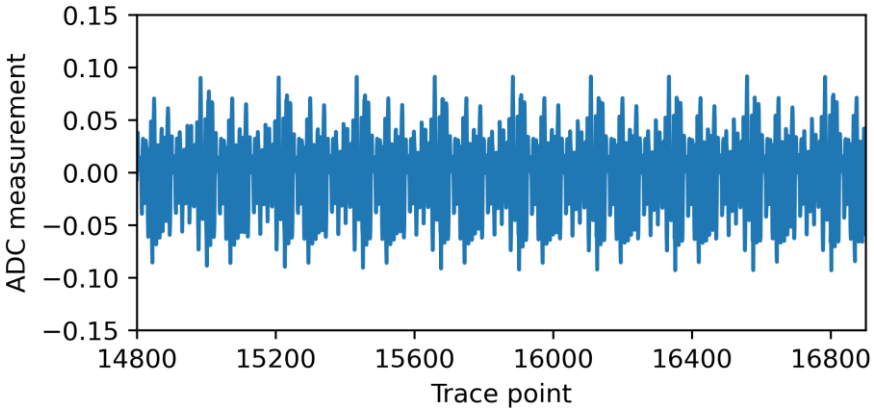
Ravi, P., Bhasin, S., Roy, S., Chattopadhyay, A., On exploiting message leakage in (few) NIST PQC candidates for practical message recovery and key recovery attacks, https://eprint.iacr.org/2020/1559.pdf

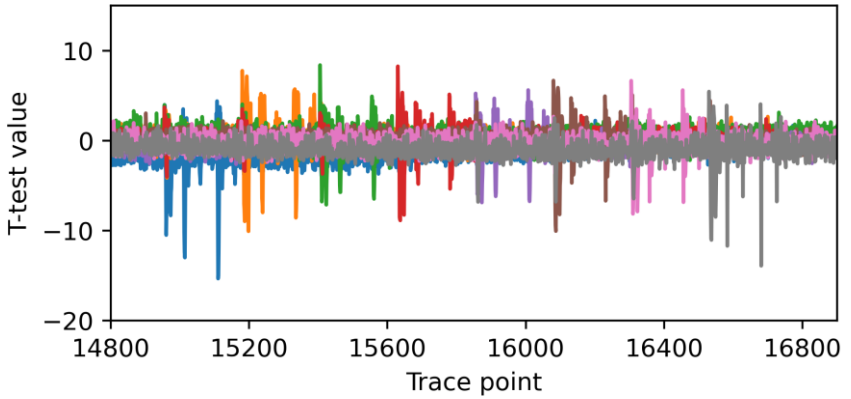
Ngo, K., Dubrova, E., Johansson, T., Breaking masked and shuffled CCA secure Saber KEM by power analysis, ASHES'2021

Fisher-Yates (FY) index usage

```
void masked_poly_tomsg(uint8 msg[2][32],
uint16 poly[2][256])
uint16 c[2];
uint8 permutation[256];
```

```
1: FY_Gen(permutation, 256);
2: for (x = 0; x < 256; x++) do
3:    x_rand = permutation[x];
4:    i = x_rand / 8;
5:    j = x_rand % 8;
6:    ... Processing ...
7:    msg[0][i] += ((c[0] >> 15) & 1)<<j;
8:    msg[1][i] += ((c[1] >> 15) & 1)<<j;
9: end for
```

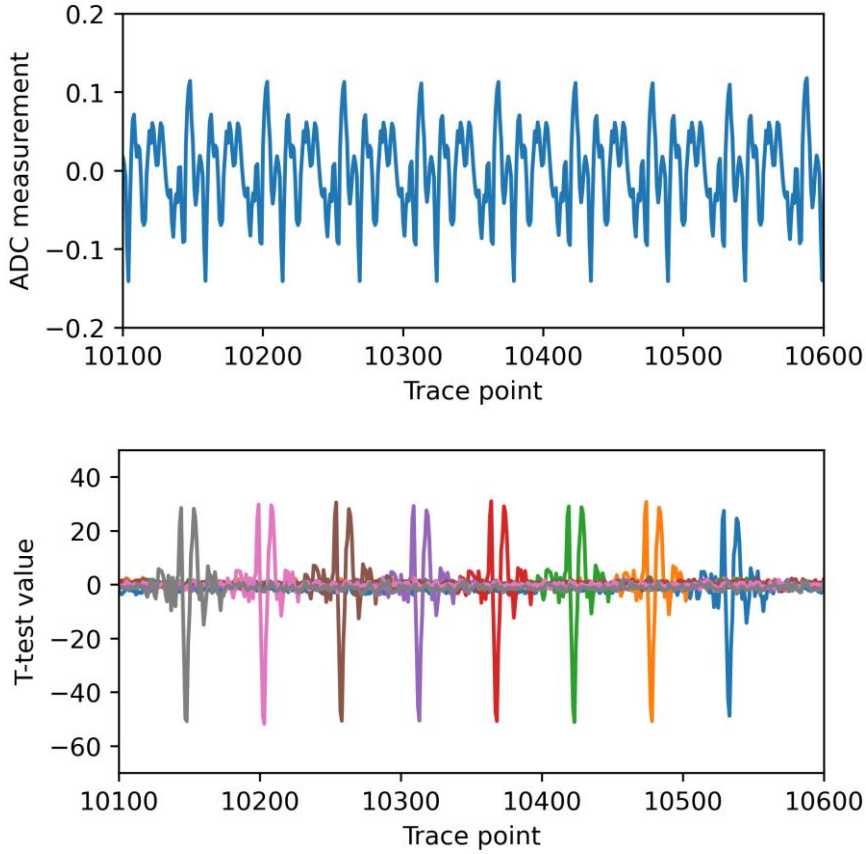




T-test on 5K traces

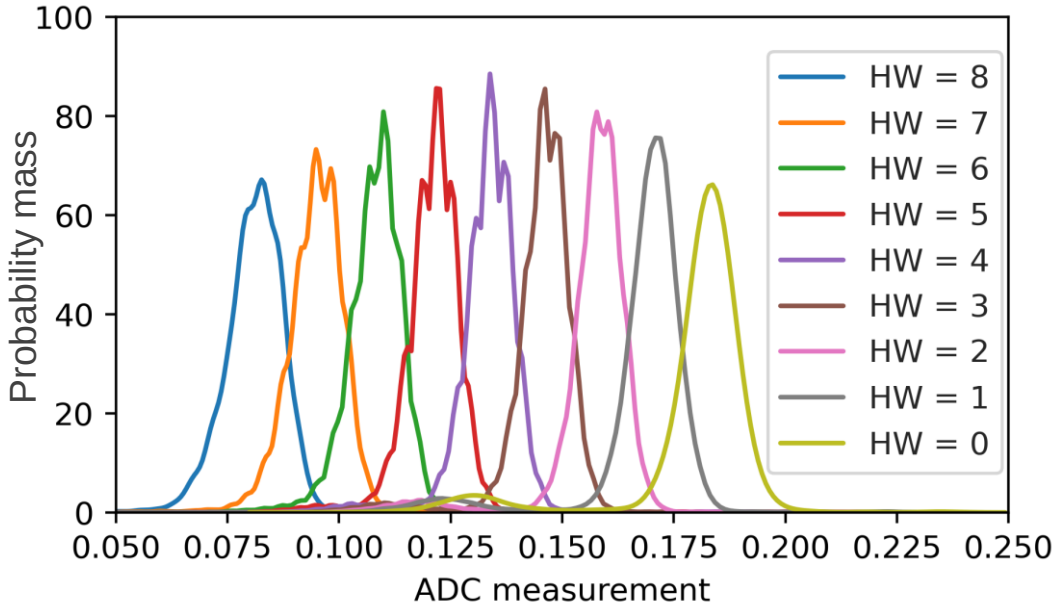
FY index generation

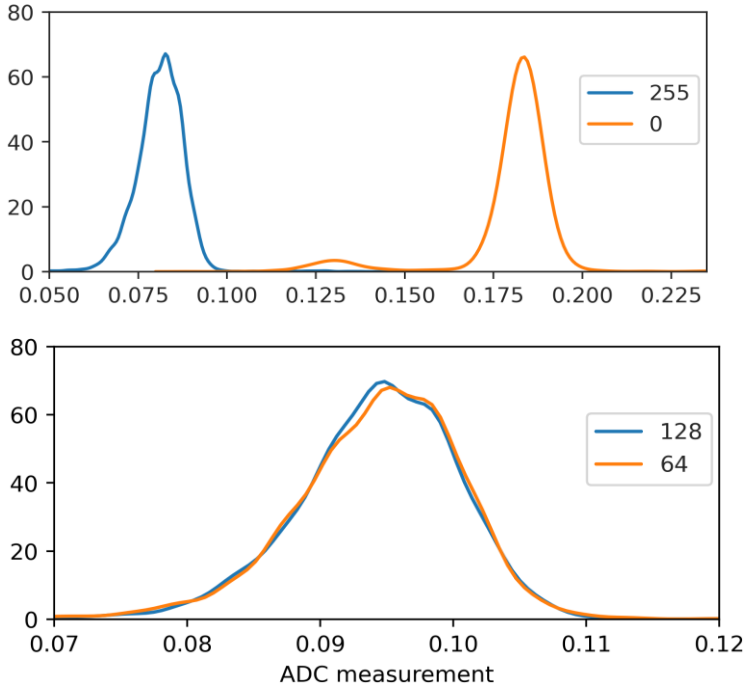
```
void FY_Gen(uint8* permutation, int max)
 1: for (i = 0; i < max; i++) do
 2:    permutation[i] = i;
 3: end for
 4: for (i = max - 1; i > 0; i=i-1) do
 5:    int index = rand() % (i+1);
 6:    uint8 temp = permutation[index];
 7:    permutation[index] = permutation[i];
 8:    permutation[i] = temp;
 9: end for
```



T-test on 5K traces

Distributions of power consumption for FY index generation





Only FY indexes with HW = 0 and 8 can be distinguished with a high probablitiy (from a single trace)

Solution: Recover message bits with these indexes only & rotate cyclically

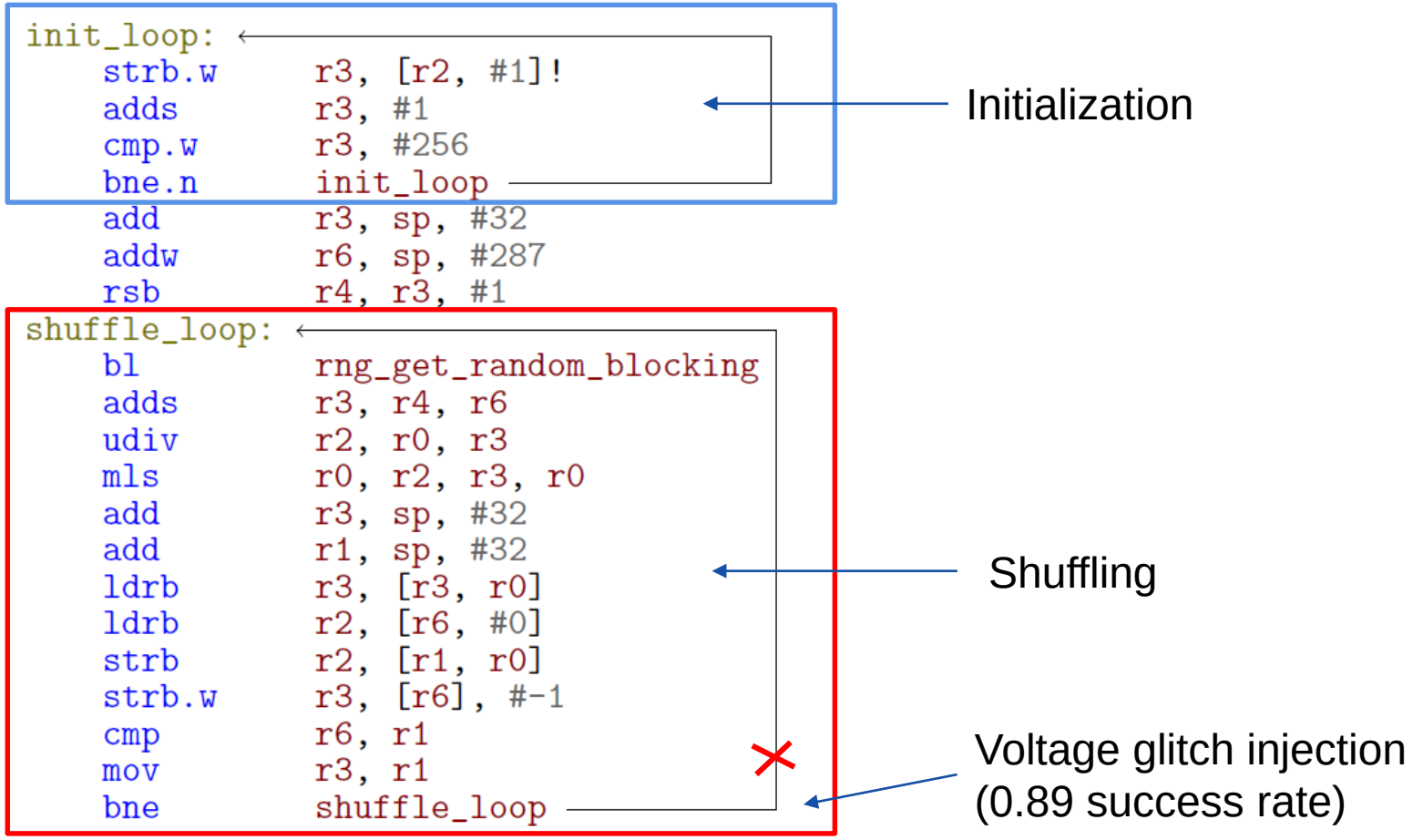
The maximum number of traces required for secret key recovery in all 10 attacks

Algorithm	Code distance		
	8	6	4
Saber ASHES'21 bit flipping method	-	-	61680
Saber FY index recovery method	-	4608	9216
Kyber FY index recovery method	48384	38016	59136

13 times smaller

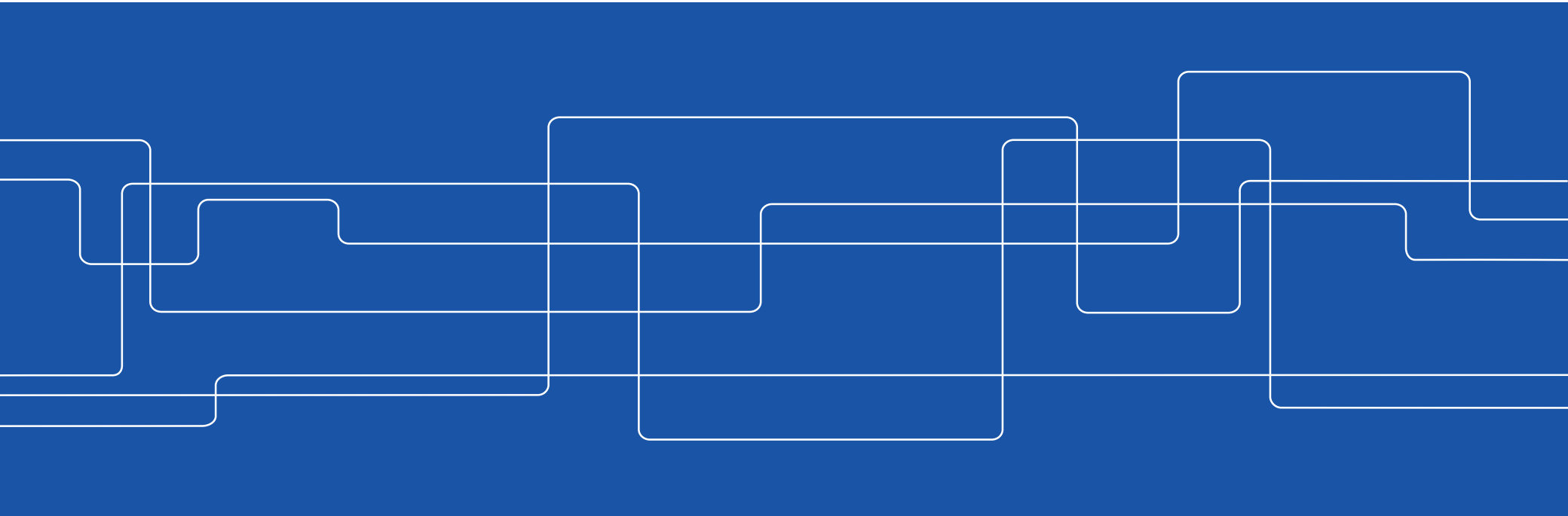
Skipping shuffling with a fault injection

FY index generation



Message recovery attack on a higher-order masked ML-KEM

Breaking a Fifth-Order Masked Implementation of CRYSTALS-Kyber by Copy-Paste, E. Dubrova, K. Ngo, J. Gärtner R. Wang, *RWC'23, APKC'23*



Attack details

- Kyber-768 C implementation (extended to higher orders):
  - Heinz, D.,Kannwischer, M.J., Land, G., Pöppelmann, T., Schwabe, P., Sprenkels: First-order masked Kyber on ARM Cortex-M4. Cryptology ePrint Archive, Report 2022/058 (2022)
  - Complied with optimization level -O3
- Attack point:
  - Re-encryption step of decapsulation
    - message encoding
- Target board:
  - ARM Cortex-M4 in CW308TSTM32F4

photo credit: Kalle Ngo

Non-masked message encoding in Kyber implementation of Kannwischer et al.

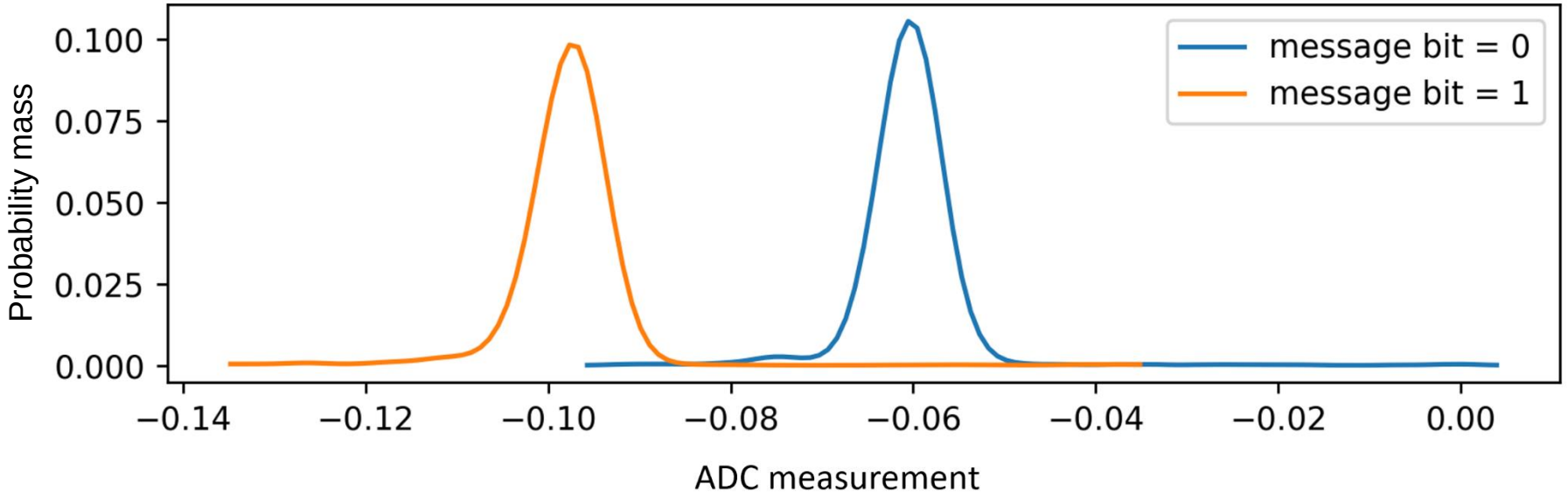
```
function POLY_FROMMSG(poly *r, unsigned char msg[32])
    uint16 mask
    for (int i=0; i < 32; i++) do
        for (int j=0; j < 8; j++) do
            mask = -((msg[i] >> j) &) 1 )          /* bit extraction */
            r.coeff[8*i+j] = mask & ((KYBER_Q + 1)/2)
        end for
    end for
end function
```

Mask takes values 0x0000 or 0xFFFF

Large difference in Hamming weight ⇒ easy to distinguish

First described by Amiet et al. for NewHope KEM, ICPQC'2020

Distributions of power consumption for message bits



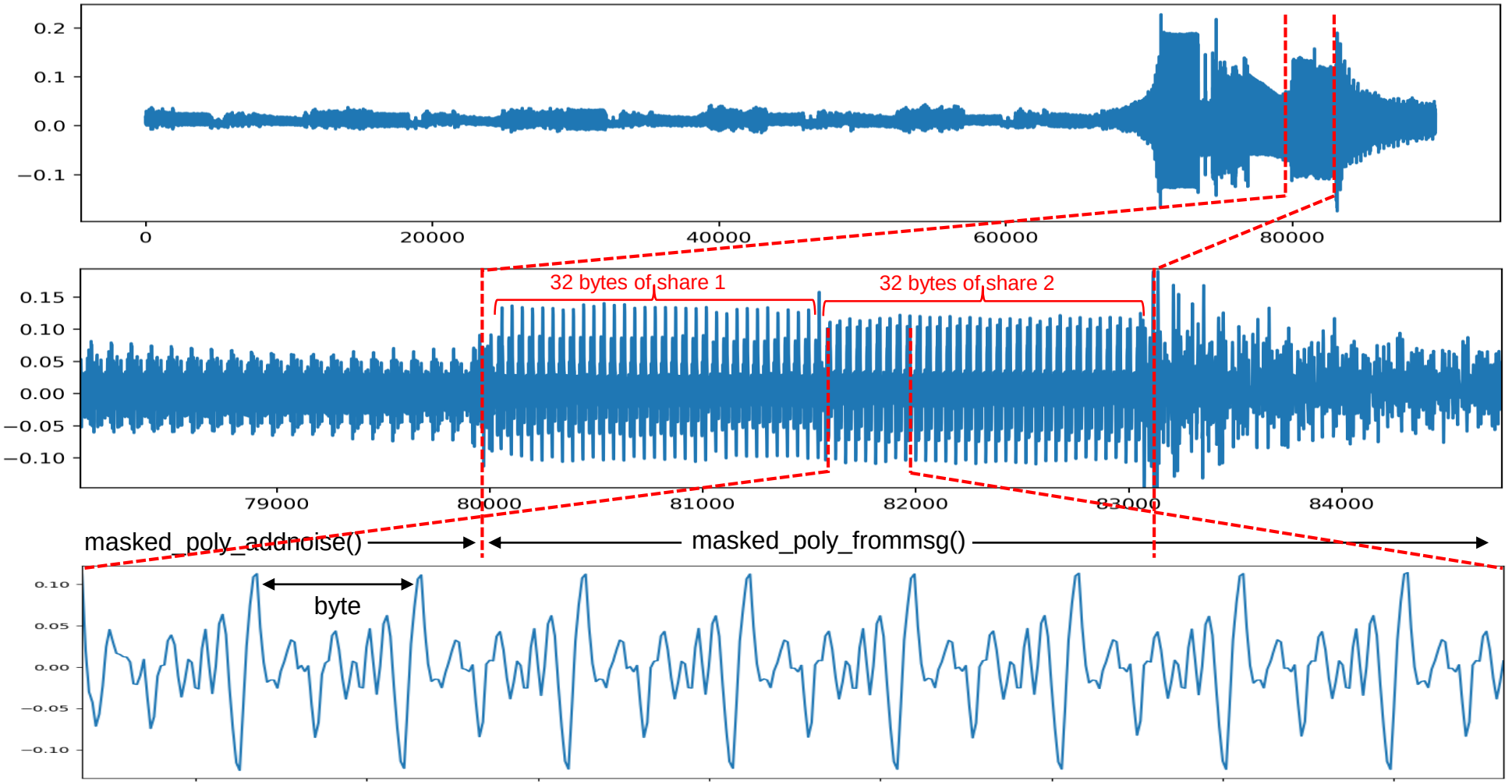
Non-overlapping distributions $\Rightarrow$ easy to distinguish

Masked message encoding in Kyber implementation on Heinz et al.

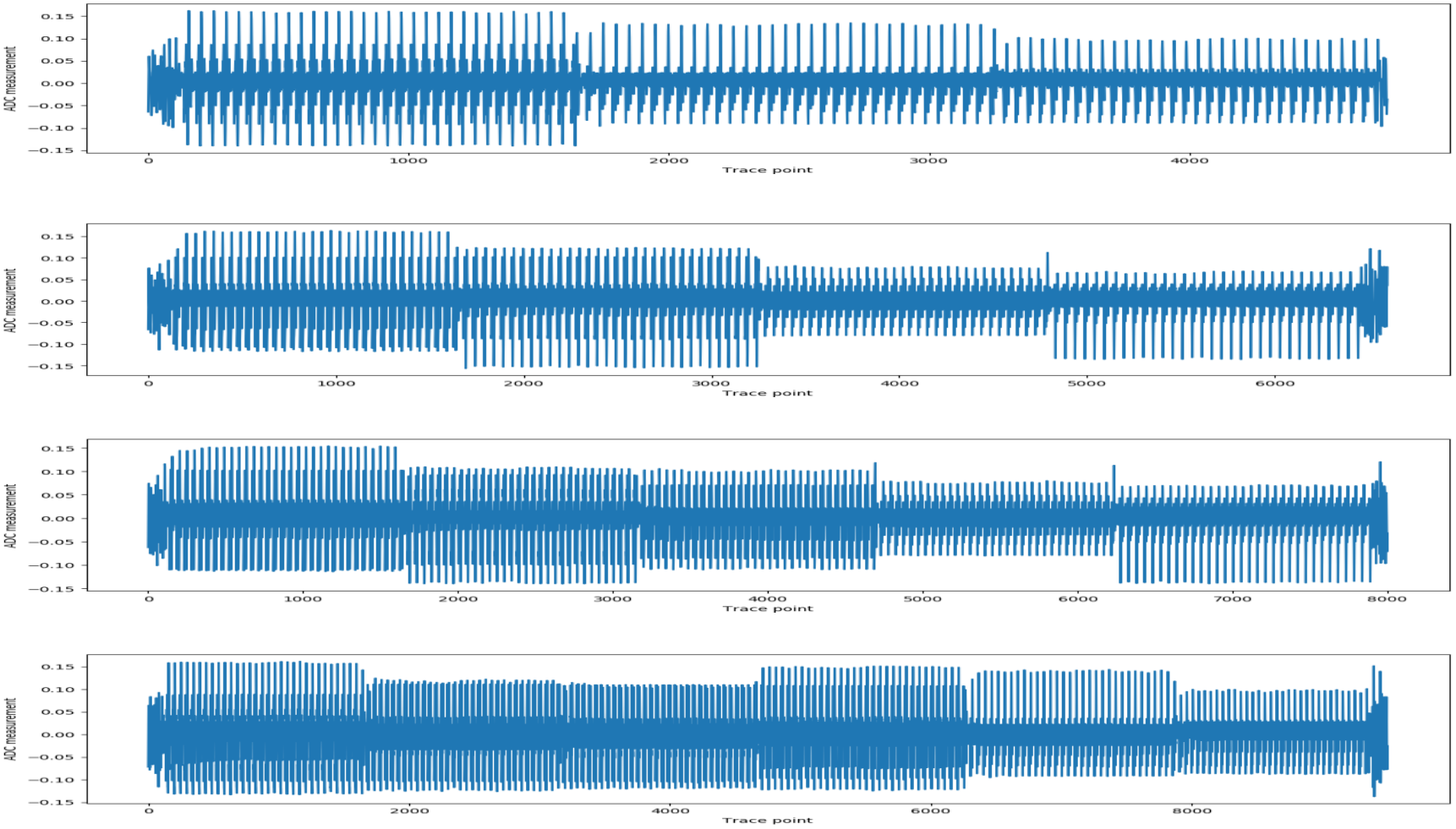
```
void masked_poly_frommsg(uint16 poly[2][256],
uint8 msg[2][32])
```

```
1:  for (i = 0; i < 32; i++) do
2:    for (j = 0; j < 8; j++) do
3:      mask = -((msg[0][i] » j) & 1);    /* Boolean share 0 bit extraction */
4:      poly[0][8*i+j] += (mask&((KYBER_Q+1)/2));
5:    end for
6:  end for
7:  for (i = 0; i < 32; i++) do
8:    for (j = 0; j < 8; j++) do
9:      mask = -((msg[1][i] » j) & 1);    /* Boolean share 1 bit extraction */
10:     poly[1][8*i+j] += (mask&((KYBER_Q+1)/2));
11:   end for
12: end for
13: ...
```

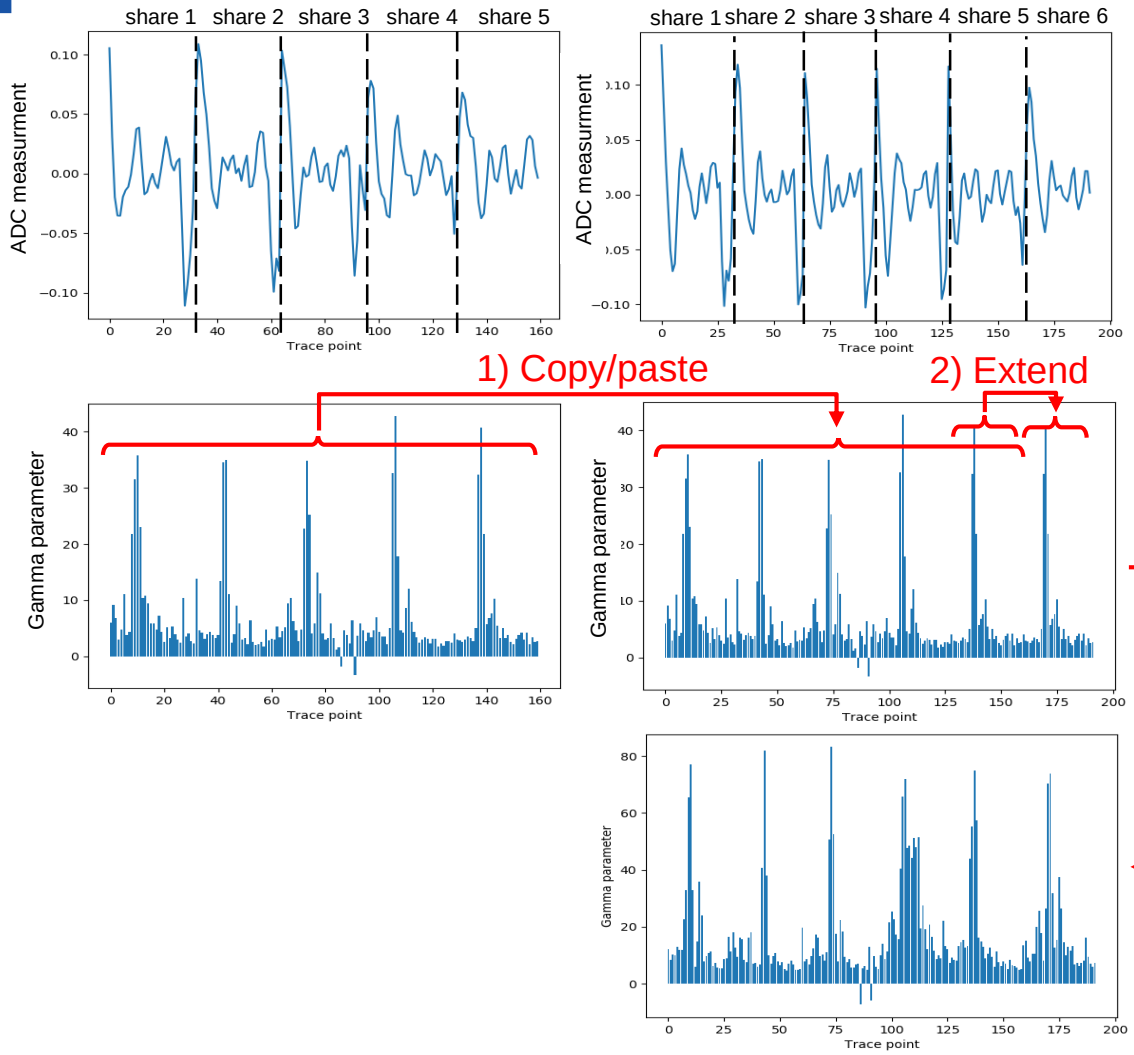
Segment of power trace of re-encryption in Kyber implementation on Heinz et al.



More shares ⇒ more 32-byte blocks



Copy-paste method



Power traces
(cut & concatenated
i^{th} bits of shares)

Weights of MLP
BatchNorm.1 layer
before training

Weights of MLP
BatchNorm.1 layer
after training

Attack results for the first-order masking

Attack type	Mean empirical probability to recover i^{th} message bit								Avg.
	0	1	2	3	4	5	6	7	
Single-trace	0.9992	0.9989	0.9953	0.9841	0.9876	0.9835	0.9393	0.9067	0.9743
With 4 rotations	0.9994	0.9991	0.9993	0.9990	0.9988	0.9885	0.9993	0.9992	0.9991

20-trace attack results for 5-order masking (with 4 negacyclic rotations and 5 repetitions)

ω	Mean empirical probability to recover i^{th} message bit								Avg.
	0	1	2	3	4	5	6	7	
5	1.0000	0.9987	1.0000	0.9989	1.0000	0.9992	1.0000	0.9988	0.9995

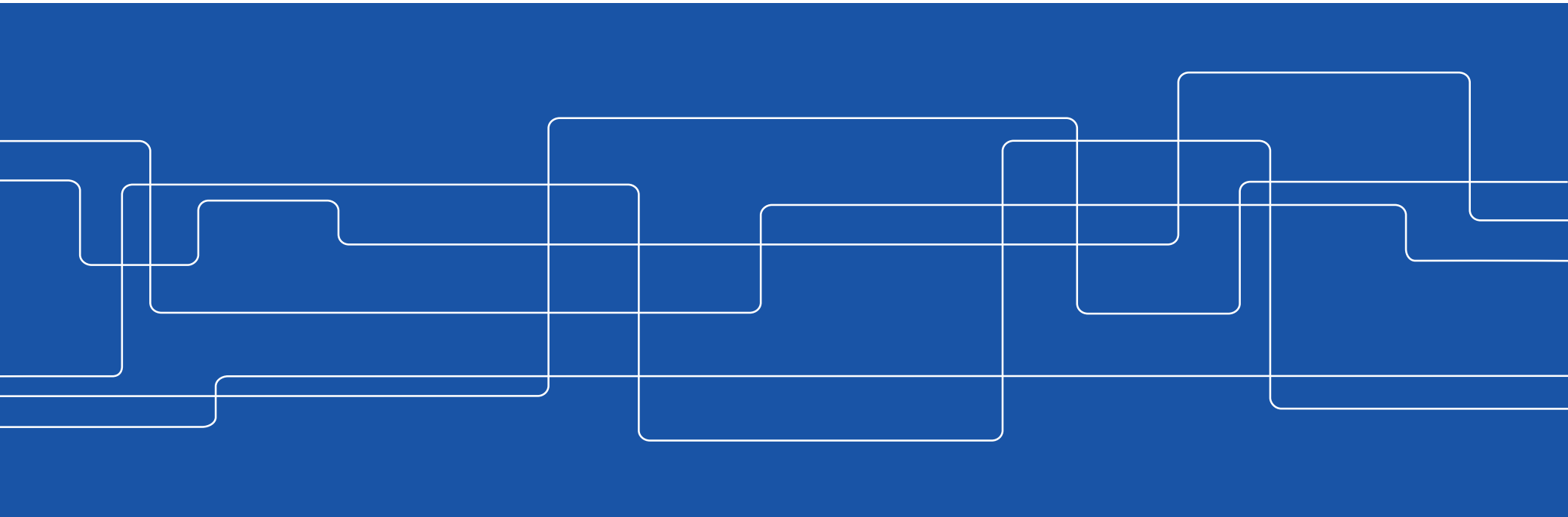
ω	5
p_{mesage}	0.8709

Since ranom masks are updated at each execution, errors in repeated measurments are more independent than in the non-masked case

Secret Key Recovery Attacks on ML-DSA

Ruize Wang, Kalle Ngo, Joel Gärtner, Elena Dubrova, Ruize et al. Unpacking Needs Protection: A Single-Trace Secret Key Recovery Attack on Dilithium, IACR Communications in Cryptology, 2024, 1(3)

A Single-Trace Fault Injection Attack on Hedged Module Lattice Digital Signature Algorithm (ML-DSA), S. Jendral, J. P. Mattsson and E. Dubrova, *FDTC'2024*



Attack details

- Dilithium-2 C implementation (unprotected):
  - Abdulrahman, A., Hwang, V., Kannwischer, M.J., Sprenkels, A., Faster Kyber and Dilithium on the Cortex-M4, ACNS'2022
  - Complied with optimization level -O3
- Attack point:
  - Secret key unpacking *skDecode* at the first step of step of signing algorithm
- Target board:
  - ARM Cortex-M4 in CW308TSTM32F4

photo credit: Kalle Ngo

C code of secret key unpacking procedure

```
void unpack_sk(uint8_t rho, uint8_t tr, uint8_t key, polyvec *t0, smallpoly
s1, smallpoly s2, uint8_t sk)
unsigned int i;
 1:  ...  unpacking rho, tr and key ...
 2: for (i = 0; i < L; ++i)  do
 3:     small_polyeta_unpack(&s1[i], sk + i*POLYETA_PACKEDBYTES);
 4: end for
 5: sk += L*POLYETA_PACKEDBYTES;
 6: for (i = 0; i < K; ++i)  do
 7:     small_polyeta_unpack(&s2[i], sk + i*POLYETA_PACKEDBYTES);
 8: end for
    ...
```

Called $\ell = 4$ times to unpack 96 bytes of each of ℓ polynomials of s_1 into $n = 256$ coefficients in the range $[-\eta, \eta]$, $\eta = 2$

Called $k = 4$ times to unpack 96 bytes of each of k polynomials of s_2 into n coefficients in the range $[-\eta, \eta]$

C code of small_polyeta_unpack() procedure

```
void small_polyeta_unpack(smallpoly *r, uint8_t *a)
unsigned int i;
 1: for (i = 0; i < N/8; ++i)  do
 2:     r->coeffs[8*i+0] = (a[3*i+0] >> 0) & 7;
 3:     r->coeffs[8*i+1] = (a[3*i+0] >> 3) & 7;
 4:     r->coeffs[8*i+2] = ((a[3*i+0] >> 6) | (a[3*i+1] << 2)) & 7;
 5:     r->coeffs[8*i+3] = (a[3*i+1] >> 1) & 7;
 6:     r->coeffs[8*i+4] = (a[3*i+1] >> 4) & 7;
 7:     r->coeffs[8*i+5] = ((a[3*i+1] >> 7) | (a[3*i+2] << 1)) & 7;
 8:     r->coeffs[8*i+6] = (a[3*i+2] >> 2) & 7;
 9:     r->coeffs[8*i+7] = (a[3*i+2] >> 5) & 7;

10:     r->coeffs[8*i+0] = ETA - r->coeffs[8*i+0];
11:     r->coeffs[8*i+1] = ETA - r->coeffs[8*i+1];
12:     r->coeffs[8*i+2] = ETA - r->coeffs[8*i+2];
13:     r->coeffs[8*i+3] = ETA - r->coeffs[8*i+3];
14:     r->coeffs[8*i+4] = ETA - r->coeffs[8*i+4];
15:     r->coeffs[8*i+5] = ETA - r->coeffs[8*i+5];
16:     r->coeffs[8*i+6] = ETA - r->coeffs[8*i+6];
17:     r->coeffs[8*i+7] = ETA - r->coeffs[8*i+7];
18: end for
```

range [0, 2η] (in reverse)

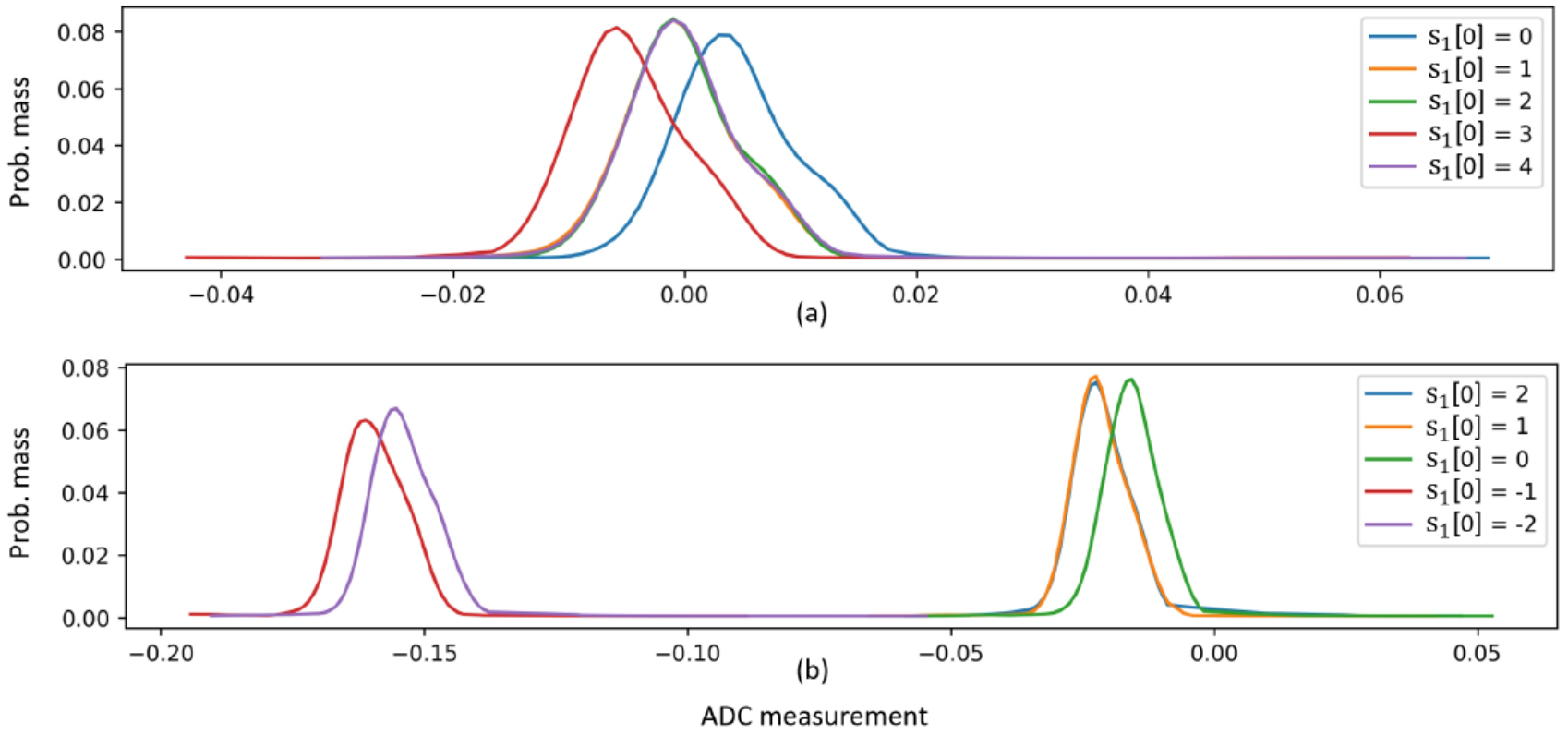
range [$-\eta$, η]

Power analysis

- Power consumption in a software implementation is typically proportional to the Hamming weight (HW) of processed data
- The coefficients of s_1 and s_2 are represented as 16-bits integers
- Negative numbers are represented in two's complement

  $\Rightarrow$ -1 = 0xFFFF and -2 = 0xFFFE
- The Hamming weight of (4, 3, 2, 1, 0) is HW = (1, 2, 1, 1, 0)
- The Hamming weight of (−2,−1, 0, 1, 2) is HW = (15, 16, 0, 1, 1)
- The pairs are unique: ((1, 15), (2, 16), (1, 0), (1, 1), (0, 1))

Distributions of power consumption



Distributions of power consumption during the processing of the 1st coefficient of s_1 by small_polyeta_unpack(): (a) in the range [0, 2η], (b) in the range [$-\eta$, η]

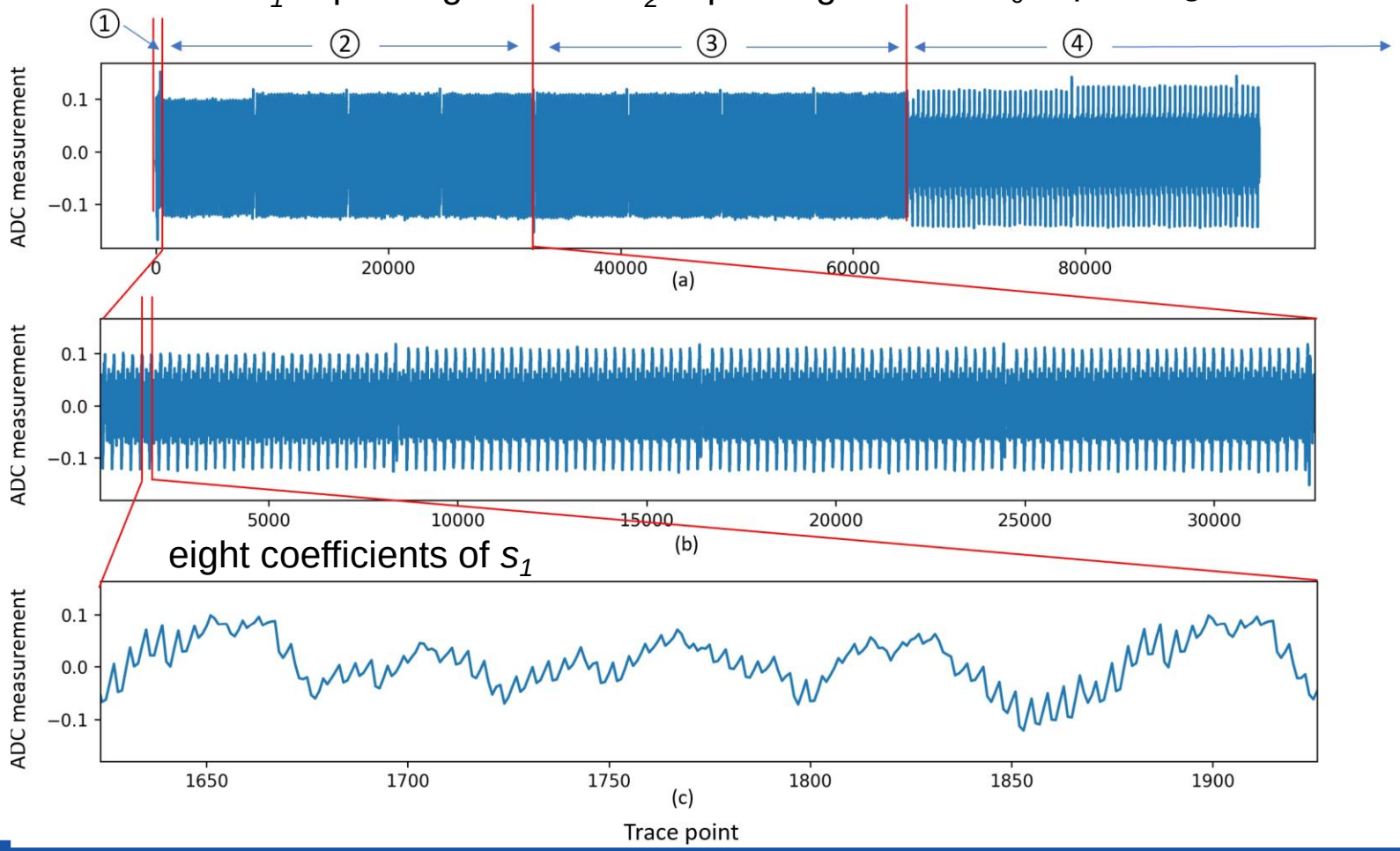
Segment of trace of small_polyeta_unpack()

ρ, tr and *K* unpacking

s_1 unpacking

s_2 unpacking

t_0 unpacking



Time for capturing the training and test sets

Training set	Time for capturing 5×2.5K traces
	4.8 hrs

Test set	Time for capturing N traces		
	$N = 1$	$N = 100$	$N = 1000$
	1.2 sec	36.6 sec	358.2 sec

- Eight neural network models were trained
- The training time for a single model is less than 40 min

  $\Rightarrow$ the total profiling time is less than 10 hours

Experimental results

Table: Empirical probability to recover a single coefficient of s_1, s_1[j], from N traces (mean over all s_1[j] with the same j mod 8, for $j \in \{0, 1, \ldots, 1023\}$)

N	$j \bmod 8$							
	0	1	2	3	4	5	6	7
1	0.942	0.925	0.975	0.967	0.945	0.920	0.924	0.784
10	0.980	0.951	0.999	0.993	0.988	0.951	0.956	0.863
100	0.983	0.952	0.999	0.995	0.991	0.954	0.959	0.870
1000	0.983	0.954	0.999	0.995	0.991	0.956	0.960	0.871

Probability flattens after $N = 100$

The compiler does not allocate *store* and *mask* instructions in the same way as for the other coefficients
$\Rightarrow$ values are manipulated less
$\Rightarrow$ weaker leakage

Two post-processing methods

To complement power analysis, we use two methods:

1. Solving a system of linear equations induced by $t = A s_1 + s_2$
   - assumes the knowledge of t_0

2. Lattice reduction
   - aims to find a new basis for the same lattice but with shorter, more orthogonal basis vectors

Linear algebra (LA)-based method

1. Predict all coefficients of s_1 and s_2 using NN models
2. Sort by the maximum predicted probability
3. Accept as correct the top half
4. Derive the rest by solving linear equations

Probability to recover 1024 coefficients of $\mathbf{s}_1$ and $\mathbf{s}_2$				LA post-processing
$N = 1$	$N = 10$	$N = 100$	$N = 1000$	CPU time
0.09	0.83	0.99	1	2 sec

Lattice reduction-based method

1. Predict all coefficients of s_1 using NN models
2. Sort by the maximum predicted probability
3. Accept as correct a top fraction x
4. Derive the rest by lattice reduction
   - Block Korkin-Zolotarev (BKZ) algorithm

estimate of Core-SVP bit hardness, $\alpha \approx 0.292\beta$

BKZ blocksize

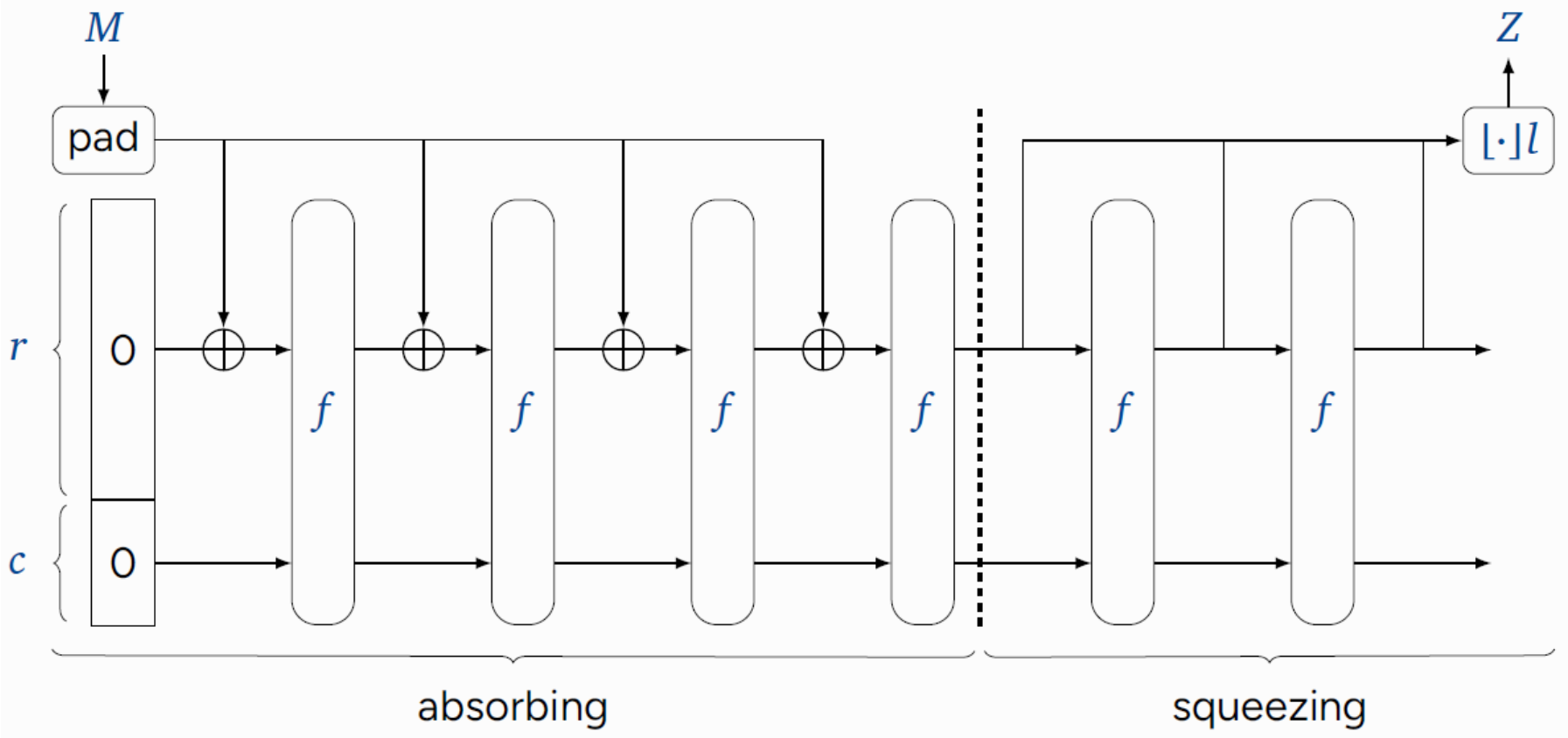
Fraction x	Probability to recover $\lfloor x \cdot 1024 \rfloor$ coeff. of $\mathbf{s}_1$				BKZ post-processing	
	$N = 1$	$N = 10$	$N = 100$	$N = 1000$	β	α
3/4	0	0.16	0.70	0.82	160	46.7
4/5	0	0.04	0.42	0.54	115	33.6
5/6	0	0.01	0.20	0.26	86	25.2

attack takes ≈ 6 hours

Skipping SHAKE256 absorbtion by fault injection

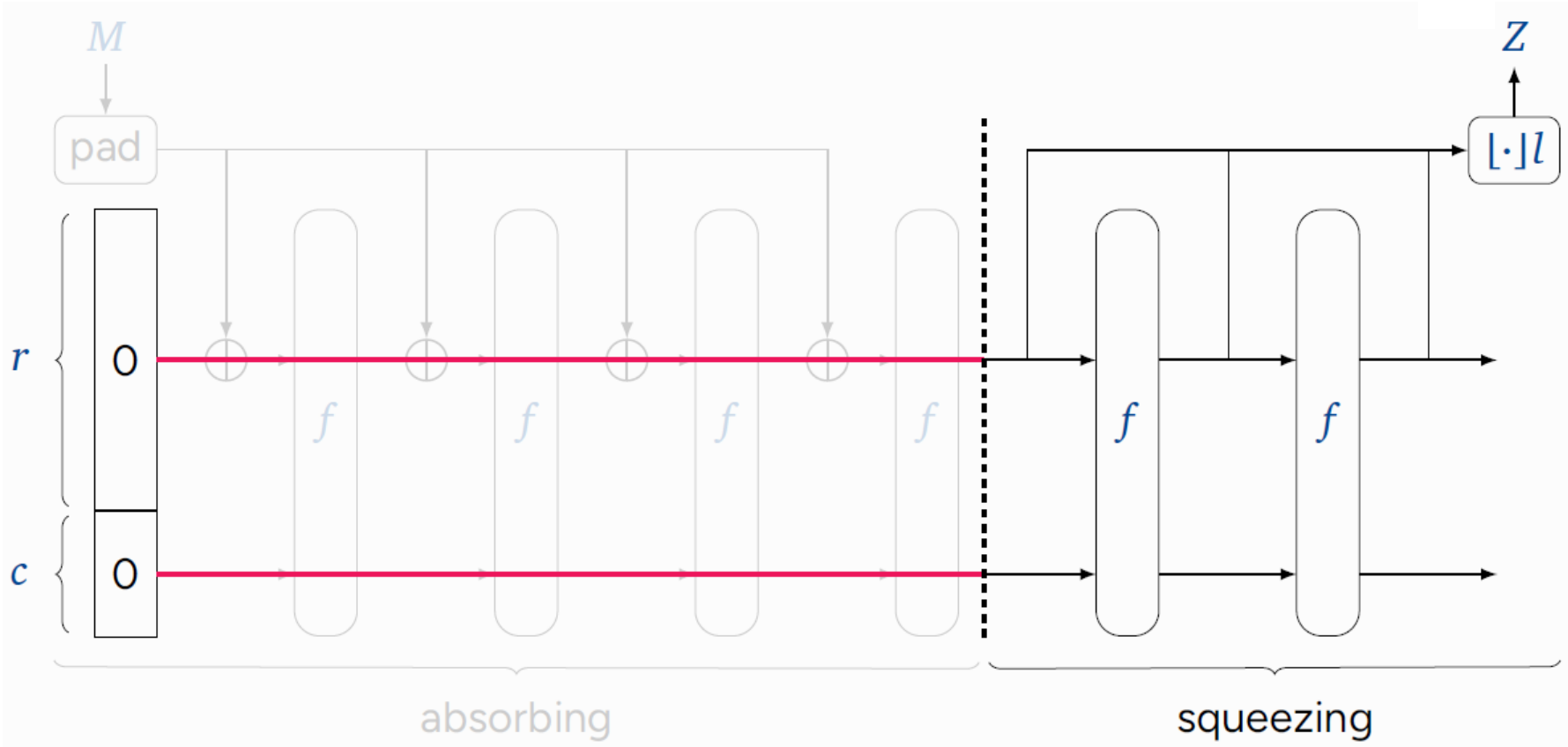
- SHAKE256 is an eXtendable Output Function (XOF) based on Keccak family
  - Uses sponge construction (absorbtion + squeezing)
  - Used in ML-DSA to expand seeds, hash messages and sample matrices/vectors

Sponge construction



Skipping absorbtion

Output is a constant



Key recovery from a known ρ´

Input: Private key sk, message M
Output: Signature σ

1: $(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) \leftarrow \text{skDecode}(sk)$
2: $\mathbf{A} \leftarrow \text{ExpandA}(\rho)$
3: $\mu \leftarrow \text{H}(tr \parallel M, 512)$
4: $rnd \leftarrow \{0,1\}^{256}$
5: $\rho' \leftarrow \text{H}(K \parallel rnd \parallel \mu, 512)$
6: $\kappa \leftarrow 0$
7: $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
8: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
9: $\quad \mathbf{y} \leftarrow \text{ExpandMask}(\rho', \kappa)$
10: $\quad \tilde{c} \leftarrow \mathbf{Ay}$
11: $\quad c \leftarrow \text{H}(\tilde{c} \parallel \mu)$
12: $\quad \mathbf{z} \leftarrow \mathbf{y} + cs_1$
13: $\quad$ **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ **then** $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
14: $\quad \kappa \leftarrow \kappa + l$
15: $\sigma \leftarrow \text{sigEncode}(\tilde{c}, \mathbf{z} \bmod^{\pm} q)$
16: **return** σ

Summary

- Practical side-channel and fault attacks on software implementations of ML-KEM and ML-DSA are possible
- Implementations protected by both masking and shuffling, or higher-order masking, may also be vulnerable
  - Features of lattice-based algorithms such as bit flipping and cyclic rotation are helpful for the attacker
- Stronger countermeasures against physical attacks on software implementations of ML-KEM and ML-DSA are needed


SXQgaXMgcG9zc2libGUgdG8g
aW52ZW50IGEgc2luZ2xlIG1h
Y2hpbmUgd2hpY2ggY2FuIGJl
IHVzZWQgdG8gY29tcHV0ZSBh
bnkgY29tcHV0YWJsZSBzZXF1
ZW5jZS4gSWYgdGhpcyBtYWNo
CDIS
aCBpcyB3cml0dGVuIHRoZSBT
LkQgb2Ygc29tZSBjb21wdXRp
bmcgbWFjaGluZSBNLCB0aG
VuIFUgd2lsbCBjb21wdX
RlIHRoZSBzYWllIH
NlcXVlbmNlIG
FzIEOuCg
==


MSB
Myndigheten för
samhällsskydd
och beredskap

Thank you!


TECOSA


VINNOVA