Introduction to Side-Channel Security of NIST PQC Standards

Think openly, build securely

April 4, 2023 – NIST PQC Seminar

Dr. Markku-Juhani O. Saarinen
Staff Cryptography Architect, PQShield Ltd
Professor of Practice, University of Tampere

Recap: (July 2022) NIST PQC Algorithms

Post-Quantum Crypto transition affects all Applications

NIST Post-Quantum Crypto: Selected July 2022, Standards 2024.

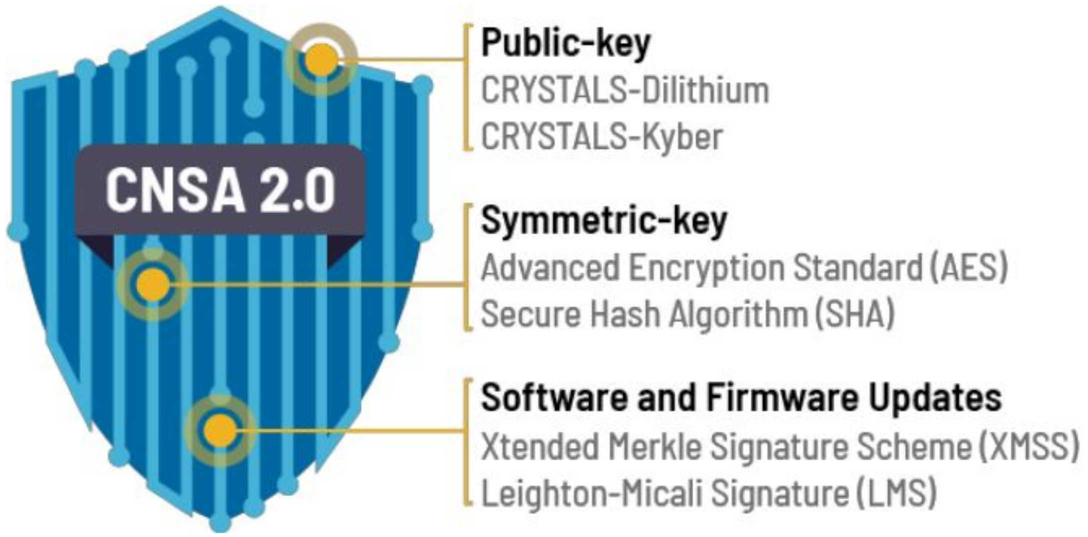
Kyber (+ Round 4 KEMs)
Replaces EC(DH), RSA key establishment.

Dilithium, Falcon, SPHINCS+
Replaces EC(DSA), RSA signatures.

Especially for U.S. Government Entities:

- *Active transition effort expected (presidential directives NSM-08, NSM-10).*
- *Regulations mandate FIPS 140-3 cryptography -> also for PQC modules.*

Recap: (September 2022) CNSA 2.0 / NIAP



CNSA 2.0
Public-key
CRYSTALS-Dilithium
CRYSTALS-Kyber
Symmetric-key
Advanced Encryption Standard (AES)
Secure Hash Algorithm (SHA)
Software and Firmware Updates
Xtended Merkle Signature Scheme (XMSS)
Leighton-Micali Signature (LMS)

Transition 2025-2030-2035:

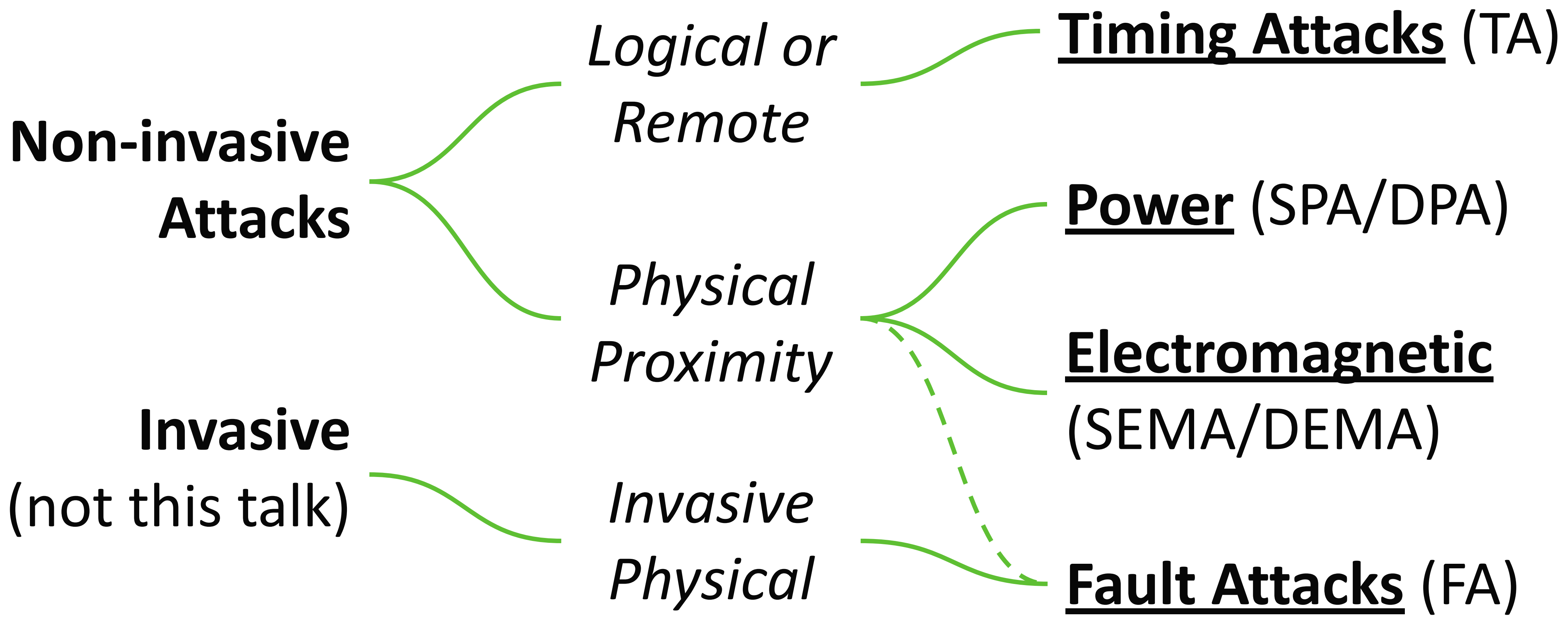
"Note that this will effectively deprecate [in NSS] the use of RSA, Diffie-Hellman (DH), and elliptic curve cryptography (ECDH and ECDSA) when mandated."

Table III: CNSA 2.0 quantum-resistant public-key algorithms

Algorithm	Function	Specification	Parameters
CRYSTALS-Kyber	Asymmetric algorithm for key establishment	TBD	Use Level V parameters for all classification levels.
CRYSTALS-Dilithium	Asymmetric algorithm for digital signatures	TBD	Use Level V parameters for all classification levels.

Side-Channel Attacks

Some Security Requirements in Applications



Everybody needs..

Timing Countermeasures

> A Quarter of a Century of Timing Attacks

Some Greatest Hits (in asymmetric crypto TA) Along the Years:

- P.C. Kocher: *"Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems."* (CRYPTO 1996. Target: RSAREF 2.0 running on MS-DOS.)
- D. Brumley and D. Boneh: *"Remote timing attacks are practical."* (USENIX Security 2003. OpenSSL RSA remote key recovery, CVE-2003-0147.)
- B. Brumley and N. Toveri: *"Remote Timing Attacks Are Still Practical."* (ESORICS 2011. OpenSSL ECDSA remote key recovery, CVE-2011-1945.)
- Q. Guo, T. Johansson. A. Nilsson, *"A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM."* (Crypto 2020, PC Oracle, demoed against a claimed const-time impl.)

Every generation gets to learn the special implementation tricks!

Basic Sources of Timing Leaks

(That are avoidable with careful programming)

1. Secret-controlled branches and loops:

```
if <secret> then { delay1(); } else { delay2(); }
```

2. Memory accesses (cache timing attacks). Can be a load or store.

```
ct = SBox[pt ^ key];        // observe latency with different inputs.
```

3. Arithmetic operations whose processing time just depends on inputs

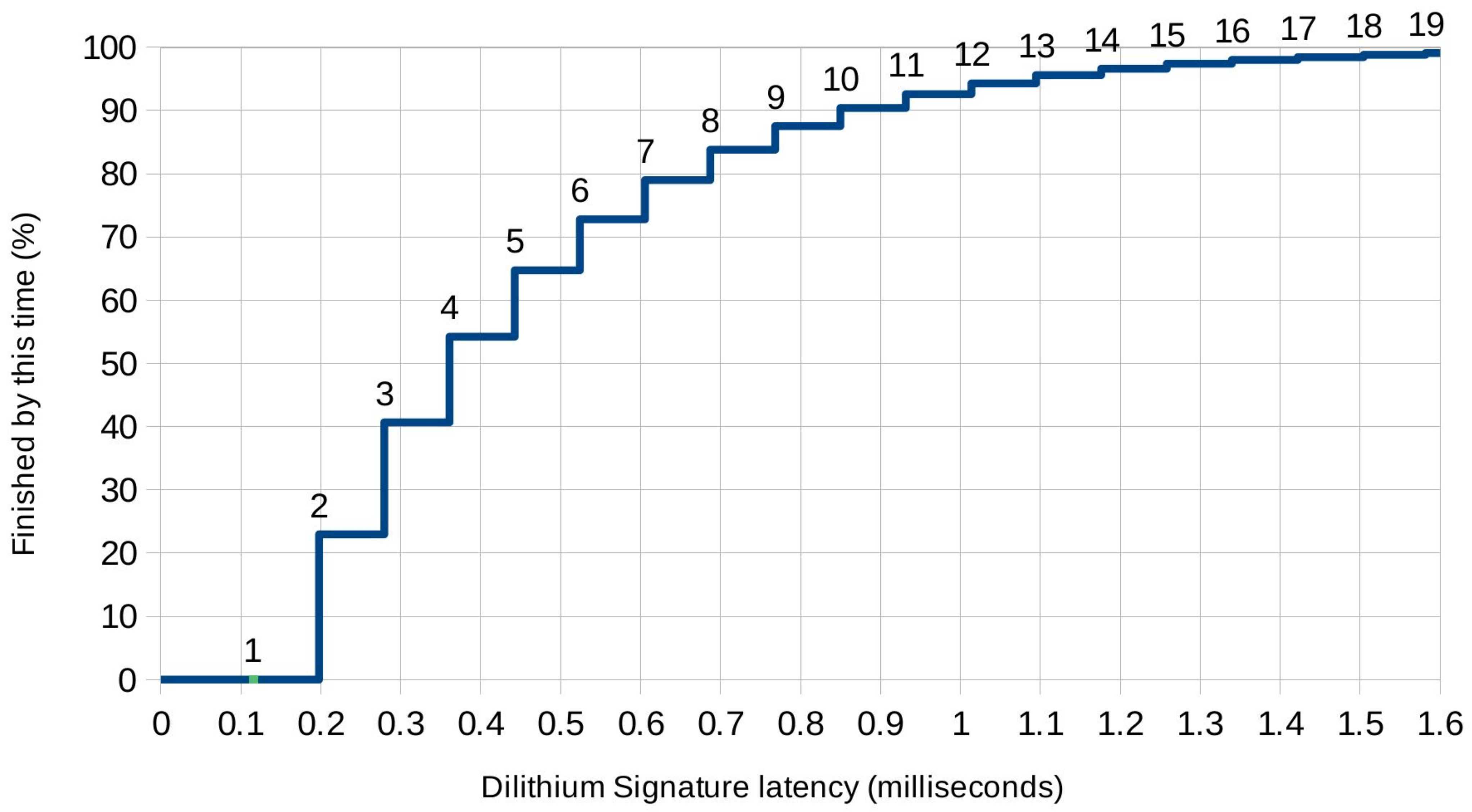
```
x = y % q;        // division and remainder ops are rarely constant-time.
```

PQC: Rejection Sampling

A lot of this in PQC: We don't mean *literally* "constant time"..

- You have a fair **6-sided dice** and want to have random numbers 1..5:
- Just reject sixes. The remaining 1..5 are uniform in that range.
- Number of rolls (time) and even the pattern of rejects can be public.
- But secure! does not leak 1,2,3,4,5.
- The same **Rejection Sampling** idea extends to arbitrary distributions. *(Dilithium does R.S. in four ways..)*

Dilithium Sign() is one big rejection sampler!



PQC: Sorting/Permutation Networks

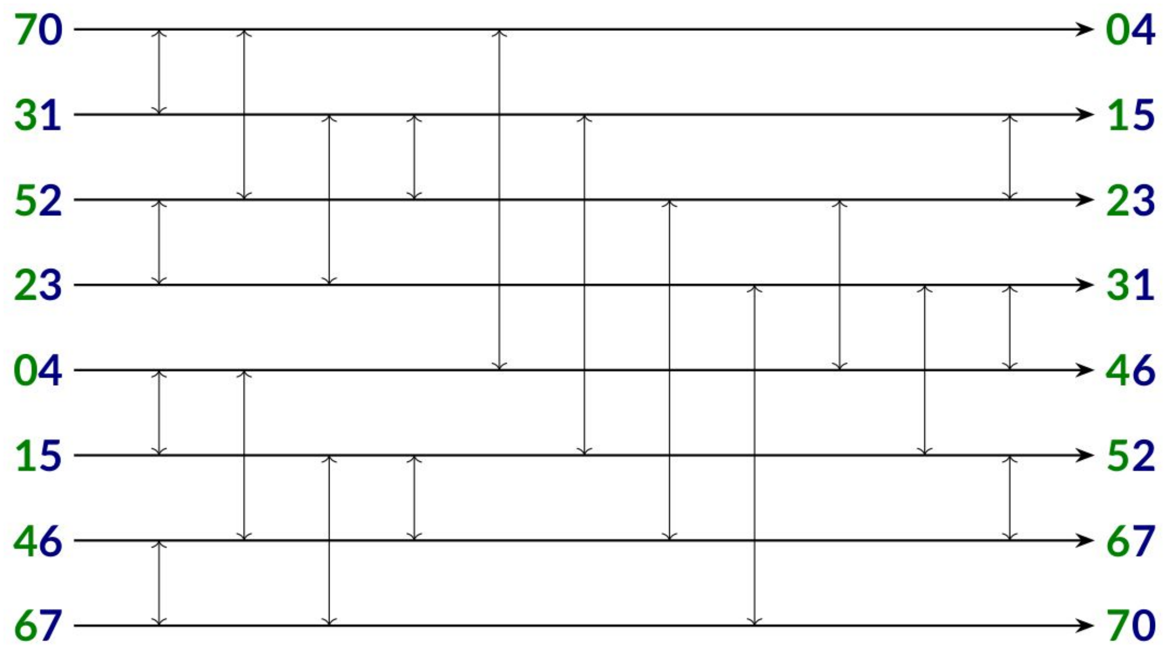
General CT technique – but explicit Classic McEliece

- **Example**: Insert permutation (or its inverse) in the most significant digits, data in least significant. Extract permuted digits from the sorted array.

- **Batcher's Odd-Even mergesort**: $2^m(m^2-m+4)/4-1$ *compare-swaps*.

- **Beneš network**: $2^m(2m-1)/2$ *controlled swaps*.

  (Note: Determining control bits in constant time is slow.)



When Hiring a (Post-Quantum) Crypto Dev..

Constant-time coding / algorithm knowledge is fundamental

- Have a “library” of solid CT replacements for **`memcmp()`** and similar functions.

- Identify conditionals, transform to straight-line code using Boolean operations 🤔

  `x = s ? a : b;` *vs.* `x = b ^ ((-(s & 1)) & (a ^ b));`

- Table-lookups: Bit-slicing (entire thing as a Boolean circuit), “full scan / collect”.

- No division instructions in modular arithmetic (use Montgomery, Barrett etc.)

- Know how to test with symbolic execution (e.g. valgrind) or on instruction level..

.. etc .. these are core crypto programming skills!

RISC-V “Data Independent Execution Time”

Portable constant-timeliness is already codified into the ISA

- One can use static analysis or dynamic variable tainting (in emulator) to verify that compiled code is using only the right instructions to handle secret data.

- But: ``*Constant-timeness*’’ of **Intel** and **ARM** instructions: mostly from experiments.

- RISC-V CETG codified timing as the **Zkt** extension for scalar, and the *(brand new)* **Zvkt** DIEL instruction list for vector. https://github.com/riscv/riscv-crypto/releases

- Official **RISC-V PQC ISA** is starting: https://lists.riscv.org/g/tech-pqc-cryptography
- Work is ongoing on *microarchitectural* side-channel assistance on RISC-V ISA level.

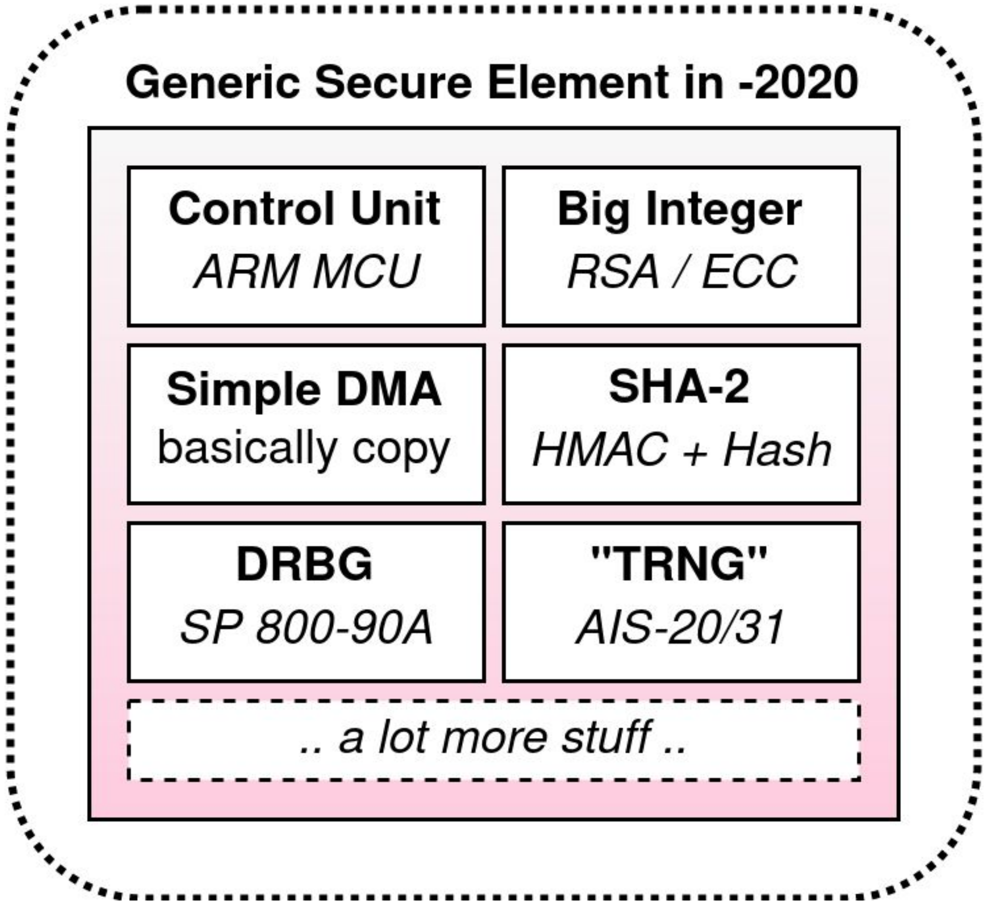
Couple of things about..

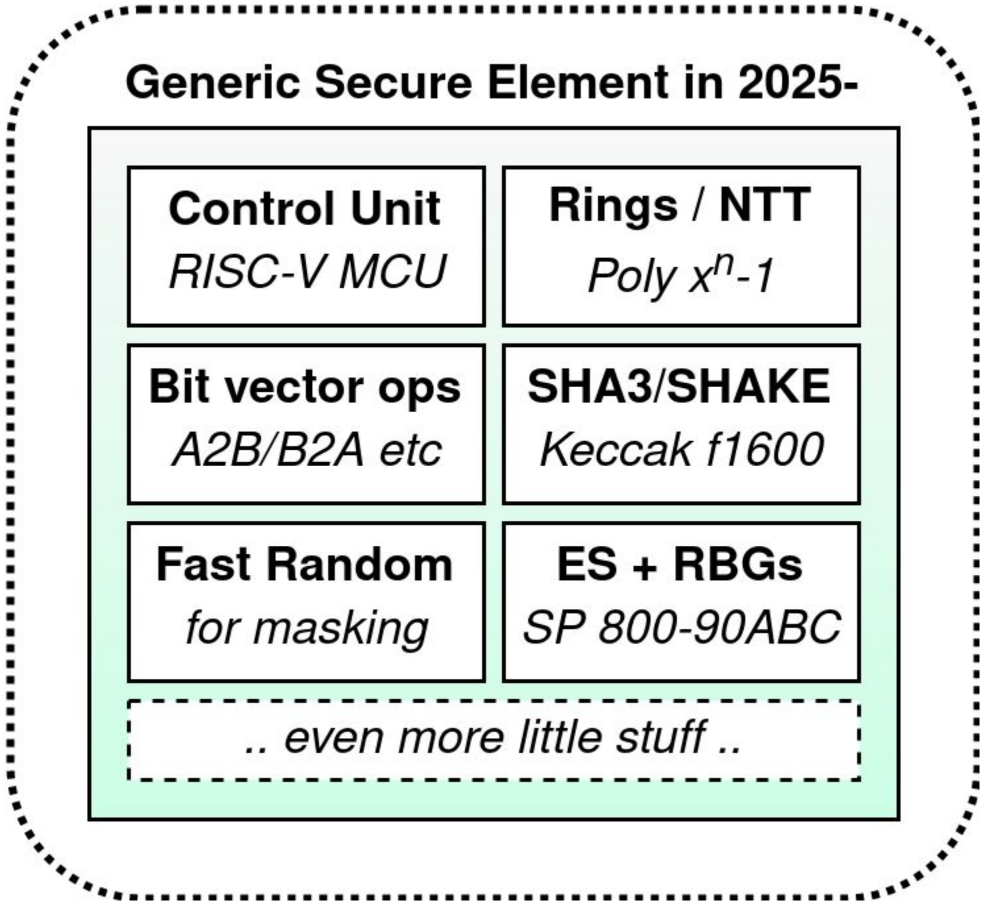
DPA & DEMA

Countermeasure “Transition”

Inherited SCA Application Requirements

Platform Security / RoT, Smart Cards, Authentication Tokens, etc





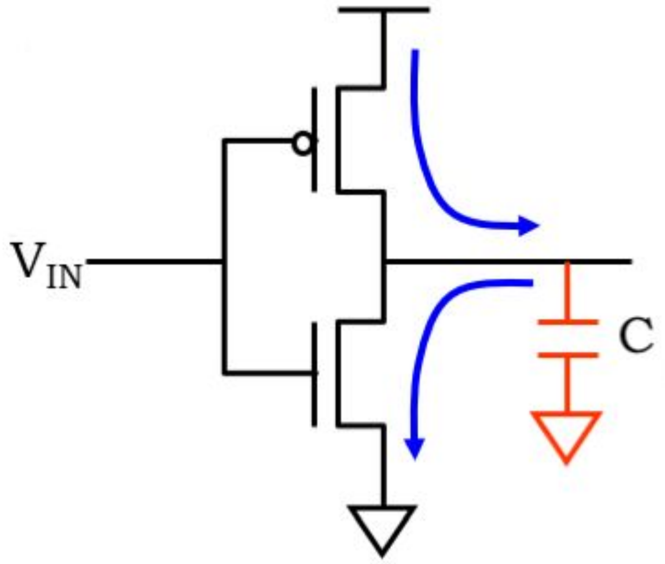
Power and EM Leakage: Think of “Toggling”

Physical models are complex, but basic mental model is simple.

- Logic changes ($0 \rightarrow 1$ or $1 \rightarrow 0$) consume ***dynamic power***. (There is also ***static power***, but resting bits generally don’t leak.)

- State changes also emanate on the ***electromagnetic spectrum***.

- User-visible CPU registers and memory are just one part of the vast logic machinery that handles your secret bits, and whose registers & data paths **toggle accordingly**.



Some Classical countermeasures – RSA and ECC

These were extremely simple algorithms + had algebraic structure

RSA: Blinding and masking (D. Chaum 1982, P. Kocher 1996+)

- Message blinding: Pick random **r**, compute blinded $\mathbf{c' = c r^{e}}$ (mod n), decrypt/sign **c'** instead of **c**: $\mathbf{m' = c'^{d}}$ (mod n), normalize by $\mathbf{m = m' r^{-1}}$.
- Exponent masking: use **d'** = **(p-1)(q-1)r + d** to randomize exponentiation.

ECC: J.-S. Coron, *"Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems."* Proc. CHES'99, pp. 292–302, 1999

- For 20+ years: Randomization of the Private Exponent [Scalar], Blinding the [Base] Point P, Randomized Projective Coordinates, + misc.

Basically 1 step – ModExp (RSA) or Scalar Mult (ECC) – to protect

New Countermeasures: Kyber and Dilithium

PQC needs masking (+ blinding, shuffling, random delays ..)

- Masking splits secrets into "shares." Successful measurement of an individual share does not leak secret info. Design "Masking Gadgets" to perform arithmetic steps.

Type:	Relationship:	Algebraic object:
A/Q:	$X = X0 + X1 \pmod{q}$	Prime q is 3329 (Kyber) or 8380417 (Dilithium).
A/N:	$X = X0 + X1 \pmod{2^N}$	May also be needed by Kyber and Dilithium.
B:	$X = X0 \oplus X1$	Various nonlinear functions, shifts, etc.

- **Most cryptographers agree**: Masking and other attack mitigation techniques for PQC algorithms are technically more complex than for older cryptography.

- **Why?** The algorithms are not homogenous like RSA or ECC but contain a number of dissimilar steps. One may have to design dozen different gadgets for one algorithm.

First question: What needs to be protected?

Mostly just the CSPs – “Critical Security Parameters”

Classification in Crypto Module world:

- Public Security Parameter (**PSP**) needs integrity only: Can’t be modified.
- Critical Security Parameter (**CSP**) needs confidentiality (secrecy) and integrity.
- Together these are Sensitive Security Parameters (**SSP** ≅ *All variables in crypto!*)

Section 7.8 of ISO/IEC 19790:2012(E) and 19790:2022(E): *``Non-invasive attacks attempt to compromise a cryptographic module by acquiring knowledge of the module’s* ***CSP****s without physically modifying or invading the module.’’*

For us, a CSP is any information that helps (the attacker) directly or indirectly to:

1. Determine a shared secret in a key establishment scheme or
2. Forge a signature in a signature scheme.

Background: Masking Generic Circuits

When we don't have a handy algebraic structure

Each bit **x** is split into d uniform random shares: $[[\mathbf{x}]] = x_0 \oplus x_1 \oplus .. \oplus x_{d-1}$.

*"We prove that the amount of side channel information required **grows exponentially in** [**d**], the number of shares."*

[S. Chari, C. S. Jutla, J. R. Rao, P. Rohatgi. CRYPTO '99]

*"We show that any circuit with n gates can be transformed into a circuit of size **O(nt²)** that is perfectly secure against all probing attacks leaking up to t bits at a time."*

("t-probing model") [Y. Ishai, A. Sahai, D. Wagner, CRYPTO '03]

Example: Some bit-level first-order gadgets..

Random bits may be required every time

SecXor:	**$[[X]] = [[A]] \oplus [[B]]$**	$X0 = A0 \oplus B0$
	(*no share mixing!*)	$X1 = A1 \oplus B1$
SecAnd:	**$[[X]] = [[A]] \wedge [[B]]$**	$X0 = (A0 \wedge B0) \oplus R \oplus (A0 \wedge B1)$
	(R is a random bit.)	$X1 = (A1 \wedge B0) \oplus R \oplus (A1 \wedge B1)$

Evaluation order matters (here from left to right). SecXor is $O(d)$ but SecAnd complexity increases in **quadratic** $O(d^2)$ fashion with the number of shares.

Recently, **quasi-linear** $O(d \log d)$ masking complexity has been achieved for some functions, but not for full Dilithium or Kyber. *More about this later..*

NIST PQC: Good News and Bad News

Kyber and Dilithium benefit from this a lot..

Core (Structured/Unstructured - Ring/Module) LWE: “irreversibility” of:

$$\mathbf{t} := \mathbf{A} \cdot [[\mathbf{s}]] + [[\mathbf{e}]]$$

..where **A** is a public generator, **s** and **e** are secret. The **t** is public (collapsed).

Good news: There is no multiplication between two secrets (say, **[[s]]** · **[[r]]**). Since **A** is public, the core operation *is linear: shares don’t interact.* It’s O(d) .

Bad News: The **s** and **e** distributions are non-uniform. Kyber and Dilithium require a lot of *mixing (mod q) arithmetic masking with Boolean masking*.

A bit similar to SHA2 vs SHA3

Symmetric Support Primitives

A Really Important Gadget: Masked Keccak

Both Kyber and Dilithium need a SCA-Secure SHA-3/SHAKE

- The **SHA3** hash and **SHAKE** Extendable-Output Function (XOF) are built on the 1600-bit, 24-round **Keccak** permutation. [Defined in FIPS 202, 2015]

- Luckily SHA-3 is a ***"Post-SCA"*** algorithm: The designers of Keccak had worked in the semiconductor industry, knew about side-channel security.

  [G. Bertoni, J. Daemen, M. Peeters, G. Van Assche. *"Building power analysis resistant implementations of Keccak."* The second SHA-3 Candidate Conference, NIST, 2010]

- The NIST LWC Winner **Ascon** inherits many of the same SCA features.

XORs are Good, Nonlinear ops are Expensive

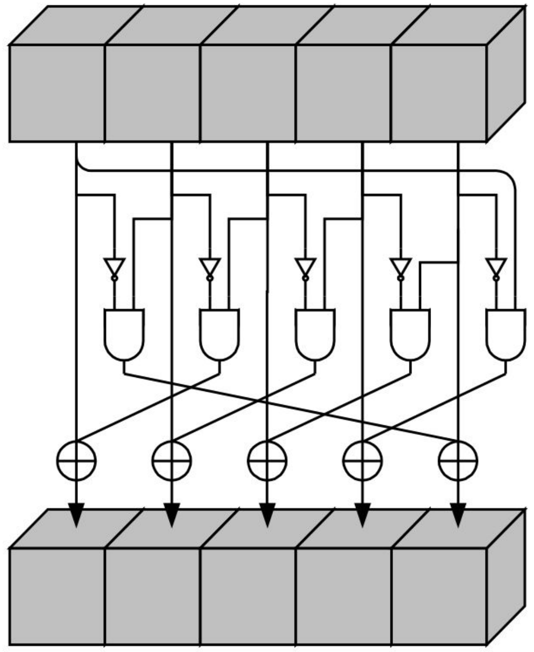
Keccak has a lot of XORs mixing, extremely simple non-linear Layer

- Keccak description has 5 "mappings" (**θ**, **ρ**, **π**, **χ**, **ι**), out of which 4 are linear – **θ**, **ρ**, **π**, **ι** can be implemented with XORs (not mixing shares!)

- The "S-Box" Chi (**χ**) is basically just 1 layer of ANDs.

- Due to structure, one can also save/recycle masking randomness with a **"threshold implementation" (TI)**.

- Relatively simple, but a Masked Keccak can be rather large; perhaps 100k GE. However would be *really fast*.



Chi (χ) "s-box" – FIPS 202 art!

So why not SHA-2?

For starters, it’s not a XOF. Also it’s a “pre-SCA” design.

- SHA-2 implementations need side-channel security if used as a HMAC or for key derivation (HKDF). It is **very** poorly suited for this.

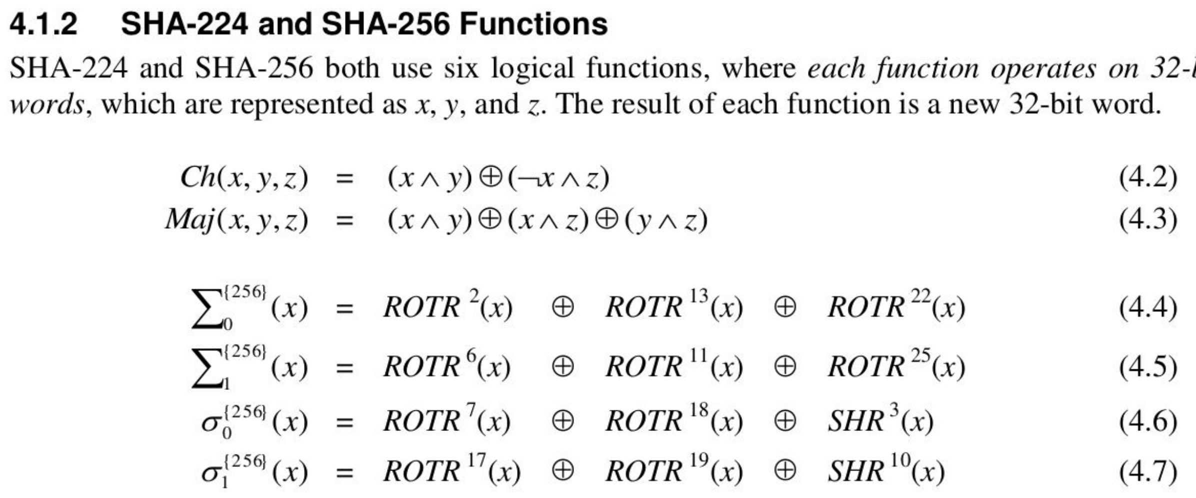
- Why? Main reason: SHA2-256/512 mixes **32/64-bit additions** with **Boolean XORs and nonlinear operations** in literally every round.

- This requires Boolean-domain adders or A2B/B2A, which is complex/slow.

[L. Goubin. *“A Sound Method for Switching between Boolean and Arithmetic Masking.”* CHES 2001] (See also M. Karroumi et al. *“Addition with blinded operands.”* COSADE 2014.)
[J.-S. Coron, J. Großschädl, M. Tibouchi, P. K. Vadnala. *“Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity.”* FSE 2015]

So why not SHA-2?

Mixing Additions with Boolean operations is literally all it does



4.1.2 SHA-224 and SHA-256 Functions

SHA-224 and SHA-256 both use six logical functions, where *each function operates on 32-bit words*, which are represented as *x*, *y*, and *z*. The result of each function is a new 32-bit word.

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z) \qquad (4.2)$$

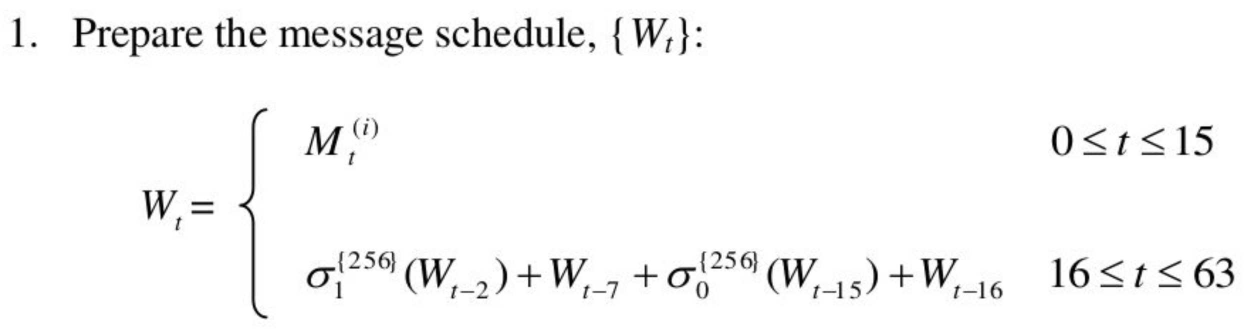
$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \qquad (4.3)$$

$$\textstyle\sum_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \qquad (4.4)$$

$$\textstyle\sum_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \qquad (4.5)$$

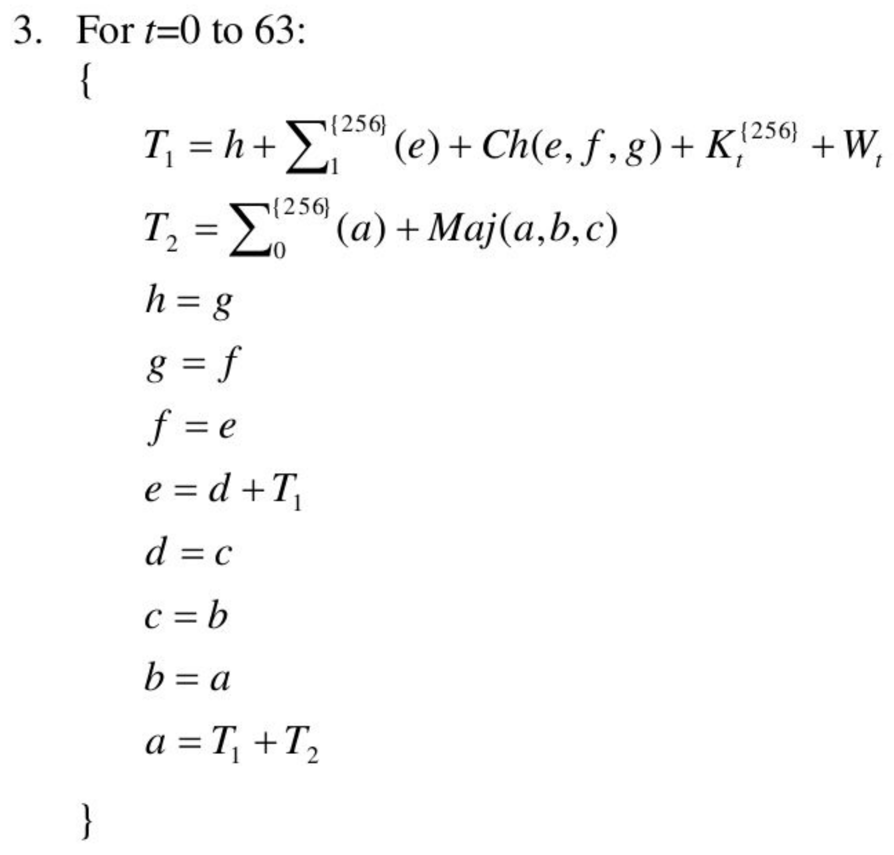
$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \qquad (4.6)$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x) \qquad (4.7)$$



1. Prepare the message schedule, $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

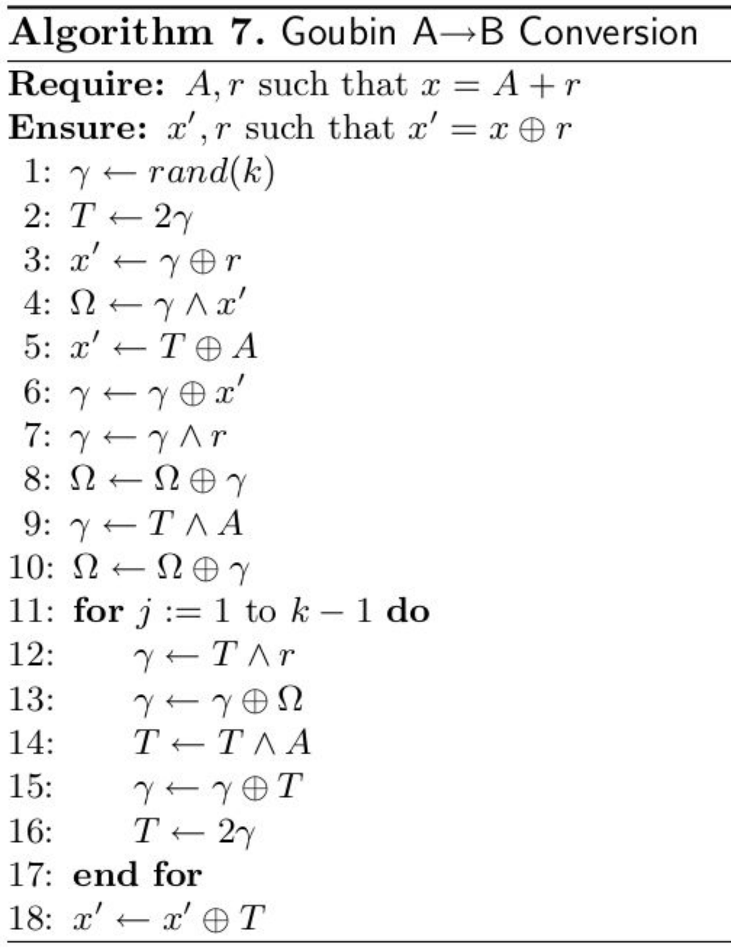


3. For t=0 to 63:

```
{
    $T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$
    $T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$
    $h = g$
    $g = f$
    $f = e$
    $e = d + T_1$
    $d = c$
    $c = b$
    $b = a$
    $a = T_1 + T_2$
}
```

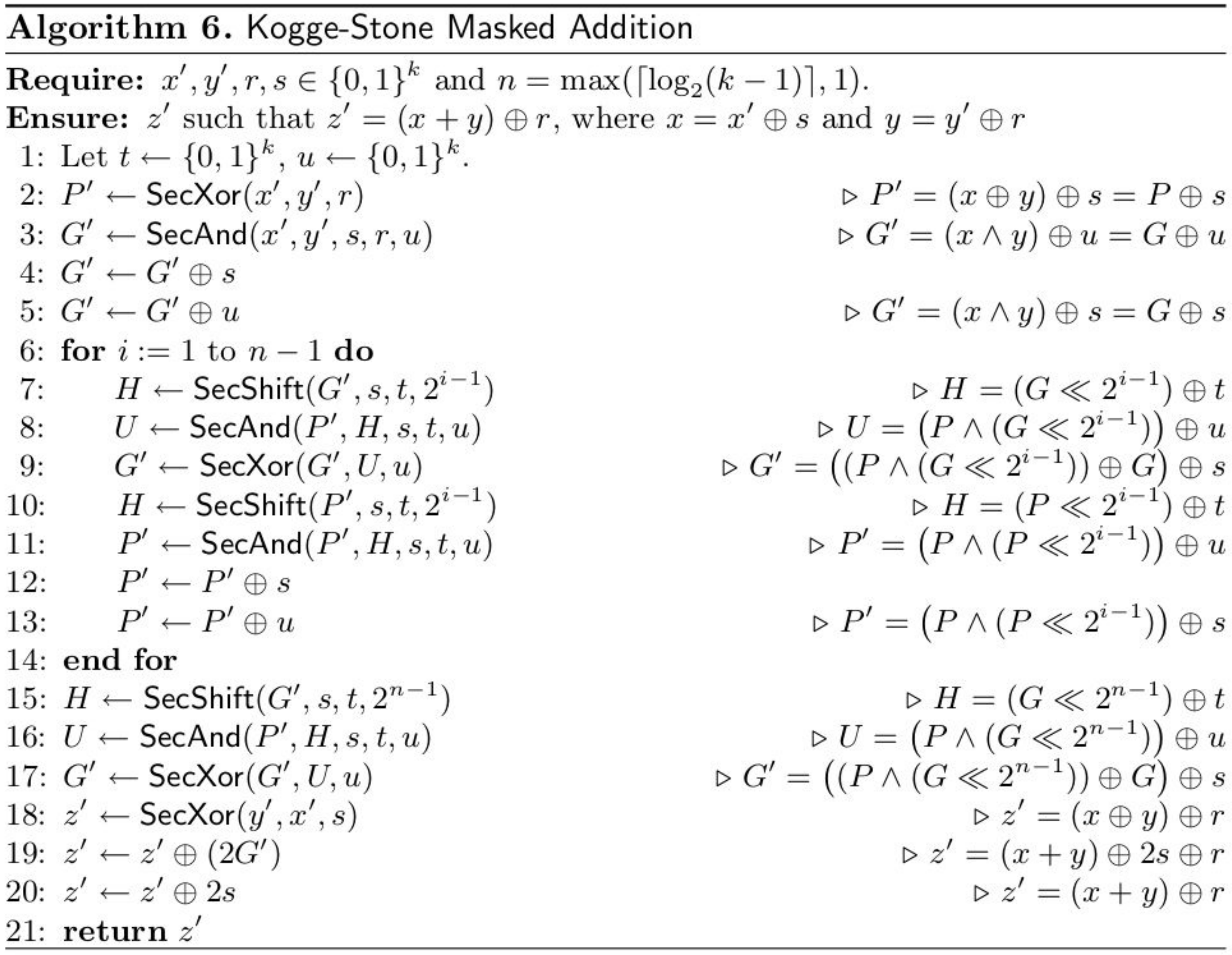
A2B and B2A transforms are complicated

Each mixing can be like..



Algorithm 7. Goubin A→B Conversion

```
Require: $A, r$ such that $x = A + r$
Ensure: $x', r$ such that $x' = x \oplus r$
 1: $\gamma \leftarrow rand(k)$
 2: $T \leftarrow 2\gamma$
 3: $x' \leftarrow \gamma \oplus r$
 4: $\Omega \leftarrow \gamma \wedge x'$
 5: $x' \leftarrow T \oplus A$
 6: $\gamma \leftarrow \gamma \oplus x'$
 7: $\gamma \leftarrow \gamma \wedge r$
 8: $\Omega \leftarrow \Omega \oplus \gamma$
 9: $\gamma \leftarrow T \wedge A$
10: $\Omega \leftarrow \Omega \oplus \gamma$
11: for $j := 1$ to $k - 1$ do
12:     $\gamma \leftarrow T \wedge r$
13:     $\gamma \leftarrow \gamma \oplus \Omega$
14:     $T \leftarrow T \wedge A$
15:     $\gamma \leftarrow \gamma \oplus T$
16:     $T \leftarrow 2\gamma$
17: end for
18: $x' \leftarrow x' \oplus T$
```



Algorithm 6. Kogge-Stone Masked Addition

```
Require: $x', y', r, s \in \{0,1\}^k$ and $n = \max(\lceil \log_2(k-1) \rceil, 1)$.
Ensure: $z'$ such that $z' = (x + y) \oplus r$, where $x = x' \oplus s$ and $y = y' \oplus r$
 1: Let $t \leftarrow \{0,1\}^k$, $u \leftarrow \{0,1\}^k$.
 2: $P' \leftarrow$ SecXor$(x', y', r)$                    ▷ $P' = (x \oplus y) \oplus s = P \oplus s$
 3: $G' \leftarrow$ SecAnd$(x', y', s, r, u)$              ▷ $G' = (x \wedge y) \oplus u = G \oplus u$
 4: $G' \leftarrow G' \oplus s$
 5: $G' \leftarrow G' \oplus u$                            ▷ $G' = (x \wedge y) \oplus s = G \oplus s$
 6: for $i := 1$ to $n - 1$ do
 7:     $H \leftarrow$ SecShift$(G', s, t, 2^{i-1})$       ▷ $H = (G \ll 2^{i-1}) \oplus t$
 8:     $U \leftarrow$ SecAnd$(P', H, s, t, u)$            ▷ $U = (P \wedge (G \ll 2^{i-1})) \oplus u$
 9:     $G' \leftarrow$ SecXor$(G', U, u)$                 ▷ $G' = ((P \wedge (G \ll 2^{i-1})) \oplus G) \oplus s$
10:     $H \leftarrow$ SecShift$(P', s, t, 2^{i-1})$       ▷ $H = (P \ll 2^{i-1}) \oplus t$
11:     $P' \leftarrow$ SecAnd$(P', H, s, t, u)$           ▷ $P' = (P \wedge (P \ll 2^{i-1})) \oplus u$
12:     $P' \leftarrow P' \oplus s$
13:     $P' \leftarrow P' \oplus u$                        ▷ $P' = (P \wedge (P \ll 2^{i-1})) \oplus s$
14: end for
15: $H \leftarrow$ SecShift$(G', s, t, 2^{n-1})$           ▷ $H = (G \ll 2^{n-1}) \oplus t$
16: $U \leftarrow$ SecAnd$(P', H, s, t, u)$                ▷ $U = (P \wedge (G \ll 2^{n-1})) \oplus u$
17: $G' \leftarrow$ SecXor$(G', U, u)$                     ▷ $G' = ((P \wedge (G \ll 2^{n-1})) \oplus G) \oplus s$
18: $z' \leftarrow$ SecXor$(y', x', s)$                    ▷ $z' = (x \oplus y) \oplus r$
19: $z' \leftarrow z' \oplus (2G')$                        ▷ $z' = (x + y) \oplus 2s \oplus r$
20: $z' \leftarrow z' \oplus 2s$                           ▷ $z' = (x + y) \oplus r$
21: return $z'$
```

Okay, back to PQC Itself

Overall Method &
An Example from Kyber

Countermeasure Design Methodology – PQC

Requires good cryptanalytic understanding of the algorithms

PQC algorithms don't have "nice group structure" that would allow such 1-step techniques to be used. About dozen different gadgets are needed.

1. Identify *all* critical variables and computations / points of attack.
2. Design consistent countermeasures for each (at least masking).
3. Validate countermeasures. Apply adversarial analysis, testing.
4. Based on analysis and new research: Improve, loop to Step 3.

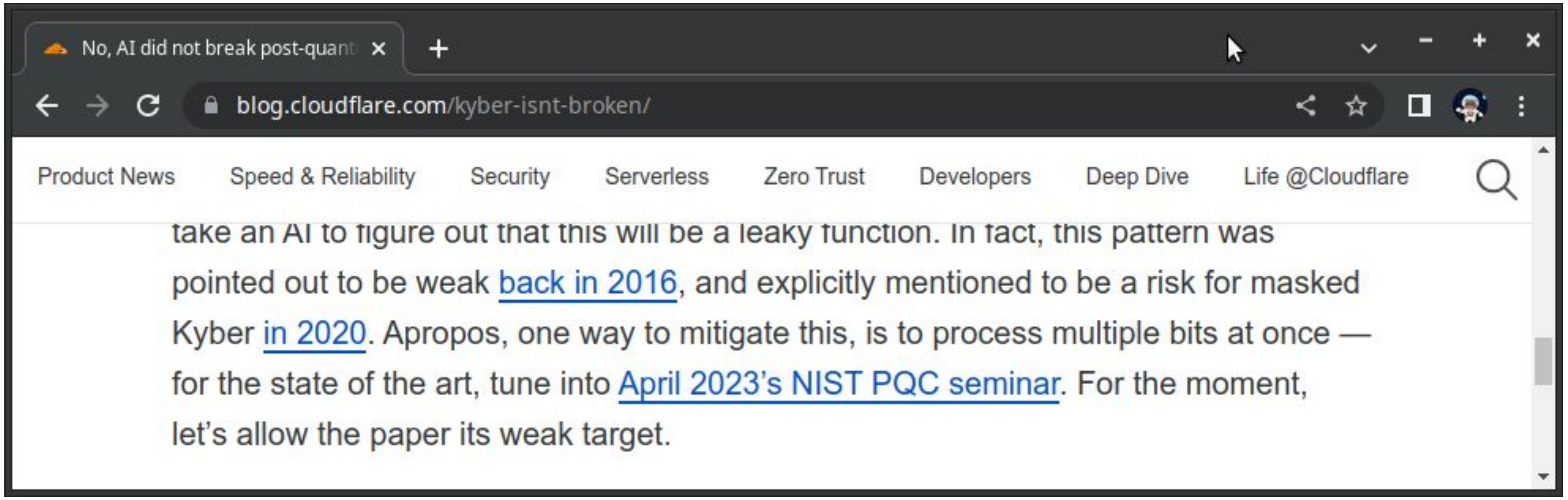
<u>State of the Art:</u> We and a few others are in the continuous iteration phase 3-4 with **Kyber** and **Dilithium**. *Note: Thus far no one (to our knowledge) has completed Step 2 for Falcon – only timing attack resistance..*

Example: Encoding Gadget in Kyber

One of dozens of required gadgets in Kyber and Dilithium

All gadgets need to be secure but this is the one whose **"mkm4" implementation** was attacked in https://ia.cr/2022/1713 (RWC '23 + lots of news) and https://ia.cr/2023/294

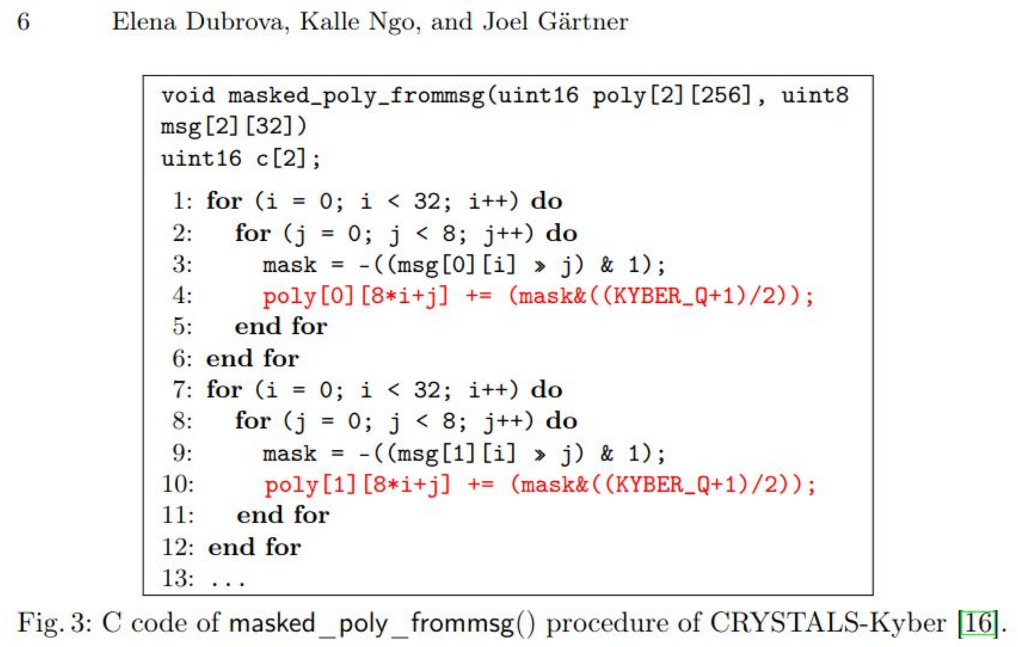
*(I was **set up** to use this example by Lejla, Stjepan, and Bas in the Cloudflare blog..! 😊)*



take an AI to figure out that this will be a leaky function. In fact, this pattern was pointed out to be weak back in 2016, and explicitly mentioned to be a risk for masked Kyber in 2020. Apropos, one way to mitigate this, is to process multiple bits at once — for the state of the art, tune into April 2023's NIST PQC seminar. For the moment, let's allow the paper its weak target.

Anyway, Where is the Bad Gadget?

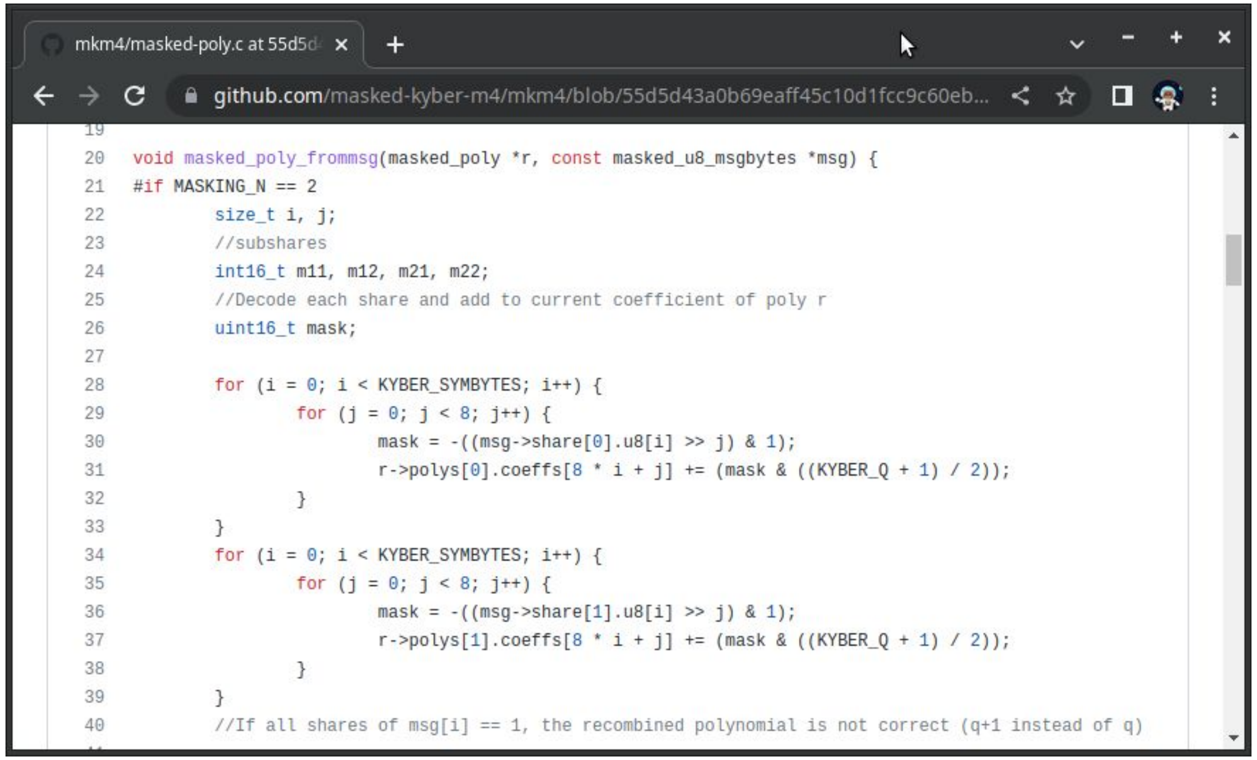
It's in an open-source implementation called "mkm4"



6 Elena Dubrova, Kalle Ngo, and Joel Gärtner

```
void masked_poly_frommsg(uint16 poly[2][256], uint8
msg[2][32])
uint16 c[2];
for (i = 0; i < 32; i++) do
    for (j = 0; j < 8; j++) do
        mask = -((msg[0][i] >> j) & 1);
        poly[0][8*i+j] += (mask&((KYBER_Q+1)/2));
    end for
end for
for (i = 0; i < 32; i++) do
    for (j = 0; j < 8; j++) do
        mask = -((msg[1][i] >> j) & 1);
        poly[1][8*i+j] += (mask&((KYBER_Q+1)/2));
    end for
end for
...
```

Fig. 3: C code of masked_poly_frommsg() procedure of CRYSTALS-Kyber [16].

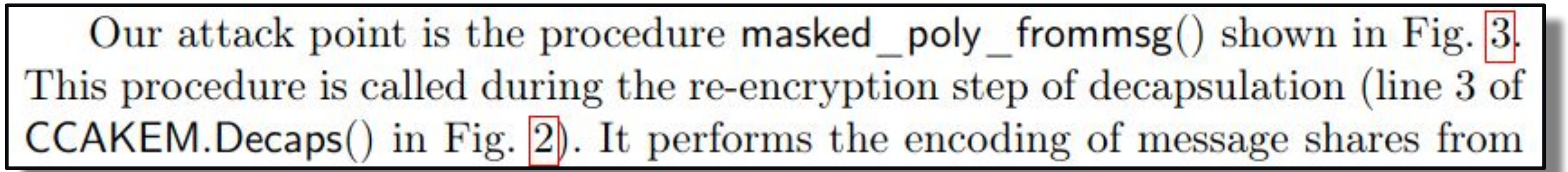


mkm4/masked-poly.c at 55d5d

github.com/masked-kyber-m4/mkm4/blob/55d5d43a0b69eaff45c10d1fcc9c60eb...

```
void masked_poly_frommsg(masked_poly *r, const masked_u8_msgbytes *msg) {
#if MASKING_N == 2
        size_t i, j;
        //subshares
        int16_t m11, m12, m21, m22;
        //Decode each share and add to current coefficient of poly r
        uint16_t mask;

        for (i = 0; i < KYBER_SYMBYTES; i++) {
                for (j = 0; j < 8; j++) {
                        mask = -((msg->share[0].u8[i] >> j) & 1);
                        r->polys[0].coeffs[8 * i + j] += (mask & ((KYBER_Q + 1) / 2));
                }
        }
        for (i = 0; i < KYBER_SYMBYTES; i++) {
                for (j = 0; j < 8; j++) {
                        mask = -((msg->share[1].u8[i] >> j) & 1);
                        r->polys[1].coeffs[8 * i + j] += (mask & ((KYBER_Q + 1) / 2));
                }
        }
        //If all shares of msg[i] == 1, the recombined polynomial is not correct (q+1 instead of q)
```



Our attack point is the procedure masked_poly_frommsg() shown in Fig. 3. This procedure is called during the re-encryption step of decapsulation (line 3 of CCAKEM.Decaps() in Fig. 2). It performs the encoding of message shares from

Kyber High-Level Encapsulation ("Alice")

Encapsulation wraps Encryption inside a *a lot of* hashing

(**CT**, **SS**) = **Kyber.CCAKEM.Enc**(**PK**):

```
1.    seed     ← random 32 bytes                     //  Seed: Unique every time.
2.    M        ← SHA3-256( seed )                    //  Hash it, "just to be sure?" 🤷
3.    ( K, R ) ← SHA3-512( M ‖ SHA3-256( PK ) )      //  Shared secret K and seed R.
4.    CT       ← Kyber.CPAPKE.Enc( PK, M, R )        //  Encrypt to create ciphertext.
5.    SS       ← SHAKE-256( K ‖ SHA3-256( CT ) )     //  Shared Secret (256 bits).
```

- CSP variables are marked in **RED**. Ciphertext **CT** is public, session key **SS** secret, etc..

The wrapper is known as the "Fujisaki-Okamoto Transform." It is essential to protect against Chosen Ciphertext Attacks (CCA) if the secret key is fixed (not ephemeral).

Kyber High-Level Decapsulation (“Bob”)

Decapsulation wraps & tests Decryption. Pretends to never fail!

```
SS = Kyber.CCAKEM.Dec( CT, SK ):

1-4.  ( S, PK, h, Z )  ← SK                                // Decode secret key.
4.    M’         ← Kyber.CPAPKE.Dec( S, CT )               // Encrypt to create ciphertext.
5.    ( K’,R’ )  ← SHA3-512( M’ ‖ SHA3-256( PK ) )         // Hash of PK is cached in “h.”
6.    CT’        ← Kyber.CPAPKE.Enc( PK, M’, R’ )          // Simulated encryption.
7.    If CT ≠ CT’ then:                                    // If re-encryption different,
10.   |   K’     ← Z                                       // .. replace key with a “fake.”
12.   SS’        ← SHAKE-256( K’ ‖ SHA3-256( CT ) )        // Shared Secret.
```

If CT is valid, one can get **SS’** without steps 6-10 – and perhaps make the decapsulation twice as fast – but this won’t be secure against (adaptive) CCA attacks. **Known Attacks!**

Highest-Level Gadgets in Kyber Encaps/Decaps

CCA Decaps needs a "more secure" CPA Enc than CCA Encaps

- Gadget 1: **Kyber.CPAPKE.Enc** with masked **M**, **R**. *(Line 4 of Kyber.CCAKEM.Enc.)*
- Gadget 2: **Kyber.CPAPKE.Dec** with masked **S**, **M'**. *(Line 4 of Kyber.CCAKEM.Dec.)*
- Gadget 3: **Kyber.CPAPKE.Enc** additionally with masked ciphertext **CT'**.
  (*Line 6 of Kyber.CCAKEM.Dec.)* That message encoding func is here!
- Gadget 4: **Secure Keccak** with Masked Input and Output.
  (A lot of instances except: SHA3-256(PK) and SHA3-256(CT).)
- Gadget 5: **Secure compare and select.** ***(****Lines 7-12 of Kyber.CCAKEM.Dec)*

[“That gadget”] Kyber Encryption (CPA)

A subroutine for both Encapsulation and Decapsulation

CT = Kyber.CPAPKE.Enc(PK, M, R):

```
1.    ( $\hat{t}$, $\rho$ ) $\leftarrow$ PK                          // Deserialize $\hat{t}$ and $\rho$ from the public key.
4.    $\hat{A}$ $\leftarrow$ gen($\rho$)                             // (Actually compute $\hat{A}$ on the fly from seed $\rho$.)
9.    r $\leftarrow$ CBD( $\eta_1$, R, 0,1,..k-1 )                  // Weights of $2\times\eta_1$ segments of SHAKE-256 output.
13.   $e_1$ $\leftarrow$ CBD( $\eta_2$, R, k,..,2k-1 )              // Error 1 is the same, but uses distribution $\eta_2$.
17.   $e_2$ $\leftarrow$ CBD( $\eta_2$, R, 2k )                     // Error 2 is a single (n=256) ring element.
18.   $\hat{r}$ $\leftarrow$ NTT( r )                                // Transform ephemeral secret.
19.   u $\leftarrow$ $NTT^{-1}$( $\hat{A}^T \circ \hat{r}$ ) + $e_1$   // First part of ciphertext: $u = A^T \cdot r + e_1$.
20.   m $\leftarrow$ $Decompress_q$(M, 1)                            // “One time pad” bits as { 0, ceil(q/2) }.
      v $\leftarrow$ $NTT^{-1}$( $\hat{t}^T \circ \hat{r}$ ) + $e_2$ + m  // Second, shorter part of ciphertext: $t^T \cdot r + e_2 + m$.
21.   return CT = ( $Compress_q(u, d_u)$, $Compress_q(v, d_v)$ )
```

[HERE!] Kyber's Encoding and "Compression"

More than slightly cumbersome bit-dropping optimization

The serialization methods mostly involve bit field packing (ignoring those for now)

Kyber also does lossy scaling to 1 ("message") and d_u, d_v bits bits: $d_{\text{bits}} \in \{ 1, 4, 5, 10, 11 \}$.

Compress$_q$: Scales a number from mod-q range $[0, q-1]$ to d-bit range $[0, 2^d-1]$.

$$\text{Compress}_q(x, d) = \lceil (2^d / q) \cdot x \rfloor \bmod 2^d.$$

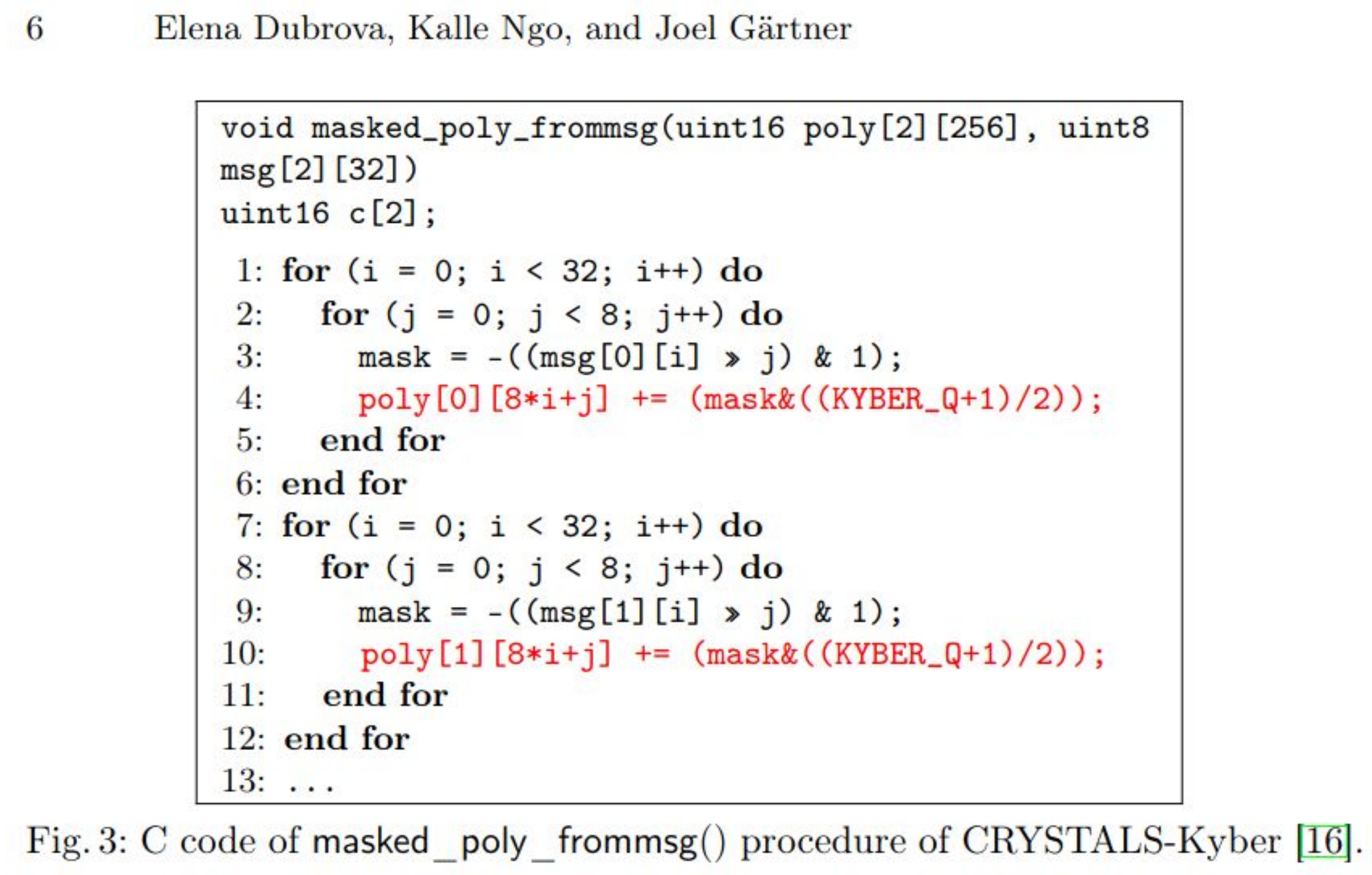
Decompress$_q$: Scales a number from d-bit range $[0, 2^d-1]$ to mod-q range $[0, q-1]$.

$$\text{Decompress}_q(x, d) = \lceil (q / 2^d) \cdot x \rfloor.$$

<u>Note</u>: $\lceil x \rfloor$ = floor(x + ½) is rounding to closest integer, with ties rounded up.

Technically it was masked..

But vulnerable to basically “1-trace Horizontal SPA”



6 Elena Dubrova, Kalle Ngo, and Joel Gärtner

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

for (i = 0; i < 32; i++) do
    for (j = 0; j < 8; j++) do
        mask = -((msg[0][i] » j) & 1);
        poly[0][8*i+j] += (mask&((KYBER_Q+1)/2));
    end for
end for
for (i = 0; i < 32; i++) do
    for (j = 0; j < 8; j++) do
        mask = -((msg[1][i] » j) & 1);
        poly[1][8*i+j] += (mask&((KYBER_Q+1)/2));
    end for
end for
...
```

Fig. 3: C code of masked_poly_frommsg() procedure of CRYSTALS-Kyber [16].

Well, I didn't do it like that

One can encode the message AFTER compression

However we can do things in different order:

There are two parts in ciphertext, canonically CT = (u,v); this is the "v" side.

$$\mathbf{CT} = (\ ..\textit{stuff}..,\ \text{Compress}_q(\ ..\textit{stuff}.. + \text{Decompress}_q(M, 1), d_v)$$

The coefficients of v are compressed from 12 bits (mod q) to $\mathbf{d_v}$ = {4,5} bits.

We create two "v" masked, compressed ciphertexts and combine them:

1. Boolean [[ctv0]]: Encryption of 256 zero bits (128 or 160 bytes/share).
2. Boolean [[ctv1]]: Encryption of 256 one bits (same but all bits refreshed).
3. Use [[M]] to select {4,5} - bit fields from [[ctv0]] and [[ctv1]] into [[ctv]].

A Peek inside a Commercial Security Module

(Often bears no resemblance to the reference implementation.)

- A coprocessor manipulates the mod q vectors with 4 coefficients per 64-bit word.

- Our hardware has a "A2A" transform that performs an intermediate conversion from (mod q) to (mod 2^N) arithmetic domain and helps with the A2B step and compression.

- There is a fast masking random number generator that is used to refresh entire representations of [[ctv0]], [[ctv1]], and the selector vector [[M]] between invocations.

- Message encoding was viewed as security-critical and is not very time-critical, hence we did it this complicated way from the beginning.

- Adversarial evaluation (**hi Timo Z!**) didn't find exploitable leakage in this particular gadget, but it is just one of dozens of possible attack points for template attacks..

The Really Complex Beast: CRYSTALS-Dilithium

NIST's Preferred PQC Signature Scheme

*"While there are multiple signature algorithms selected, NIST recommends **CRYSTALS-Dilithium** as the primary algorithm to be implemented."*

– NIST IR 8413, July 2022

For masked Dilithium I'm only aware of our proprietary hardware module and:

[V. Migliore, B. Gérard, M. Tibouchi, P.-A. Fouque. *"Masking Dilithium: Efficient Implementation and Side-Channel Evaluation."* ACNS 2019, https://ia.cr/2019/394]

[M. Azouaoui, + a lot of authors. *"Leveling Dilithium against Leakage: Revisited Sensitivity Analysis and Improved Implementations."* https://ia.cr/2022/1406]

[Identify CSPs] **Dilithium Keypair Generation**

Simplest and Fastest Operation in Dilithium

```
02.   ρ, ρ', K ← random or H(Seed)           //  Public and secret seed values.
03.   $\hat{\mathbf{A}} \leftarrow$ ExpandA(ρ)            //  Public $\hat{\mathbf{A}}$ has size $k \times l \times R_q$, derived from ρ.
04.   $\mathbf{s}_1 \leftarrow$ ExpandS(ρ', 0,2,..,l-1)    //  Secret $\mathbf{s}_1$ has size $l \times R_q$, distribution [-η, +η].
      $\mathbf{s}_2 \leftarrow$ ExpandS(ρ', l, ..,l+k-1)   //  Secret $\mathbf{s}_2$ has size $k \times R_q$, distribution [-η, +η].
05.   $\mathbf{t} \leftarrow \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2$                //  All of $\mathbf{t}$ is secure. $\mathbf{A} \cdot \mathbf{s}_1 = NTT^{-1}(\hat{\mathbf{A}} \circ NTT(\mathbf{s}_1))$.
06.   $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow$ Power2Round( $\mathbf{t}$, d )  //  Split $\mathbf{t}$; $\mathbf{t}_1$ high 13 bits, $\mathbf{t}_0$ low 10 bits.
07.   tr ← H( ρ, $\mathbf{t}_1$ )                  //  tr = SHAKE256(PK).
08.   return **PK** = ( ρ, $\mathbf{t}_1$ ), **SK** = ( ρ, K, tr, $\mathbf{s}_1$, $\mathbf{s}_2$, $\mathbf{t}_0$ )
```

- The actual secret key is just ($\mathbf{s}_1$, $\mathbf{s}_2$). The K variable is only used in non-randomized signing (where the same message and **SK** always give the same sig.)
- Note that ExpandS(ρ') deterministic sampling is only useful in testing. If one can get uniform [-η, +η] numbers (basically $\mathbb{Z}_5$ and $\mathbb{Z}_9$) directly in shares, this is better.

[Identify CSPs] **Signature Generation** ***(1 of 2)***

Create a randomized “challenge” based on the message

```
09.  $\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$          // A has size $k \times l \times R_q$, derived from ρ.
10.  $\mu \leftarrow H(tr \,||\, M)$                 // 512-bit message hash with H(PK) prefix.
11.  $\kappa \leftarrow 0,\ (\mathbf{z}, \mathbf{h}) \leftarrow \perp$             // Iteration counter κ, Iteration result.
12.  $\rho' \leftarrow$ random [ or $H(K, \mu)$ ]       // [ Use hash in deterministic signing. ]
13.  while $(\mathbf{z}, \mathbf{h}) = \perp$ do:            // — REJECTION LOOP —
14.  |  $\mathbf{y} \leftarrow \text{ExpandMask}(\rho', \kappa..)$    // $\mathbf{y}$ is $l \times R_q$ sampled from $[-\gamma_1, +\gamma_1]$.
15.  |  $\mathbf{w} \leftarrow \mathbf{A} * \mathbf{y}$                 // Compute as $\mathbf{w} = NTT^{-1}(\hat{\mathbf{A}} \circ NTT(\mathbf{y}))$.
16.  |  $\mathbf{w}_1 \leftarrow \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$      // $\mathbf{w}_1$ range is $(q-1)/2\gamma_2$ so [0,15] or [0,43].
17.  |  $\tilde{c} \leftarrow H(\boldsymbol{\mu}, \mathbf{w}_1)$              // $\tilde{c}$ is derived from message and public key.
18.  |  $c \leftarrow \text{SampleInBall}(\tilde{c})$        // $c$ is in $R_q$, has τ non-zero (±1) coefficients.
19.  |  $\mathbf{z} \leftarrow \mathbf{y} + c * \mathbf{s}_1$              // It’s better to store $NTT(\mathbf{s}_1)$ – as shares.
```

That’s the arithmetic for $\tilde{c}$ *and* **z**. *We must reject them and “goto 14” if some checks fail..*

[Identify CSPs] **Signature Generation *(2 of 2)***

Based on “Fiat-Shamir with Aborts” - Rejection Iteration

```
20.   |   $r_0 \leftarrow$ LowBits( $w - c * s_2, 2\gamma_2$ )          // Range is basically $\pm 2\gamma_2$
21.   |   if MaxAbs($z$) $\geq \gamma_1 - \beta$ or MaxAbs($r_0$) $\geq \gamma_2 - \beta$    then: $(z, h) \leftarrow \perp$ // reject
22.   |   else:
23.   |       $h \leftarrow$ MakeHint( $- c * t_0$ , $w - c * s_2 - c * t_0$, $2\gamma_2$ )    // $h \in \{0,1\}^{kN}$
24.   |       if MaxAbs($c * t_0$) $> \gamma_2$ or CountOnes($h$) $> \omega$   then: $(z, h) \leftarrow \perp$ // reject
25.   |   $\kappa \leftarrow \kappa + l$                                  // For creating fresh y in next iteration
      end while
26.   return Sig = ( $\tilde{c}$, $z$, $h$ )                          // no longer secret
```

- Protecting just the (s_1, s_2) secret itself via masking is easy; NTT in shares.
- Leaking the one-time secret y also breaks things; use masked arithmetic.
- MaxAbs and SampleInBall are very tricky to implement in masked format.
- The protected variables become non-secret (signature) after passing the check.

Dilithium Signature Verification

For completeness – Luckily doesn't involve secrets

```
{ T, F } = Verify( Sig, M, PK ):

        ( ȼ, z, h ) ← Sig                                      //  Deserialize signature.
        ( ρ, $\mathbf{t}_1$ ) ← PK                                     //  Deserialize public key.
27.     $\hat{\mathbf{A}}$  ←  ExpandA(ρ)                                   //  "Lattice" in NTT transformed domain.
28.     μ  ←  H( H(PK), M )                                 //  Prefix the message hash with H(PK).
29.     c  ←  SampleInBall(ȼ)                               //  Hash to τ non-zero (±1) coefficients.
30.     $\mathbf{w}'_1$ ←  $\mathrm{UseHint}_q(\mathbf{h}, \mathbf{A}^*\mathbf{z} - \mathbf{c}^*\mathbf{t}_1 \cdot 2^d, 2\gamma_2)$   //  Hint helps make $w'_1$ exactly matching.

31.     if  MaxAbs( z ) < $\gamma_1 - \beta$  and  ȼ = H( μ || $\mathbf{w}'_1$)  and  CountOnes(h) ≤ ω  then:
        |    return  T 👍 "Good signature"
        else:
        |    return  F 👎 "Fail!"
```

Dilithium Summary

Designing “Post-SCA” PQC Lattice Signatures!

- **A bit hard**: Dilithium is really quite complex to secure due to bit-level gadgets, and side-channel attacks/evaluation are also less mature than for Kyber.

- The randomized version is definitely more secure than the deterministic one. (As noted also in https://ia.cr/2022/1406 – along with better CSP analysis.)

- **Suggestion** (not entirely original)**:** You can defend against bad RNGs without going fully deterministic; use **ρ’** ← H(**\$**, **K**, **μ**) where **\$** is random, **K** is from the secret key, and **μ** binds it with public key and message: **μ =** H(H(pk), M).

- **Key management** (and KeyGen) also needs security: https://ia.cr/2022/1499

Turning Things Around: Masked Raccoon 🦝

Designing "Post-SCA" PQC Lattice Signatures!

- So, protecting Dilithium is hard (but doable) and masking Falcon *hard hard*! Countermeasures are becoming more expensive as attacks develop.

- A current research area: Designing PQC algorithms for side-channel security. (In similar way as SHA-3 is "Post-SCA" when SHA-2 clearly is not.)

- A new lattice-based signature scheme that does not require "hard gadgets" (masked SHAKE, A2B/B2A etc). Masking is O(d log d) - We can run it at d=32!

  [R. del Pino, T. Prest, M. Rossi, M.-J. O. Saarinen. ***"High-Order Masking of Lattice Signatures in Quasilinear Time."*** Proc. IEEE Security & Privacy 2023.]

Reading Traces

On Testing and Certification

How do we test SCA-secure things in design?

Physical testing is just the final step

1. **Design secure gadgets.** Ideally the gadgets should be *provably secure* in appropriate model (t-probing model, noisy leakage model), perhaps have SNI (Strong Non-Interference) composability, etc.

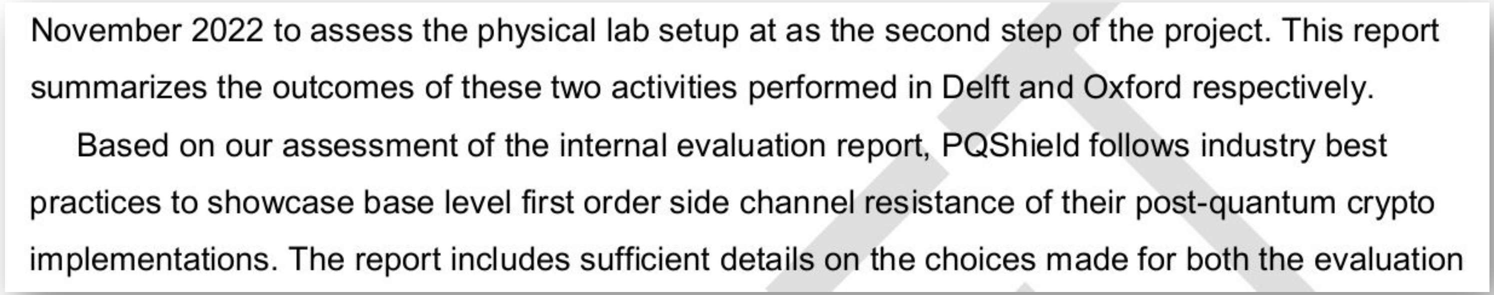
2. **Leakage simulation** in microcontroller and **Pre-Silicon** for hardware.
   - The models range from very simple & fast – based on bit toggling – to extremely advanced "3D" physical models.
   - Attack modeling with leakage simulation can be very advanced.

3. **Physical verification / Sign-Off.** Oscilloscopes running *leakage assessment* tests like TVLA. Mainly to find implementation-specific effects.

Third Party Evaluation

Ever-Continuing Process – Developing Industry Best Practices

We're working with Riscure (a well-known 3rd party security testing laboratory) to evaluate our testing methods and the reports issued to semiconductor customers.



November 2022 to assess the physical lab setup at as the second step of the project. This report summarizes the outcomes of these two activities performed in Delft and Oxford respectively.

Based on our assessment of the internal evaluation report, PQShield follows industry best practices to showcase base level first order side channel resistance of their post-quantum crypto implementations. The report includes sufficient details on the choices made for both the evaluation

"Based on our assessment of the internal evaluation report, PQShield follows industry best practices to showcase base level first order side channel resistance of their post-quantum crypto implementations"

"In conclusion, the test methods PQShield uses for gaining a base level assurance on the side channel attack resistance of the implementations in a continuous integration environment is logical and follows industry best practices"

(But methodology needs to be continuously developed.)

Certification: FIPS 140-3 vs. Common Criteria

Standardized Checks vs. Penetration Testing

- **FIPS 140-3** ("Security Requirements for Cryptographic Modules") Mostly a checklist / functional testing approach. Levels 3 and 4 mandate *"non-invasive attack mitigation"* testing *"if claimed."*

- **Common Criteria (CC)** can mean many things! High-assurance **Protection Profiles (PP)** contain **AVA_VAN.4** or **.5** (Advanced) methodical vulnerability analysis with *"attack potential"* scores.

- **NSS (U.S. DoD / IC) NIAP** also defines Common Criteria Protection Profiles, but borrows many things from FIPS testing.

Common Criteria: AVA_VAN.5

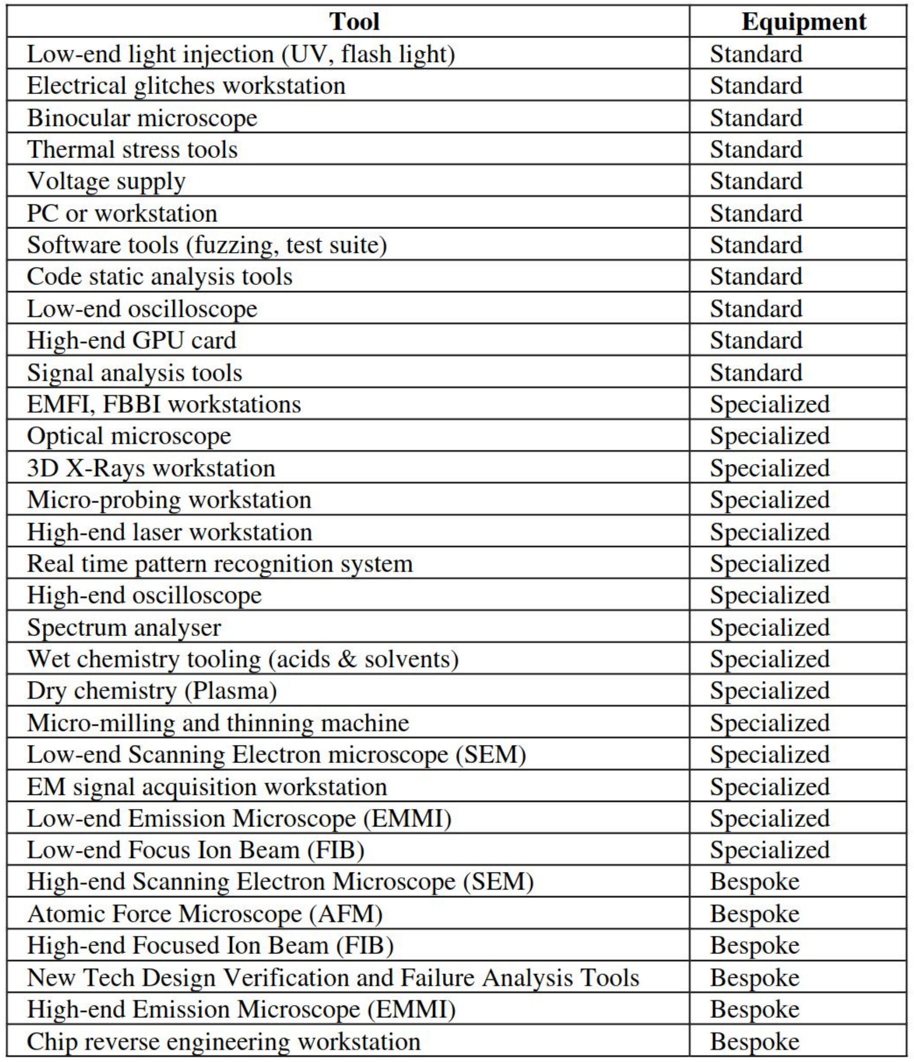
Evaluation of “High Attack Potential”

- **Score-based system.** High attack potential (well-resourced) lab spends 1-3 months, assigns an score (≅$ cost) on attack: *required time, expertise, knowledge/access of TOE, equipment, open samples.*

- Covers things like (machine learning) template attacks, but is agnostic to PQC vs Classical !

- Practical: Aims at **key recovery** or similar break.

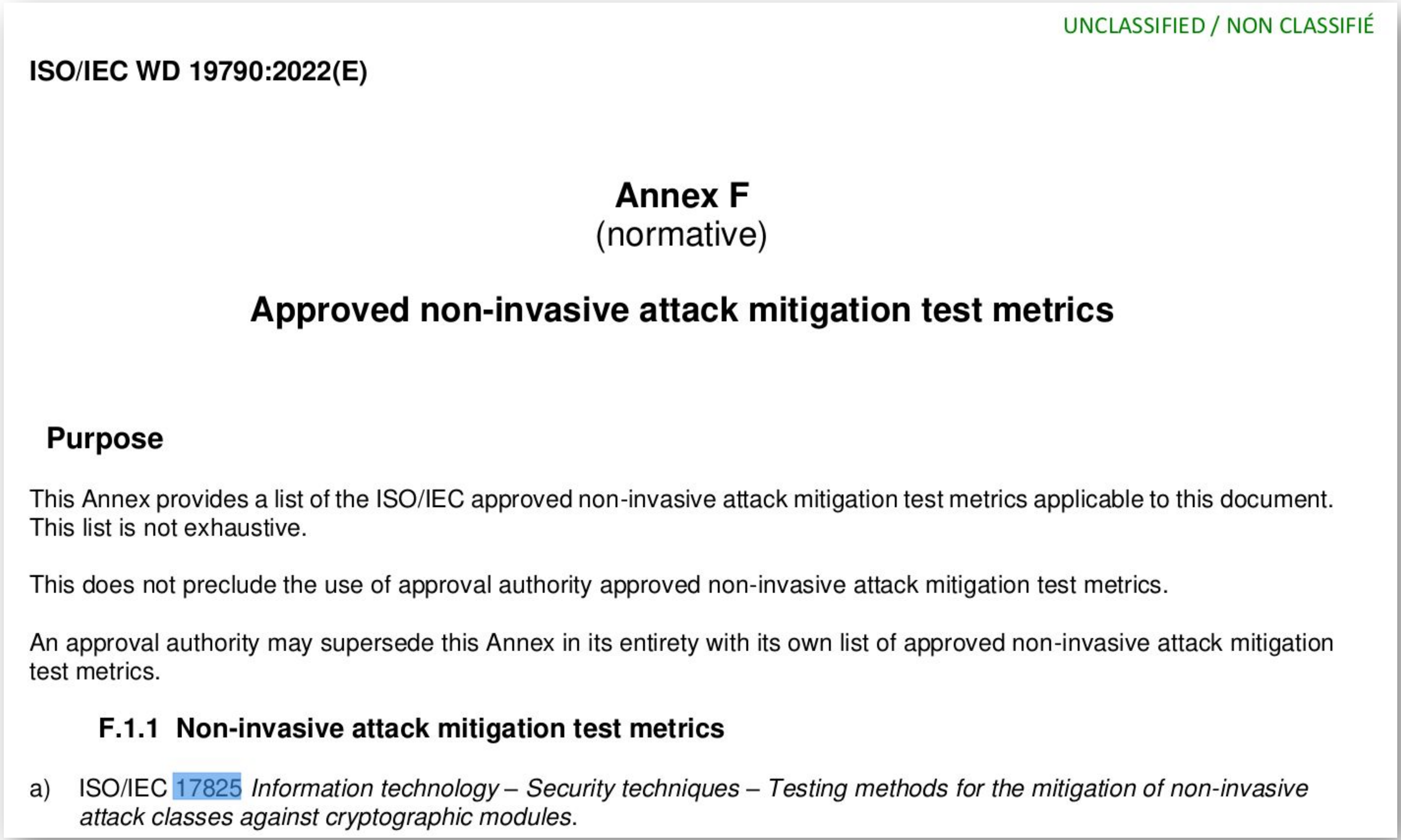
<u>Used with Smart Cards and similar devices:</u>
https://www.sogis.eu/documents/cc/domains/sc/JIL-Application-of-Attack-Potential-to-Smartcards-v3.2.pdf



Tool	Equipment
Low-end light injection (UV, flash light)	Standard
Electrical glitches workstation	Standard
Binocular microscope	Standard
Thermal stress tools	Standard
Voltage supply	Standard
PC or workstation	Standard
Software tools (fuzzing, test suite)	Standard
Code static analysis tools	Standard
Low-end oscilloscope	Standard
High-end GPU card	Standard
Signal analysis tools	Standard
EMFI, FBBI workstations	Specialized
Optical microscope	Specialized
3D X-Rays workstation	Specialized
Micro-probing workstation	Specialized
High-end laser workstation	Specialized
Real time pattern recognition system	Specialized
High-end oscilloscope	Specialized
Spectrum analyser	Specialized
Wet chemistry tooling (acids & solvents)	Specialized
Dry chemistry (Plasma)	Specialized
Micro-milling and thinning machine	Specialized
Low-end Scanning Electron microscope (SEM)	Specialized
EM signal acquisition workstation	Specialized
Low-end Emission Microscope (EMMI)	Specialized
Low-end Focus Ion Beam (FIB)	Specialized
High-end Scanning Electron Microscope (SEM)	Bespoke
Atomic Force Microscope (AFM)	Bespoke
High-end Focused Ion Beam (FIB)	Bespoke
New Tech Design Verification and Failure Analysis Tools	Bespoke
High-end Emission Microscope (EMMI)	Bespoke
Chip reverse engineering workstation	Bespoke

Table 10: Categorisation of Tools

SP 800-140Fr1 & New ISO 19790 ⟶ ISO 17825



UNCLASSIFIED / NON CLASSIFIÉ

ISO/IEC WD 19790:2022(E)

Annex F
(normative)

Approved non-invasive attack mitigation test metrics

Purpose

This Annex provides a list of the ISO/IEC approved non-invasive attack mitigation test metrics applicable to this document. This list is not exhaustive.

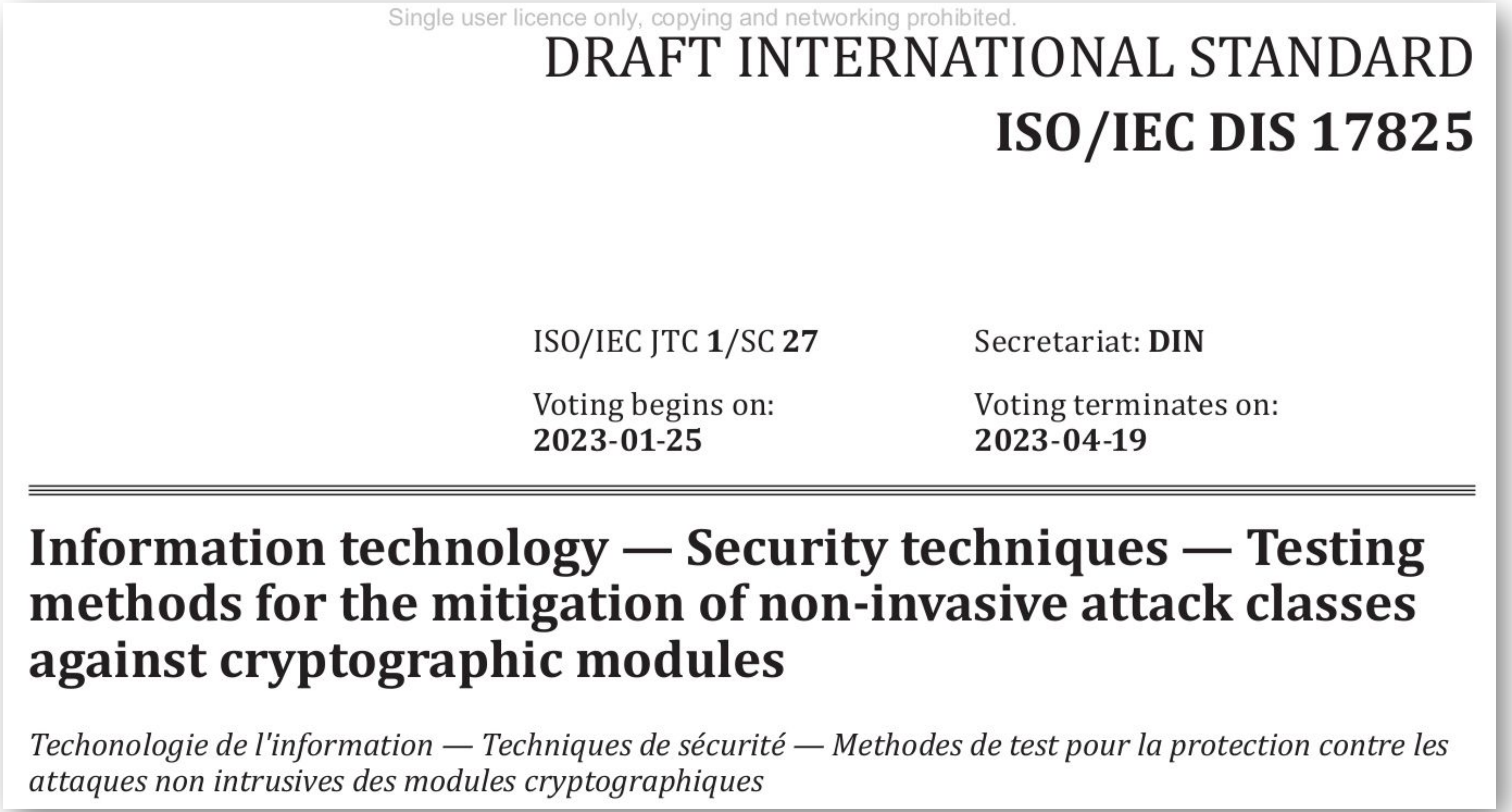
This does not preclude the use of approval authority approved non-invasive attack mitigation test metrics.

An approval authority may supersede this Annex in its entirety with its own list of approved non-invasive attack mitigation test metrics.

F.1.1 Non-invasive attack mitigation test metrics

a) ISO/IEC 17825 *Information technology – Security techniques – Testing methods for the mitigation of non-invasive attack classes against cryptographic modules.*

New ISO 17825 (Nothing specifically on PQC)



DRAFT INTERNATIONAL STANDARD

ISO/IEC DIS 17825

ISO/IEC JTC **1**/SC **27** Secretariat: **DIN**

Voting begins on:
2023-01-25

Voting terminates on:
2023-04-19

Information technology — Security techniques — Testing methods for the mitigation of non-invasive attack classes against cryptographic modules

Techonologie de l'information — Techniques de sécurité — Methodes de test pour la protection contre les attaques non intrusives des modules cryptographiques

ISO 17825 Leakage Analysis Scenario

DPA and DEMA: Power and Electromagnetic Emission Traces

- **Standard attack setting**: Tester can set inputs to the module at the IO boundary (API). Can choose inputs and synchronize to the start of the operation.

- **Oscilloscope measures power** (or electromagnetic emissions) of operations at high precision, ≥1 samples per cycle, thousands of times. Results are "traces."

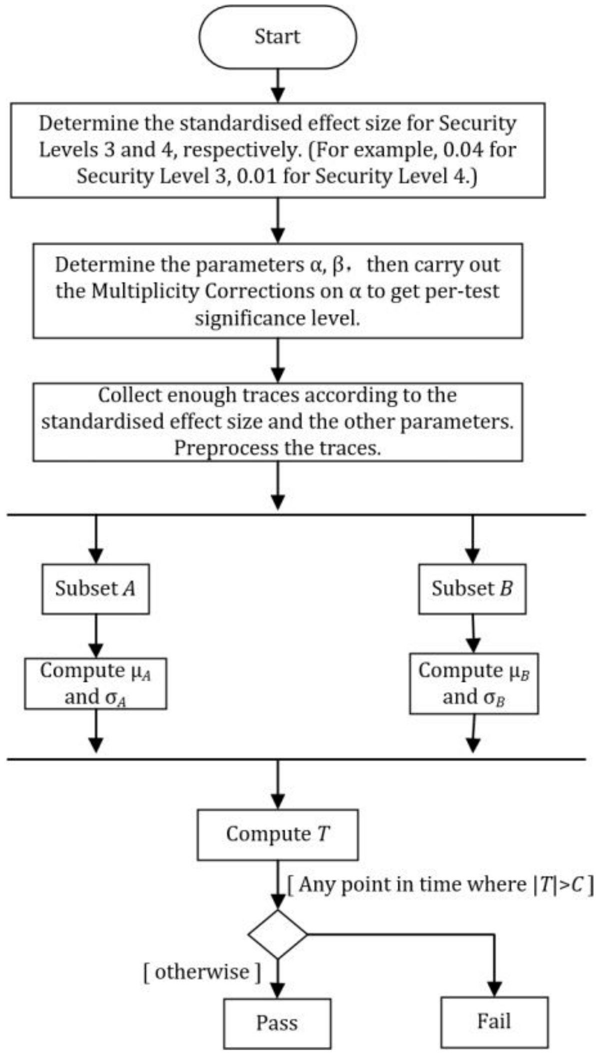
- **Traces are analyzed** to detect leakage. In leakage analysis the analyst can know or choose keys: Is looking for correlations between keys and and the traces.

- **Statistical analysis of significance. "TVLA"** PASS/FAIL metric (no key recovery).

SCA Test Automation for Post-Quantum Crypto

Detects “leakage” – no key recovery (easily False Positives)

- ISO 17825 has a “general statistical test procedure.”
- The current version of these tests create data subsets A and B of measurements (e.g., trace waveforms) with the IUT.
- But the trace sets A and B need input test vectors!
- **Example**: Set A may use a fixed bit value in a CSP, while measurements in set B use random CSP values.
- If the A/B measurement sets can be distinguished from each other – with the Welch t-test with high enough statistical confidence – this is taken as evidence of CSP leakage.


Start
Determine the standardised effect size for Security Levels 3 and 4, respectively. (For example, 0.04 for Security Level 3, 0.01 for Security Level 4.)
Determine the parameters α, β, then carry out the Multiplicity Corrections on α to get per-test significance level.
Collect enough traces according to the standardised effect size and the other parameters. Preprocess the traces.
Subset A
Subset B
Compute μ_A and σ_A
Compute μ_B and σ_B
Compute T
[Any point in time where $|T|>C$]
[otherwise]
Pass
Fail

Automatic: PQC SCA Continuous Integration

Spring 2022: CI starts running (photo of an early set-up in Oxford)

Test Automation: Construction of A and B sets

Fixed vs Random (“FIX”) and A/B Classification (“ABC”)

1. **Fixed vs Random** (non-specific t-test) can be used in “live” testing:
   - Trace set A: Fixed CSP for every trace.
   - Trace set B: New random CSP secret for each trace.

2. **A/B Categorization** works with capture-then-analyze flow:
   - Records traces with detailed test vector metadata; CSPs are known in analysis.
   - Traces are categorized *after capture* to A and B sets based on CSP selection criteria, Examples: a specific internal CSP variable or secret key bit, “plaintext checking” bit.
   - The same trace data can be categorized to A and B in a number of different ways.

In both cases: Set A and Set B statistically differentiable with t-test = **FAIL**.

TVLA: How to read a "T-Trace" like this?

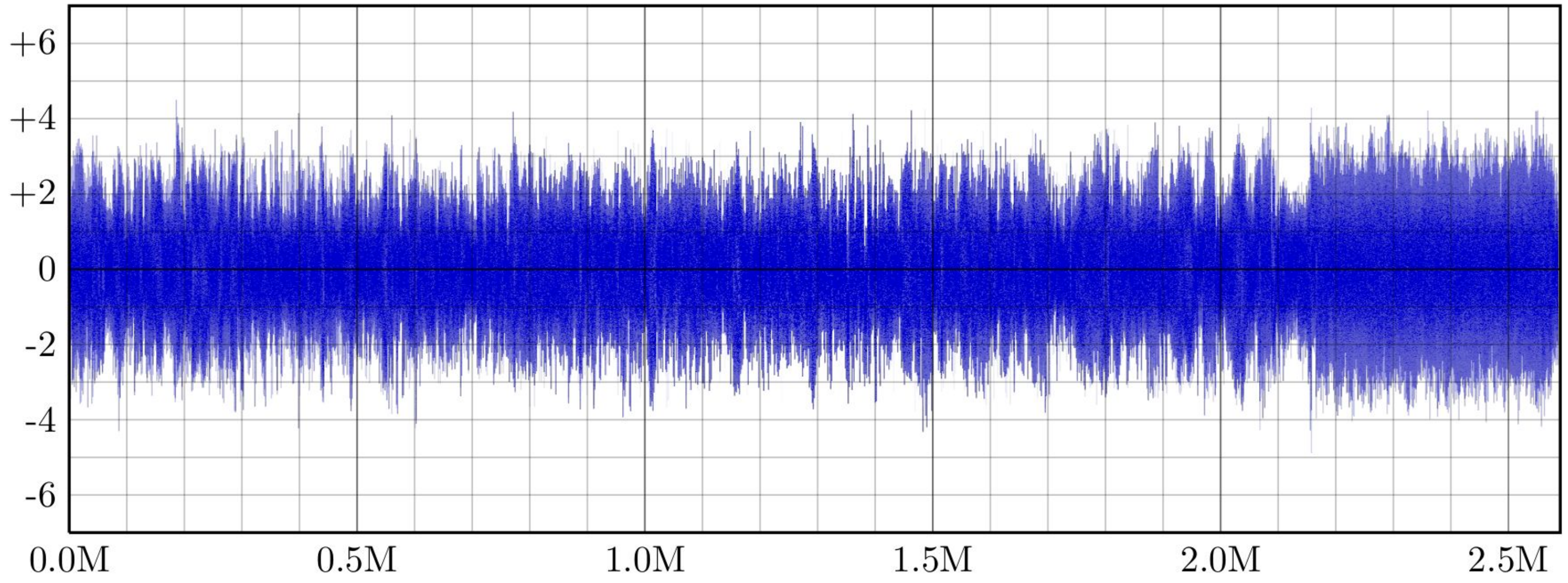
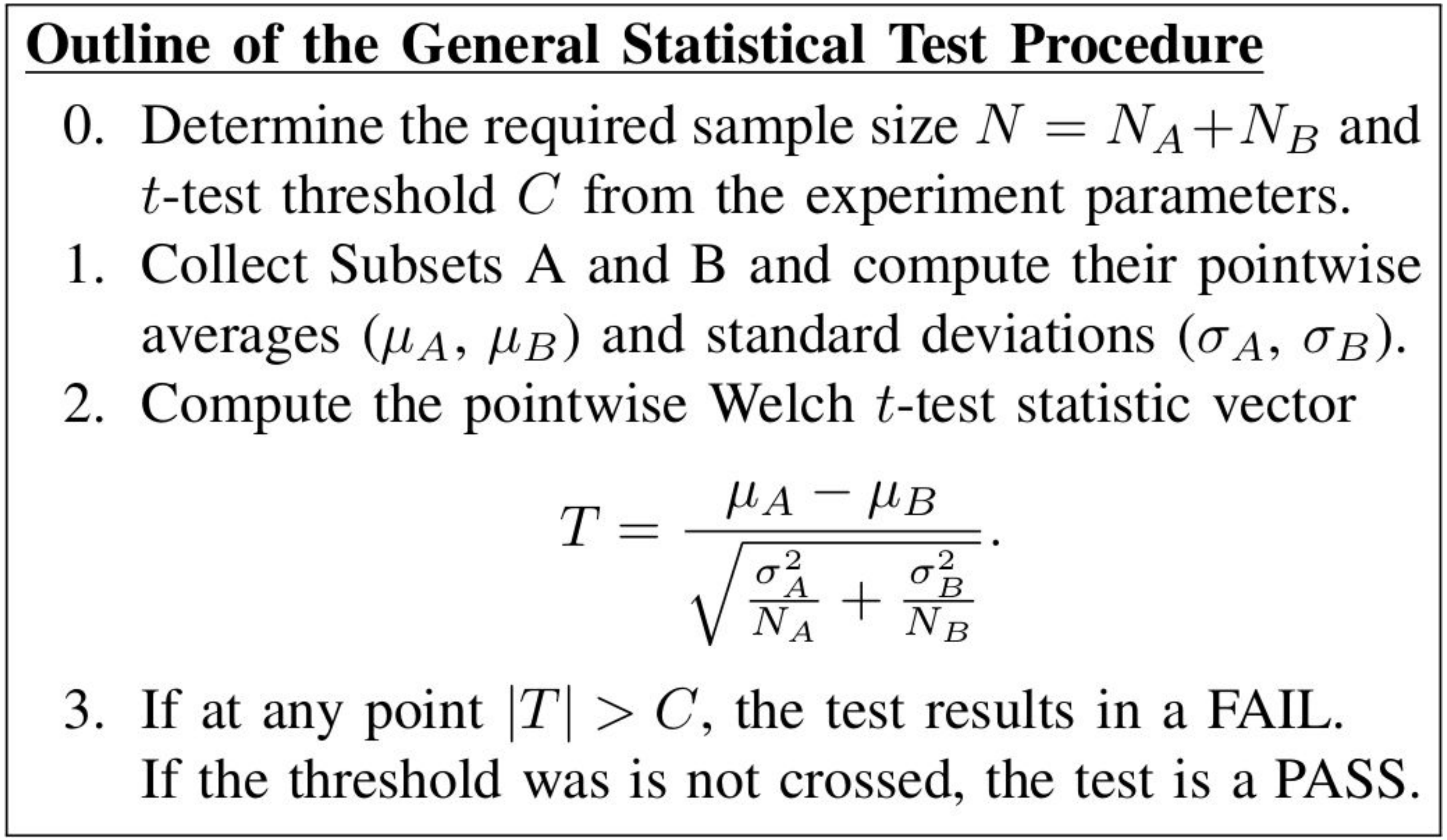


Figure 1: No secret key leakage was detected in a 200,000-trace leakage assessment of Raccoon-128 ($d = 2$) signature function. Each trace had 2.59×10^6 time points (one for each cycle – horizontal axis); a total of 518 billion measurements. The vertical axis has the t-statistic, which was bound by $|t| < 4.89$. This is well under the critical value $C = 6.94$.

[..reference..] **ISO 17825 “TVLA”**



Outline of the General Statistical Test Procedure

0. Determine the required sample size $N = N_A + N_B$ and t-test threshold C from the experiment parameters.
1. Collect Subsets A and B and compute their pointwise averages (μ_A, μ_B) and standard deviations (σ_A, σ_B).
2. Compute the pointwise Welch t-test statistic vector

$$T = \frac{\mu_A - \mu_B}{\sqrt{\frac{\sigma_A^2}{N_A} + \frac{\sigma_B^2}{N_B}}}.$$

3. If at any point $|T| > C$, the test results in a FAIL. If the threshold was is not crossed, the test is a PASS.

NIST PQC Side-Channel Security: Summary

Doable, but not default in hardware implementations

1. Core MLWE (**t** := **A** · [[s]] + [[e]]) is “simple” (like RSA or ECC) but Kyber and Dilithium countermeasures are made difficult by bit-manipulation operations on secrets. Kyber and Dilithium are getting relatively mature, Falcon is TA only.

2. **Common Criteria** methodology **AVA_VAN** can be already applied to PQC.
   - *Covers advanced attacks against it (e.g. ML assisted template attacks).*
   - *Assigns a cost to attack. Requires a “high-attack potential” lab + personnel.*

3. **FIPS 140-3 / ISO 18725** non-invasive mitigation testing can be **automated**:
   - *Leakage assessment PASS/FAIL verifies the existence of mitigations in every component that handles CSPs: Keygen, Key Export, Import, Encaps, Decaps, Sign.*

Thanks for listening!

Q & A

[..reference..] **Dilithium Algorithm Parameters**

A Signature Algorithm based on MLWE and SIS

- Coefficients / elements work in $\mathbb{Z}_q$ with $q = 8380417 = 2^{23} - 2^{13} + 1$ fitting a 23 bits.
- Ring again is of type $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ with $n=256$. NTT arithmetic is used.
- **A** has two dimensions: **k** and **l**, so the total dimension is $\mathbf{k} \times \mathbf{l} \times \mathbf{n}$.
- Public key compression (bit dropping): $d = 13$ bits.
- Challenge distribution has τ non-zero ± 1 coefficients and $(n-\tau)$ zero coefficients.
- The secret key distribution is *uniform* but in very short range $[-\eta, +\eta]$.
- Uniform **y** sampling range $[-\gamma_1, +\gamma_1]$ and low-order rounding range is $[-\gamma_2, +\gamma_2]$.
- Furthermore we have rejection bounds β (for signature) and ω (for carry hint **h**).

Parameter Set	(k, l)	τ	η	γ_1	$(q-1)/\gamma_2$	β	ω	Reps	Classic	Quant
Dilithium 2:	(4, 4)	39	2	2^{17}	88	78	88	4.3	2^{123}	2^{112}
Dilithium 3:	(6, 5)	49	4	2^{19}	32	196	55	5.1	2^{182}	2^{165}
Dilithium 5:	(8, 7)	60	2	2^{19}	32	120	75	4.0	2^{252}	2^{229}