Worcester Polytechnic Institute

An Open-Source Reference for Reproducible Side-Channel Assessment of Post-Quantum Standards

Dillibabu Shanmugam Patrick Schaumont

OSCAR Workshop 2026

Vernam Lab, Worcester Polytechnic Institute, USA

Why post-quantum, why now

Why now: it is now mandated (US & EU)

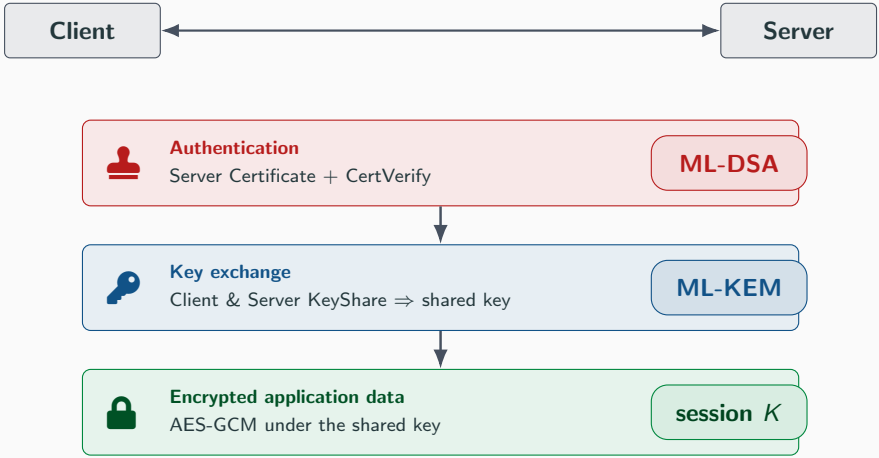
White House Executive Order

"Securing the Nation Against Advanced Cryptographic Attacks"

June 22, 2026

- PQC **key establishment** by **Dec 31, 2030**
- PQC **digital signatures** by **Dec 31, 2031**
- EU **CRA** + PQC roadmap: high-risk by **end-2030**

Where PQC lands: the TLS handshake



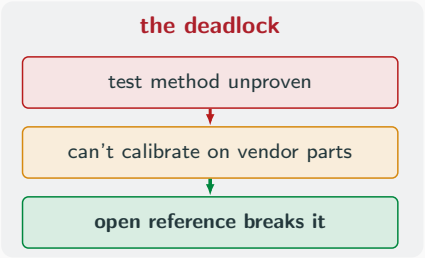
ML-KEM establishes the session key (confidentiality)

ML-DSA authenticates the server (authenticity)

The gap: the algorithm is ready, its test is not

- FIPS 203/204 prove the **algorithm**, not the **implementation**
- No **standard** for testing PQC implementations
- **ISO/IEC 17825** covers (AES, RSA, ECC): **no PQC**
- Saarinen's PQC draft: **Promising; unproven on open silicon**

Markku-Juhani O. Saarinen, "WiP: Applicability of ISO Standard Side-Channel Leakage Tests to NIST Post-Quantum Cryptography," IEEE HOST 2022 (ePrint 2022/229).



From challenge to solution

Challenge
no tested PQC silicon

One coprocessor

- ML-KEM-1024 & ML-DSA-44
- protected & unprotected
- one RTL

Challenge
vendor-locked flow

Open backend

- FPGA & ASIC
- no vendor IP
- KAT byte-exact on silicon

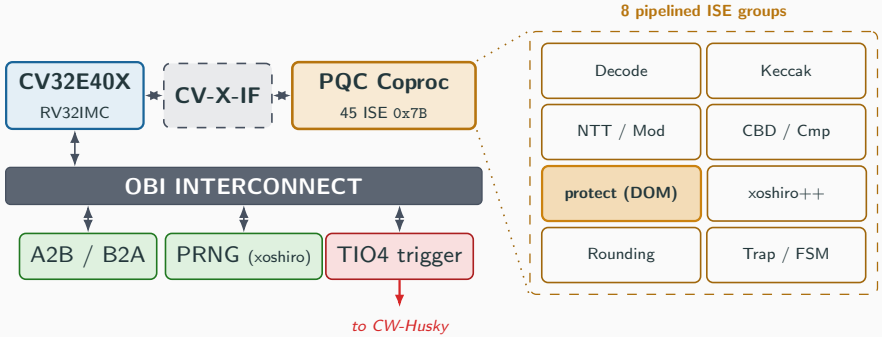
Challenge
no reproducible test

Reproducible test

- silicon & pre-silicon agree
- same trace pools

The platform: one open RTL, two silicon paths

The SoC: CV32E40X/20X + CV-X-IF + PQC coprocessor



Same bitstream for protected & unprotected

Custom instructions

`funct3` picks the group, `funct7` picks the operation; `rs1`/`rs2`=operand shares, `rd`=result

`funct3`	**Group**	**#**	**Operations (`funct7` slots)**
000	Keccak	6	`SET_C, BCOP32, SET_HI, ROL32_L/H, XOR3`
001	Mod-arith	5	`BARRETT_K, MONT_K, CADDQ, BARRETT_D, MONT_D`
010	Butterfly	4	`NTT_BFLY_CT/GS, NTT_BFLY_CT_MUL/GS_MUL`
011	CBD	12	`CBD2_1..8, CBD3_1..4`
100	Compress	6	`COMPRESS_1..5, REJ_UNIFORM`
101	ML-DSA	2	`POWER2ROUND, DECOMPOSE`
110	**Masking**	8	`SECMUL_STEP/LATCH/REUSE, MASK_REFRESH, AND_*, MASKED_AND/XOR`
111	PRNG	2	`SEED_PRNG, GET_RANDOM`
8 groups		**45**	*Mod-arith & Masking carry a `_K`/`_D` bit: same op, ML-KEM (q=3329) or ML-DSA (q=8380417)*

Protected ML-KEM-1024: operations → custom instructions → cycles

Operation	Component / module (in order)	CV-X-IF custom instruction(s)	Protected	Instances	Clock cycles
KeyGen	1. G =SHA3-512(d)	`MKECCAK SEED/LDW/PERM/RDW`	Y (nonlin)	1	≈0.40 M
	2. noise s, e =CBD(SHAKE(σ))	`MKECCAK *, B2A CONV/RD1, GET RANDOM`	Y	8192	
	3. NTT(s, e)	`MOD MONT` (+plain RV32 bfly)	Y (per-share)	8	
	4. $A\hat{s} + \hat{e}$ (basemul+add)	`MOD` (per-share), `GET RANDOM` (reshare)	Y (per-share+refresh)	16	
	5. unmask t, pack sk	(public pack)	public	0	
Encaps	1. $G(m \Vert H(pk))$	`MKECCAK *`	Y	1	≈0.46 M
	2. frommsg(m)	`B2A CONV/RD1`	Y	256	
	3. CBD(r, e_1, e_2)	`MKECCAK *, B2A CONV/RD1`	Y	9216	
	4. NTT(r)	`MOD` (+plain RV32 bfly)	Y (per-share)	4	
	5. $A^\top \hat{r}$, $\hat{t}\hat{r}$, invNTT	`MOD` (+plain RV32 bfly)	Y (per-share)	20	
	6. $+e$, +Dec(m), compress, unmask ct	`MOD, COMPRESS`	Y/public	5	
Decaps (FO)	1. masked decrypt $v - s \cdot b$ (basemul, invNTT, tomsg)	`MOD, A2B CONV/RD1` (+plain bfly)	Y	256	≈0.62 M
	2. $G(m \Vert h)$	`MKECCAK *`	Y	1	
	3. FO re-encrypt (masked CPA enc, no G)	`MKECCAK *, B2A, MOD` (+plain bfly)	Y	9 perms	
	4. **masked ct-equality (CCA.PC)**	`A2B CONV/RD1, MASKED AND`	Y	1536	
	5. rkprf $J(z \Vert c)$	`MKECCAK *` (SHAKE)	Y	∼12	
	6. masked cmov → unmask ss	`MASKED AND, GET RANDOM`	Y	256	

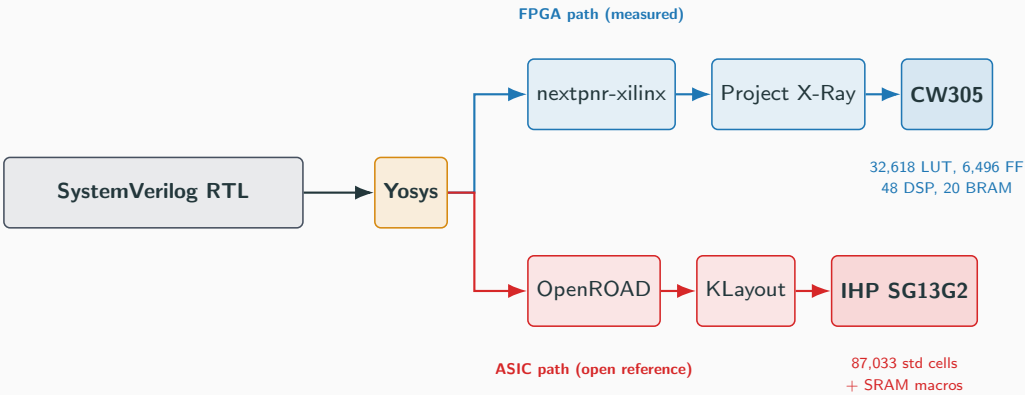
First-order protected • CV32E40X/20X • **Decaps ct-equality (CCA.PC)**

Performance: ML-KEM-1024 Decap (software / ISE / ISE+protected)

Configuration	Speed-up	LUT (post-PAR, Δ)
SW-only (no ISE)	1.0×	28,445 (baseline)
ISE (unprotected)	**13.9**×	32,618 (+14.7%)
ISE + DOM (protected)	4.0×	32,618

ML-DSA-44 follows the same pattern: 380 k → 820 k cyc protected

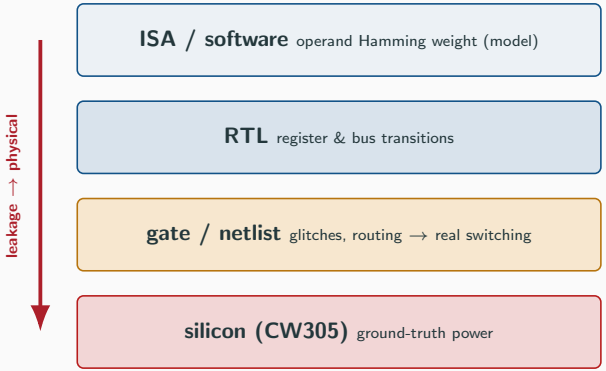
One RTL → two fully-open backends



Testing for leakage: silicon and pre-silicon

Side-channel analysis: leakage lives below the algorithm

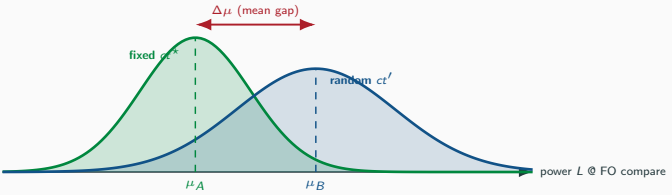
Side channel: a chip's power can leak its secret to anyone with physical access



lower layers are **harder to hide**

masking can still **break at the gate level**

What TVLA measures: the fixed-vs-random leakage gap

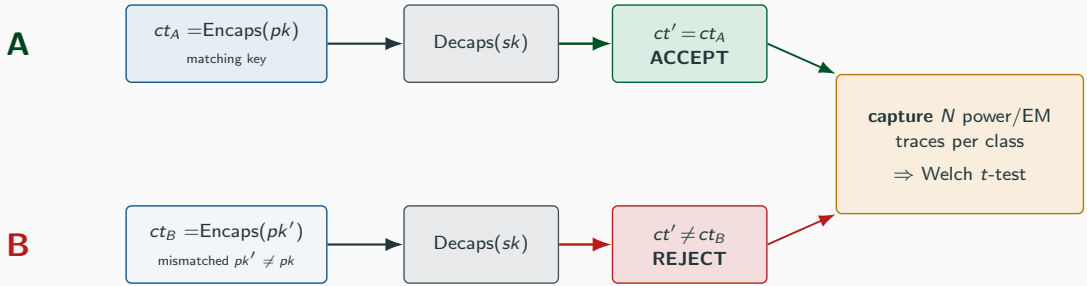


Is the mean gap real?

$$t = \frac{\Delta\mu}{\sqrt{\frac{s_A^2}{n_A} + \frac{s_B^2}{n_B}}}$$

- $|t| > 4.5$: curves **separate** $\Rightarrow$ leak
- $|t| < 4.5$: curves **overlap** $\Rightarrow$ safe

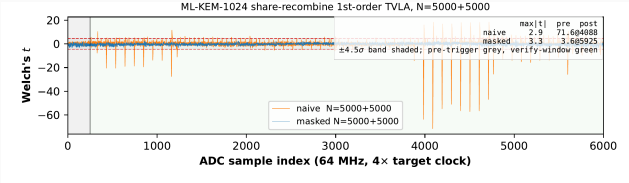
The two test classes: valid vs. invalid ciphertext



Silicon result: protection suppresses leakage $\geq 23\times$

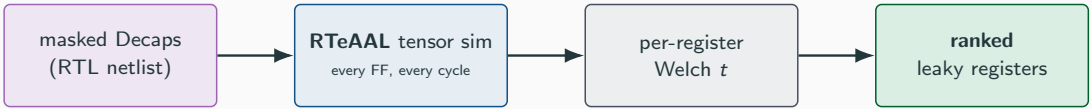
Diagnostic	Unprotected	Protected	
1st-order $\lvert t \rvert$	**71.6σ**	**3.60σ**	✓
2nd-order $\lvert t \rvert$	53.8σ	4.01σ	✓

- **$\geq 23\times$** suppression, same firmware
- follows ISO 17825 minimum-N (10,000)



Pre-silicon: RTeAAL tensor simulation

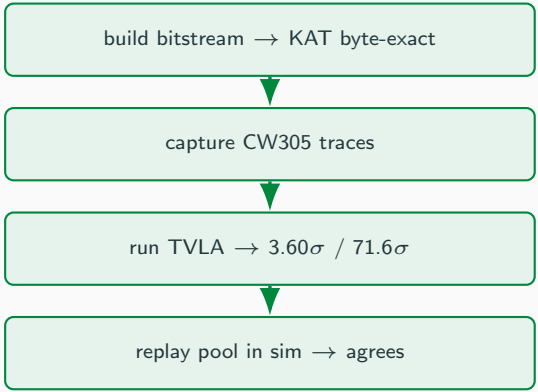
Why: silicon gives *one* verdict; RTeAAL says *which* RTL register leaks, before tape-out.



same fixed / random
CCA.PC pools as CW305

♻ Reproducible: the whole stack, one command

- Released: RTL, firmware, bitstreams, traces, scripts
- Every `.bit` byte-identical to an open rebuild



Conclusion

To our knowledge the first to combine open RTL, open FPGA & ASIC backends, and reproducible PQC TVLA:

- **One RTL**: both primitives, protected & unprotected
- **Open EDA**: FPGA and ASIC, no vendor IP
- **Reproducible test**: silicon & pre-silicon agree

The EO sets the deadline

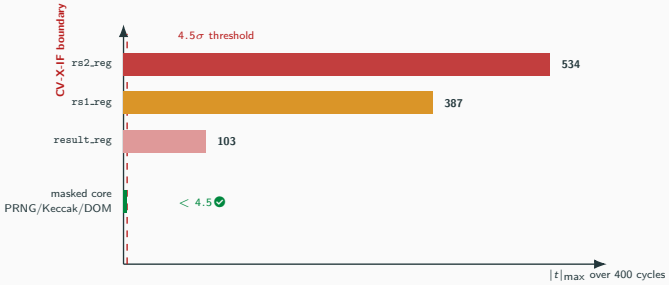
Fork it. Break it. Standardize it.

Scan for the open-source release

Secure-Embedded-Systems

Thank you. Questions?

Pre-silicon result: which register, down to the bit



Per-register Welch t (N=10k): boundary registers leak, masked core does not

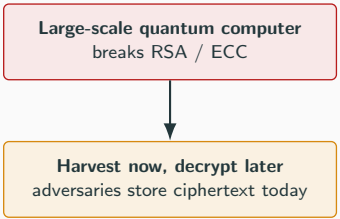
Top leaking bit (RTeAAL)

`i_sca_pqc.rs2_reg[17]`

$|t| = 534.63\,\sigma$ at cycle 137, N=10k

- one **register bit**, not a window
- same pools **agree with CW305**

The threat the order names



- Today's encrypted traffic is **being recorded**
- A future quantum computer **decrypts it retroactively**
- **Lattice PQC** (ML-KEM, ML-DSA) resists both

IND-CPA vs. IND-CCA (high level)

IND-CPA: chosen *plaintext*

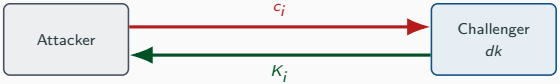
- attacker may **encrypt**
- **cannot** query decapsulation
- the inner PKE (`K-PKE`) achieves this

IND-CCA: chosen *ciphertext*

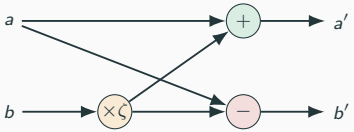
- attacker **may query Decaps** on any $c \neq c^\star$
- the realistic model: a server *is* an oracle

IND-CPA: encrypt only

IND-CCA: + Decaps oracle



Custom instruction: one NTT butterfly



$a' = a + \zeta b, \quad b' = a - \zeta b \pmod{q}$

Cooley–Tukey butterfly, twiddle ζ

Software (plain RV32)

≈40 cyc (model, ∼10 ops)

One custom instruction

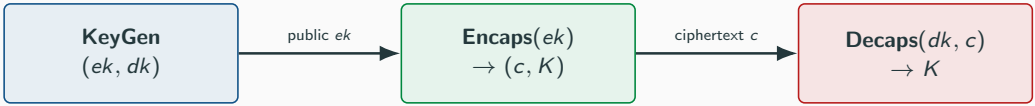
12 cyc (measured, CW305)

≈3× fewer cycles (model vs measured)

Full NTT = **896** butterflies: ≈36k → **≈12k** cycles

`NTT_BFLY_CT`: `rs1`= $a|b$, `rs2`= ζ, `rd`= $a'|b'$

Background: what a KEM does

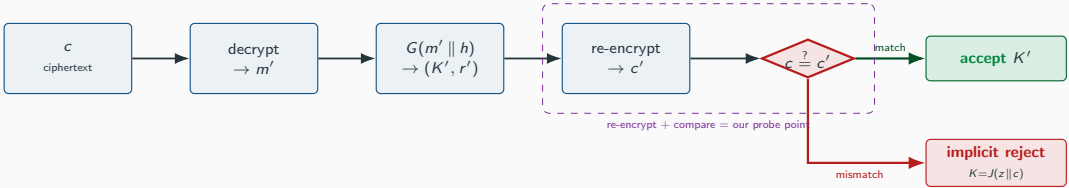


both sides share the same key K

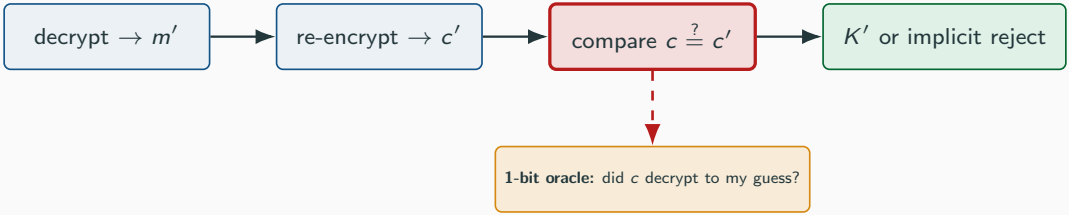
In TLS: Encaps on the client, **Decaps on the server key** The secret to protect lives *inside* Decaps

The FO transform: how ML-KEM becomes IND-CCA

K-PKE (IND-CPA) ⟶ re-encrypt & compare ⟶ ML-KEM-1024 (IND-CCA)



What we test: the FO compare (PC oracle)



Unprotected: the compare is **secret-dependent**; we make it **data-oblivious**

Leak that 1 bit per chosen $c \Rightarrow$ recover the key

Backup: one custom instruction (C / asm / header)

Example: `NTT_BFLY_CT_MUL`: Cooley-Tukey butterfly with the twiddle Montgomery-multiply folded in.

1. Header macro (`sw/include/sca_pqc_ise.h`)

```
#define SCA_NTT_BFLY_CT_MUL(rd, a, packed_zb)   \
  asm volatile(".insn r 0x7b, 2, 2, %0, %1, %2" \
      : "=r"(rd) : "r"(a), "r"(packed_zb))
```

2. Assembled 32-bit R-type word (opcode `0x7B`, group `funct3`=2, slot `funct7`=2)

```
funct7=0000010 | rs2 | rs1 | funct3=010 | rd | opcode=1111011
```

3. C call site (firmware)

```
uint32_t pzb = ((uint32_t)zeta << 16) | (b & 0xffff);
uint32_t r;  SCA_NTT_BFLY_CT_MUL(r, a, pzb);
int16_t a_plus  = r & 0xffff;   /* a + t               */
int16_t a_minus = r >> 16;      /* a - t, t=mont(b*zeta) */
```