Why “Adams Bridge” Leaks: Attacking a PQC Root-of-Trust

Markku-Juhani O. Saarinen
<markku-juhani.saarinen@tuni.fi>

Tampereen yliopisto
Tampere University

CHELPIS

Loc & Date: hardwear.io USA 2025
30-May-2025 Santa Clara, CA, USA

TALK OUTLINE

Caliptra2 & Adam's Bridge

- Why we may be running into it a lot in the near future.

Faulting Signature Verify

- A secure boot bypass strategy; malformed sig + fault.

Why Adam's Bridge Leaks

- Adam's Bridge "leaks" in TVLA as it doesn't mask secret keys. Direct exploitation may be tricky.

Time permitting: Demo https://github.com/ml-dsa/abr-sim

Caliptra2 & Adam’s Bridge

WHAT ARE WE TALKING ABOUT

CALIPTRA: A SoC Root of Trust IP jointly developed and used by { AMD, Microsoft, NVIDIA, Google, .. }

👉 Secure (“measured”) boot, FW updates, attestation.

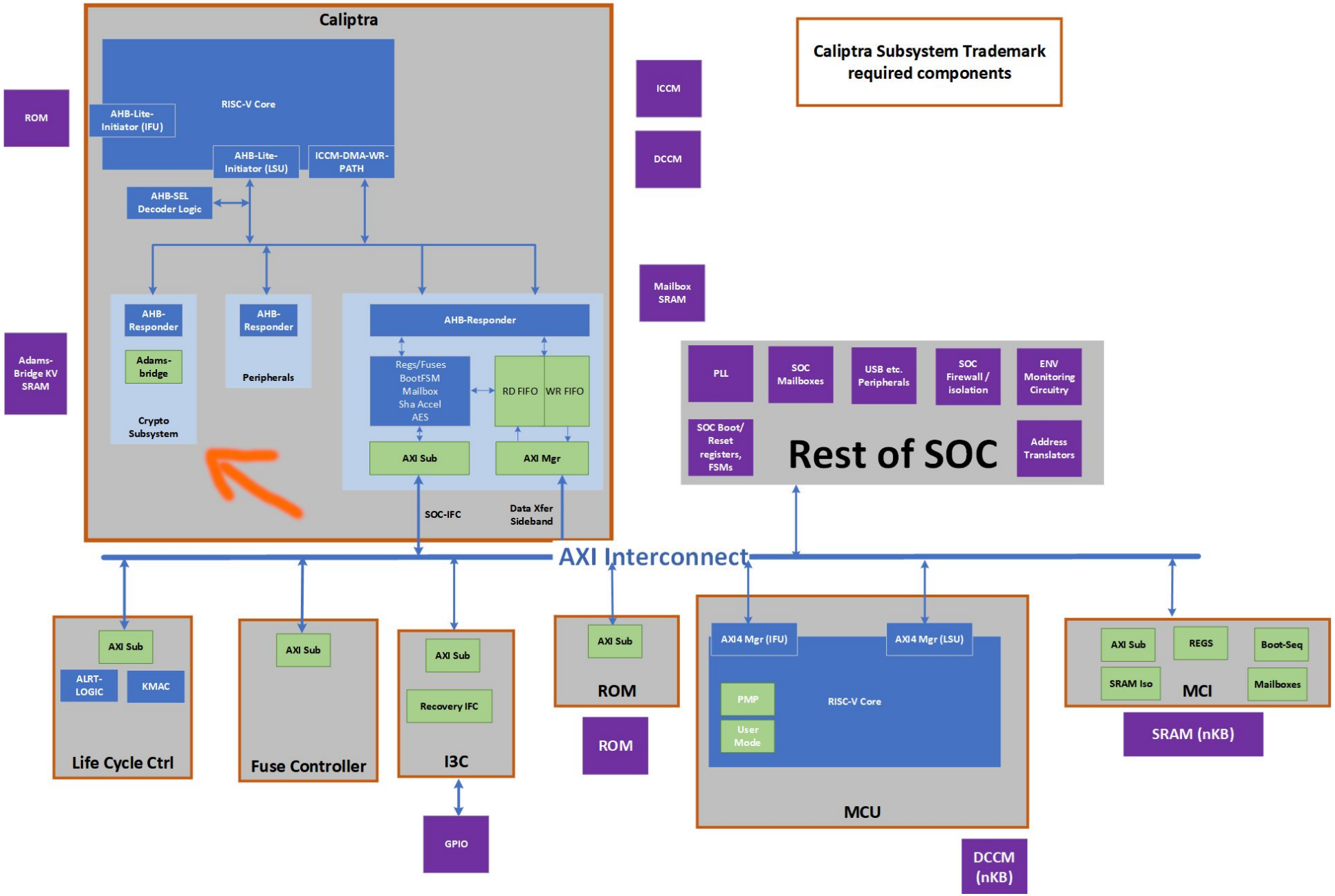
ADAM’S BRIDGE: PQC hardware accelerator for Caliptra. First version supports only ML-DSA-87 (Dilithium-5.)

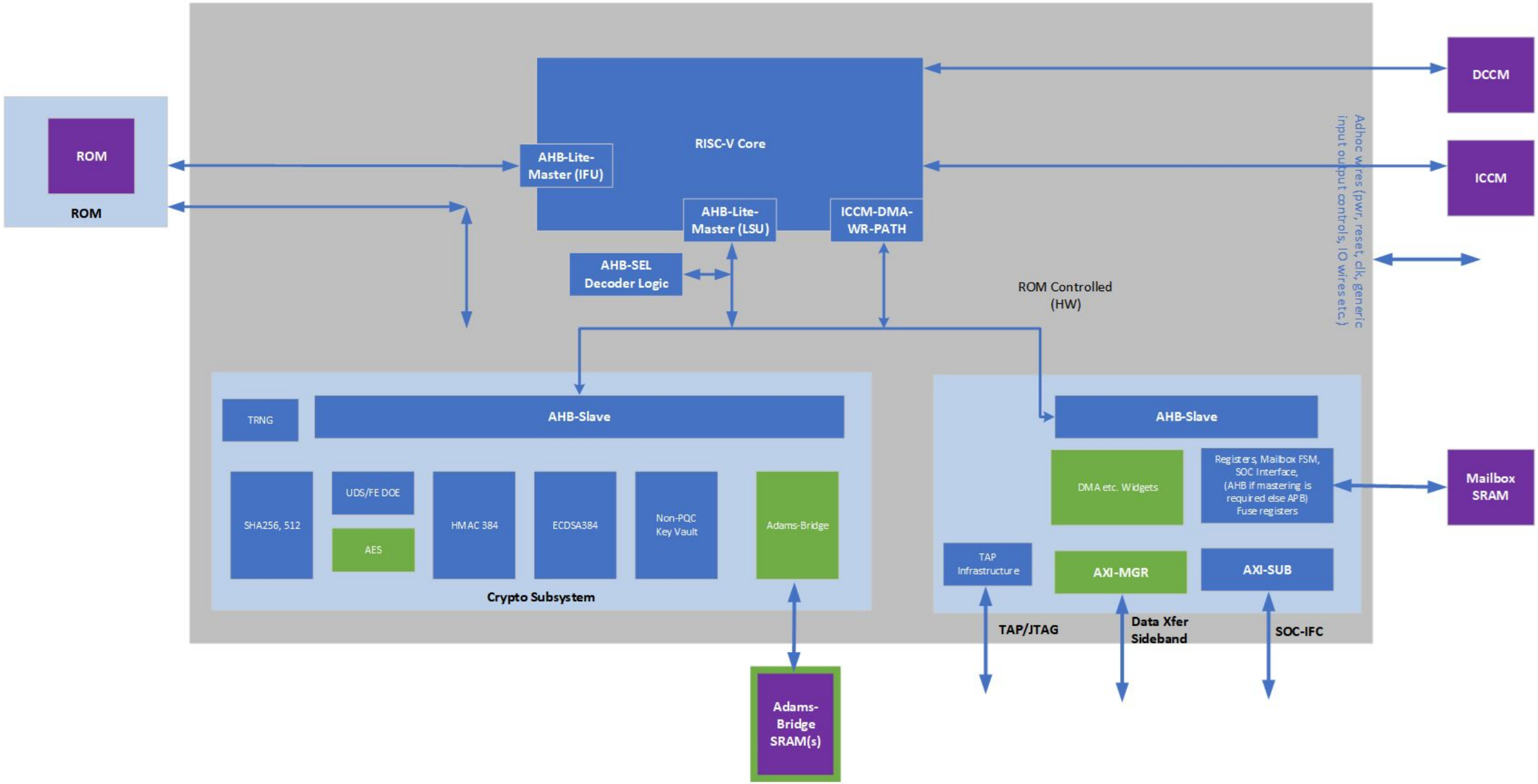
ML-DSA (Dilithium): Lattice-based signature scheme defined in FIPS 204. Post-quantum Cryptography (PQC) replacement for ECDSA in most applications.

Caliptra: Everywhere in 2026?

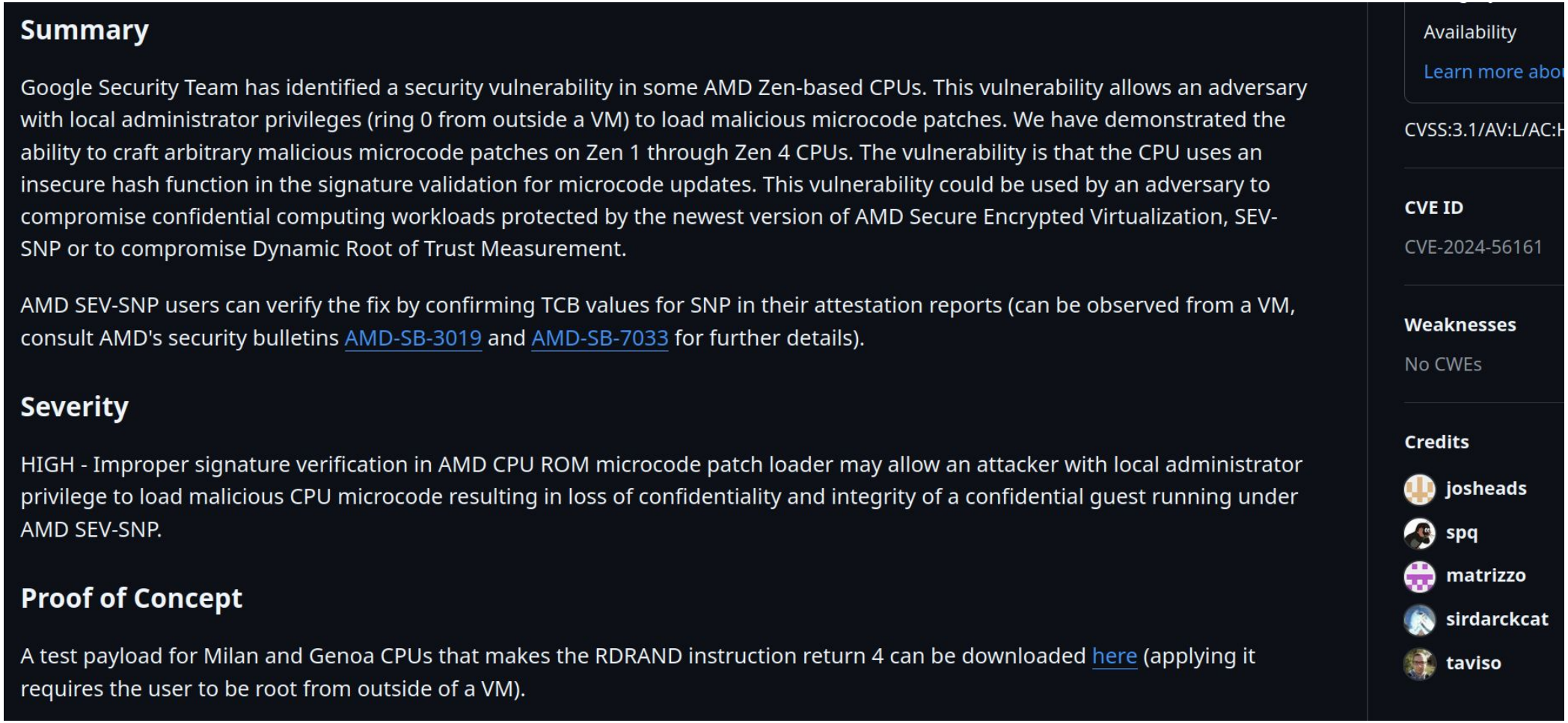
From John Traver (AMD): “An Overview of Caliptra, A Root of Trust for Measurement”

- Technical Advisory Council companies committed to deliver products with Caliptra in 2026
- Other ASIC vendors in the pipeline
- Domain experts across EDA, FW, Verification are contributing


Caliptra
RISC-V Core
AHB-Lite-Initiator (IFU)
AHB-Lite-Initiator (LSU)
ICCM-DMA-WR-PATH
AHB-SEL Decoder Logic
AHB-Responder
Adams-bridge
Crypto Subsystem
AHB-Responder
Peripherals
AHB-Responder
Regs/Fuses BootFSM Mailbox Sha Accel AES
RD FIFO
WR FIFO
AXI Sub
AXI Mgr
SOC-IFC
Data Xfer Sideband
ROM
ICCM
DCCM
Mailbox SRAM
Adams-Bridge KV SRAM
Caliptra Subsystem Trademark required components
PLL
SOC Mailboxes
USB etc. Peripherals
SOC Firewall / isolation
ENV Monitoring Circuitry
SOC Boot/ Reset registers, FSMs
Rest of SOC
Address Translators
AXI Interconnect
AXI Sub
ALRT-LOGIC
KMAC
Life Cycle Ctrl
AXI Sub
Fuse Controller
AXI Sub
Recovery IFC
I3C
GPIO
AXI Sub
ROM
ROM
AXI4 Mgr (IFU)
AXI4 Mgr (LSU)
PMP
User Mode
RISC-V Core
MCU
DCCM (nKB)
AXI Sub
REGS
Boot-Seq
SRAM Iso
MCI
Mailboxes
SRAM (nKB)


ROM
ROM
RISC-V Core
AHB-Lite-Master (IFU)
AHB-Lite-Master (LSU)
ICCM-DMA-WR-PATH
AHB-SEL Decoder Logic
ROM Controlled (HW)
DCCM
ICCM
Adhoc wires (pwr, reset, clk, generic input output controls, IO wires etc.)
TRNG
AHB-Slave
SHA256, 512
UDS/FE DOE
AES
HMAC 384
ECDSA384
Non-PQC Key Vault
Adams-Bridge
Crypto Subsystem
AHB-Slave
DMA etc. Widgets
Registers, Mailbox FSM, SOC Interface, (AHB if mastering is required else APB) Fuse registers
Mailbox SRAM
TAP Infrastructure
AXI-MGR
AXI-SUB
TAP/JTAG
Data Xfer Sideband
SOC-IFC
Adams-Bridge SRAM(s)

CPU/GPU/FPGA/etc..SoCs need a RoT?



Summary

Google Security Team has identified a security vulnerability in some AMD Zen-based CPUs. This vulnerability allows an adversary with local administrator privileges (ring 0 from outside a VM) to load malicious microcode patches. We have demonstrated the ability to craft arbitrary malicious microcode patches on Zen 1 through Zen 4 CPUs. The vulnerability is that the CPU uses an insecure hash function in the signature validation for microcode updates. This vulnerability could be used by an adversary to compromise confidential computing workloads protected by the newest version of AMD Secure Encrypted Virtualization, SEV-SNP or to compromise Dynamic Root of Trust Measurement.

AMD SEV-SNP users can verify the fix by confirming TCB values for SNP in their attestation reports (can be observed from a VM, consult AMD's security bulletins AMD-SB-3019 and AMD-SB-7033 for further details).

Severity

HIGH - Improper signature verification in AMD CPU ROM microcode patch loader may allow an attacker with local administrator privilege to load malicious CPU microcode resulting in loss of confidentiality and integrity of a confidential guest running under AMD SEV-SNP.

Proof of Concept

A test payload for Milan and Genoa CPUs that makes the RDRAND instruction return 4 can be downloaded here (applying it requires the user to be root from outside of a VM).

Availability

Learn more abo

CVSS:3.1/AV:L/AC:H

CVE ID

CVE-2024-56161

Weaknesses

No CWEs

Credits

josheads

spq

matrizzo

sirdarckcat

taviso

https://github.com/google/security-research/security/advisories/GHSA-4xq7-4mgh-gp6w

https://bughunters.google.com/blog/5424842357473280/zen-and-the-art-of-microcode-hacking

WHY THE RUSH WITH DILITHIUM?

PQC Transition timeline is super tight for hardware.

“It’s the law.” (And a bunch of NSMs, policies, directives, and regulations.) For DoD (CNSA):

NSS: *“.. Beginning 1 January 2027, unless otherwise excepted through public messaging on nsa.gov, protection profile, capabilities package, or waived through the waiver process, CNSA 2.0 algorithms will be required in all new products and services that provide cryptographic protection for users or for updates.”*

CNSSP-15 (March 2025): https://www.cnss.gov/CNSS/issuances/Policies.cfm

CNSSP-15 (December 2024 version)

Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)	Asymmetric algorithm used for key establishment	FIPS PUB 203 (Reference m)	ML-KEM-1024 to protect up to TOP SECRET
Module-Lattice-Based Digital Signature Algorithm (ML-DSA)	Asymmetric algorithm used for digital signatures	FIPS PUB 204 (Reference n)	ML-DSA-87 to protect up to TOP SECRET

https://www.cnss.gov/CNSS/issuances/Policies.cfm

ML-DSA-87 parameter set is the only one in CNSA 2.0.

NIST IR 8547 (November 2024 IPD)

Table 2: Quantum-vulnerable digital signature algorithms

Digital Signature Algorithm Family	Parameters	Transition
ECDSA [FIPS186]	112 bits of security strength	***Deprecated*** after 2030 ***Disallowed*** after 2035
	≥ 128 bits of security strength	***Disallowed*** after 2035
EdDSA [FIPS186]	≥ 128 bits of security strength	***Disallowed*** after 2035
RSA [FIPS186]	112 bits of security strength	***Deprecated*** after 2030 ***Disallowed*** after 2035
	≥ 128 bits of security strength	***Disallowed*** after 2035

https://doi.org/10.6028/NIST.IR.8547.ipd

Adam’s Bridge Status

- ABr is 47,600 lines of publicly available SystemVerilog: https://github.com/chipsalliance/adams-bridge

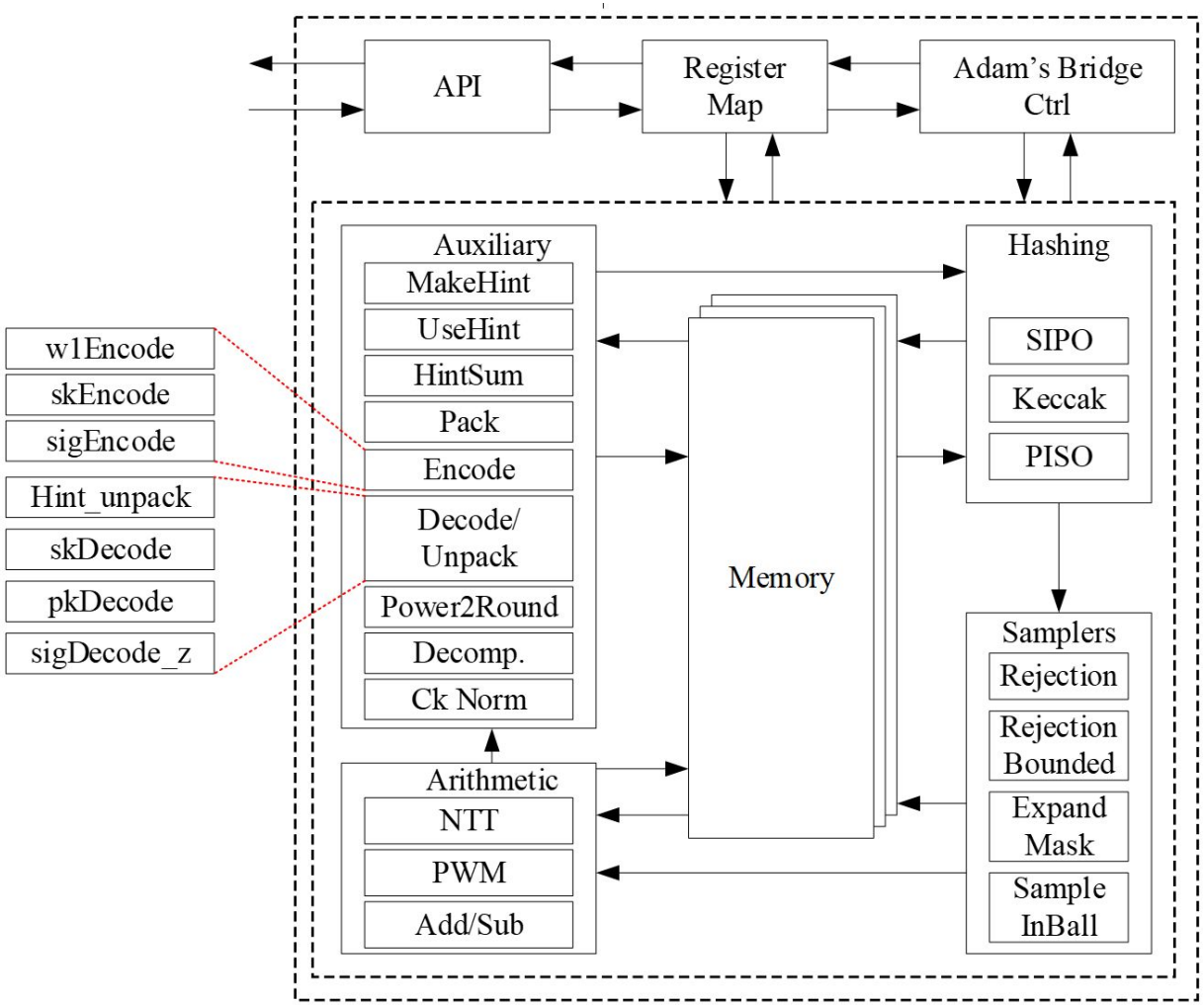
- We’ll use Adam’s Bridge release v1.0/v1.0.1 (May 16, 2025). Supports ML-DSA-87 (“Cat 5”) parameter set only.

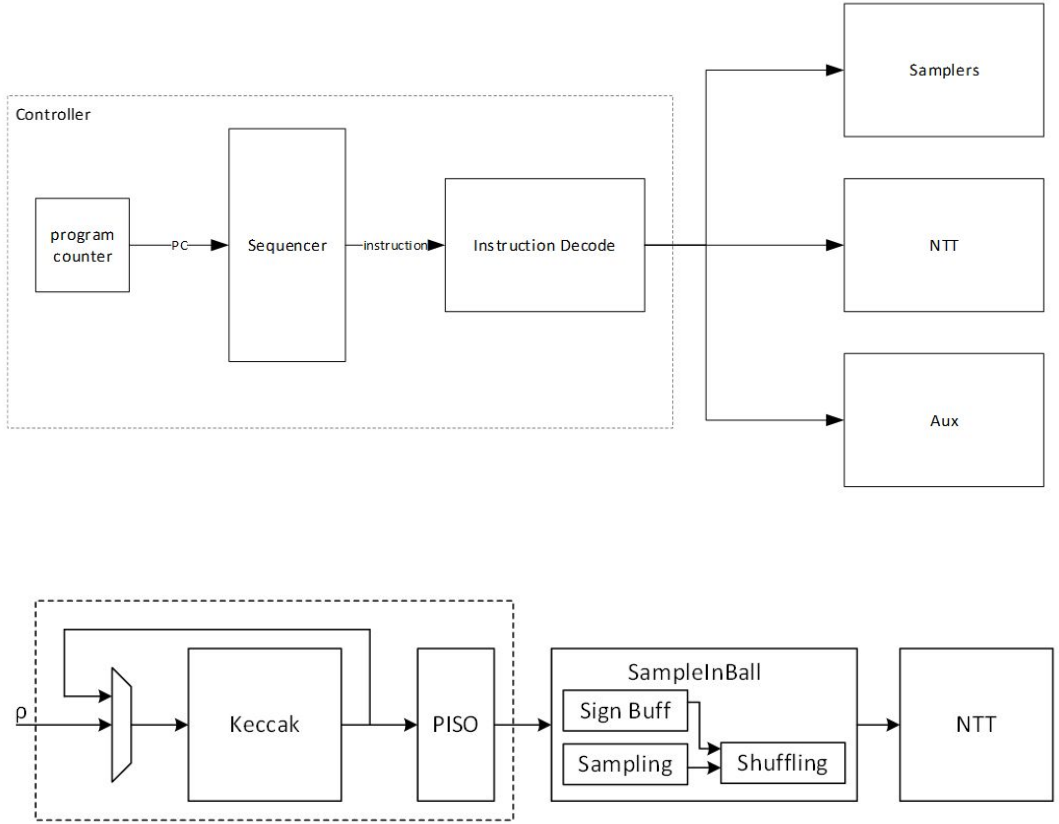
- Four functions (only): **{KeyGen, Sign, Verify, KeyGen+Sign}**.

- No programmability. Updates/patching not possible on ABr.

- The github repo has had its first ML-KEM commits in May.

Single-purpose HW Components ..





It's really *really* big..

Protected ML-DSA-87 is 0.114mm^2 @5nm (they're saying 600MHz):
"0.0921mm2 for stdcell and 0.0220mm^2 for 57.38 KB RAM"

Rough estimate from area: **3 million gate equivalents,**
~335k LUT FPGA synthesis supports this magnitude of ASIC size.

Protected Adams Bridge Area: >30x small microprocessor area!
(ARM Cortex M3 is 100-120k NAND2. IBEX on FPGA is 4k LUT).

Other secure ML-DSA HW modules are only 10%-30% of this size.

It’s pretty fast, though

ML-DSA-87 KeyGen: 15,530 Cycles

ML-DSA-87 Sign: 139,150 Cycles (average)

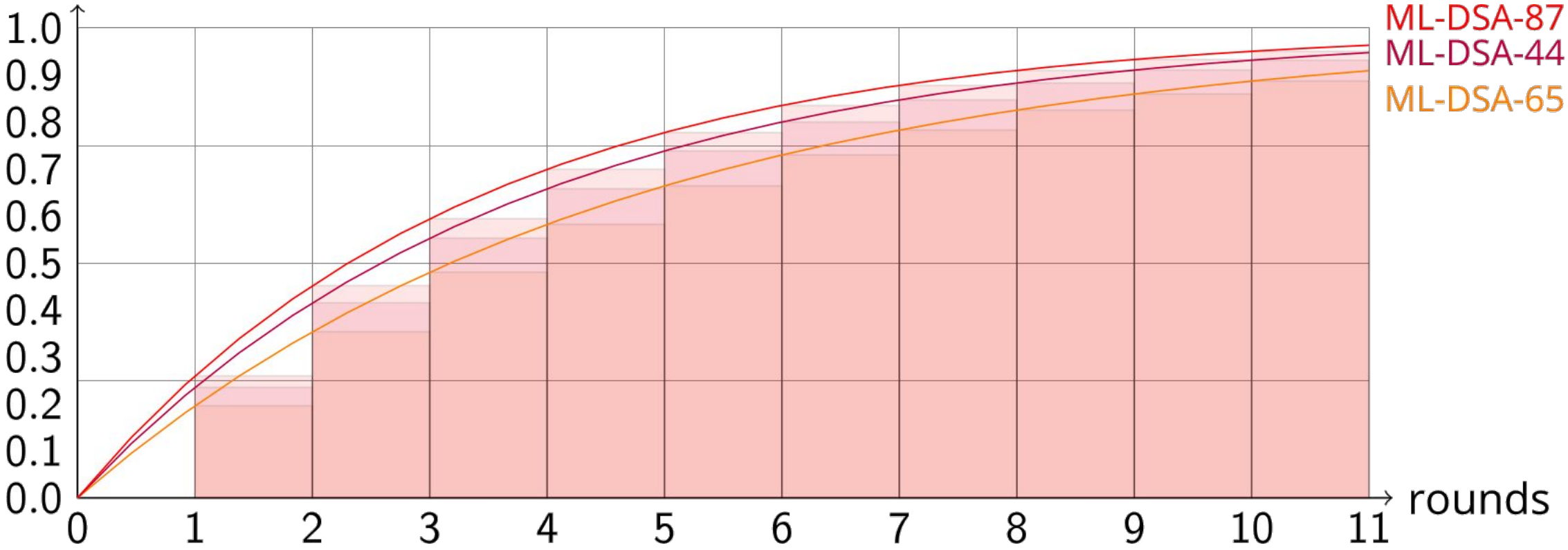
(minimum: 36,640 Cycles, 1 round: 35,958 cycles.)

ML-DSA-87 Verify: 18,770 Cycles

Signing flow example: Memory mapped (AHB) registers.

1. Host write keys, random, message, in registers.
2. Write to a control / trigger register to start.
3. Wait for status to become <ready> (perhaps intr).
4. Read the signature out from registers.

Each Signing Round: 26.0% success



Avg. ~4 rounds but ~1% of signatures need 16+ rounds!

(All ML-DSA implementations have this characteristic.)

Faulting Verification

FA Countermeasure for Sig Verify

"Classic" secure boot / firmware update bypass:
Glitch the yes/no result of signature verification.

ABr communicates the verification result in 64 bytes:

"To mitigate a possible fault attack on Boolean flag verification result, a 64-byte register is considered. Firmware is responsible for comparing the computed result with a certain segment of signature (segment c\~), and if they are equal the signature is valid."

[adams-bridge/docs/AdamsBridgeHardwareSpecification.md]

Algorithm 8 ML-DSA.Verify_internal(pk, M', σ)

Internal function to verify a signature σ for a formatted message M'.

Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$ and message $M' \in \{0,1\}^*$.
Input: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.
Output: Boolean

1: $(\rho, \mathbf{t}_1) \leftarrow$ pkDecode(pk)
2: $(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow$ sigDecode(σ) ▷ signer's commitment hash $\tilde{c}$, response $\mathbf{z}$, and hint $\mathbf{h}$
3: **if** $\mathbf{h} = \perp$ **then return** false ▷ hint was not properly encoded
4: **end if**
5: $\hat{\mathbf{A}} \leftarrow$ ExpandA(ρ) ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
6: $tr \leftarrow$ H$(pk, 64)$
7: $\mu \leftarrow$ (H(BytesToBits$(tr)||M', 64)$) ▷ message representative that may optionally be computed in a different cryptographic module
8: $c \in R_q \leftarrow$ SampleInBall$(\tilde{c})$ ▷ compute verifier's challenge from $\tilde{c}$
9: $\mathbf{w}'_{\text{Approx}} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}(\mathbf{t}_1 \cdot 2^d))$ ▷ $\mathbf{w}'_{\text{Approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$
10: $\mathbf{w}'_1 \leftarrow$ UseHint$(\mathbf{h}, \mathbf{w}'_{\text{Approx}})$ ▷ reconstruction of signer's commitment
11: ▷ UseHint is applied componentwise (see explanatory text in Section 7.4)
12: $\tilde{c}' \leftarrow$ H$(\mu||\text{w1Encode}(\mathbf{w}'_1), \lambda/4)$ ▷ hash it; this should match $\tilde{c}$
13: **return** $[[\ ||\mathbf{z}||_\infty < \gamma_1 - \beta]]$ **and** $[[\tilde{c} = \tilde{c}']]$

Algorithm 8 ML-DSA.Verify_internal(pk, M', σ)

Internal function to verify a signature σ for a formatted message M'.

Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$ and message $M' \in \{0,1\}^*$.
Input: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.
Output: Boolean

1: $(\rho, \mathbf{t}_1) \leftarrow$ pkDecode(pk) — first part of signature
2: $(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow$ sigDecode(σ) ▷ signer's commitment hash $\tilde{c}$, response $\mathbf{z}$, and hint $\mathbf{h}$
3: **if** $\mathbf{h} = \perp$ **then return** false ▷ hint was not properly encoded
4: **end if**
5: $\hat{\mathbf{A}} \leftarrow$ ExpandA(ρ) ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
6: $tr \leftarrow$ H$(pk, 64)$
7: $\mu \leftarrow$ (H(BytesToBits$(tr)||M', 64)$) ▷ message representative that may optionally be computed in a different cryptographic module
8: $c \in R_q \leftarrow$ SampleInBall$(\tilde{c})$ ▷ compute verifier's challenge from $\tilde{c}$
9: $\mathbf{w}'_{\text{Approx}} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}(\mathbf{t}_1 \cdot 2^d))$ ▷ $\mathbf{w}'_{\text{Approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$
10: $\mathbf{w}'_1 \leftarrow$ UseHint$(\mathbf{h}, \mathbf{w}'_{\text{Approx}})$ ▷ reconstruction of signer's commitment
11: ▷ UseHint is applied componentwise (see explanatory text in Section 7.4)
12: $\tilde{c}' \leftarrow$ H$(\mu||\text{w1Encode}(\mathbf{w}'_1), \lambda/4)$ — returns this ▷ hash it; this should match $\tilde{c}$
13: **return** $[[\, ||\mathbf{z}||_\infty < \gamma_1 - \beta]]$ **and** $[[\tilde{c} = \tilde{c}']]$ — firmware compares

Algorithm 8 ML-DSA.Verify_internal(pk, M', σ)

Internal function to verify a signature σ for a formatted message M'.

Input: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$ and message $M' \in \{0,1\}^*$.
Input: Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen }(\gamma_1-1))+\omega+k}$.
Output: Boolean

1: $(\rho, \mathbf{t}_1) \leftarrow$ pkDecode(pk) — first part of signature
2: $(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow$ sigDecode(σ) ▷ signer's commitment hash $\tilde{c}$, response $\mathbf{z}$, and hint $\mathbf{h}$
3: **if** $\mathbf{h} = \perp$ **then return** false ▷ hint was not properly encoded
4: **end if** — early abort: malformed signature
5: $\hat{\mathbf{A}} \leftarrow$ ExpandA(ρ) ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
6: $tr \leftarrow$ H$(pk, 64)$
7: $\mu \leftarrow$ (H(BytesToBits$(tr)||M', 64)$) ▷ message representative that may optionally be computed in a different cryptographic module
8: $c \in R_q \leftarrow$ SampleInBall$(\tilde{c})$ ▷ compute verifier's challenge from $\tilde{c}$
9: $\mathbf{w}'_{\text{Approx}} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}(\mathbf{t}_1 \cdot 2^d))$ ▷ $\mathbf{w}'_{\text{Approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$
10: $\mathbf{w}'_1 \leftarrow$ UseHint$(\mathbf{h}, \mathbf{w}'_{\text{Approx}})$ ▷ reconstruction of signer's commitment
11: ▷ UseHint is applied componentwise (see explanatory text in Section 7.4)
12: $\tilde{c}' \leftarrow$ H$(\mu||\text{w1Encode}(\mathbf{w}'_1), \lambda/4)$ — returns this ▷ hash it; this should match $\tilde{c}$
13: **return** $[[\,||\mathbf{z}||_\infty < \gamma_1 - \beta]]$ **and** $[[\tilde{c} = \tilde{c}']]$ — firmware compares

What is returned in early abort?

I have all of ML-DSA RTL in a verilator testbench – easy to find out: **The return value is 64 zero bytes.**

There is nothing preventing me (attacker) from: Offering an invalid “forged” signature with 64 zero bytes **c~** in the beginning.

64-byte comparison will match – “signature is valid”?

(For all messages and public keys!) I reported this..

Caliptra PSIRT Response on May 15

“We recognize that robust hardware-FW contract design is key to secure cryptographic integrations. Based on your report, we also will mention MLDSA signature verification fails if c~ is all zeros as we do in our Caliptra-ROM/FW documentation.”

So.. the caller/driver needs to do some additional checks on signatures and this was not mentioned in hardware documentation.

So Adam’s Bridge is not exactly “fire and forget” after all.

Indeed a firmware fix has been committed on March 21, 2025: https://github.com/chipsalliance/caliptra-sw/commit/5a675c82c97c9f15170a8333ead502ddb14967f4

caliptra-sw / drivers / src / **mldsa87.rs**

Code | Blame | 604 lines (491 loc) · 18.6 KB · | Raw

```
442         #[cfg_attr(not(feature = "no-cfi"), cfi_impl_fn)]
443         pub fn verify(
444             &mut self,
445             pub_key: &Mldsa87PubKey,
446             msg: &Mldsa87Msg,
447             signature: &Mldsa87Signature,
448         ) -> CaliptraResult<Mldsa87Result> {
449             #[cfg(feature = "fips-test-hooks")]
450             unsafe {
451                 crate::FipsTestHook::error_if_hook_set(crate::FipsTestHook::MLDSA_VERIFY_FAILURE)?
452             }
453
454             let truncated_signature = &signature.0[..MLDSA87_VERIFY_RES_WORD_LEN];
455             if truncated_signature == [0; MLDSA87_VERIFY_RES_WORD_LEN] {
456                 Err(CaliptraError::DRIVER_MLDSA87_UNSUPPORTED_SIGNATURE)?;
457             }
458
459             let verify_res = self.verify_res(pub_key, msg, signature)?;
```

That check looks unprotected

If we glitch this “all zero” comparison, the rest of the verification routine thinks that signature is ok.

Note: Firmware offers an oracle on *“how verification fails”* - code DRIVER_MLDSA87_UNSUPPORTED_SIGNATURE.

Combined Strategy:

1. Create malicious firmware with a signature block that has first 64 bytes set to zero.
2. Attempt to glitch the all-zero check in CPU.

Can’t demo 🤷 (target not ready)

This looks all feasible in theory, but the firmware is still being written. *(That part of the firmware is not even merged into the main branch yet.)*

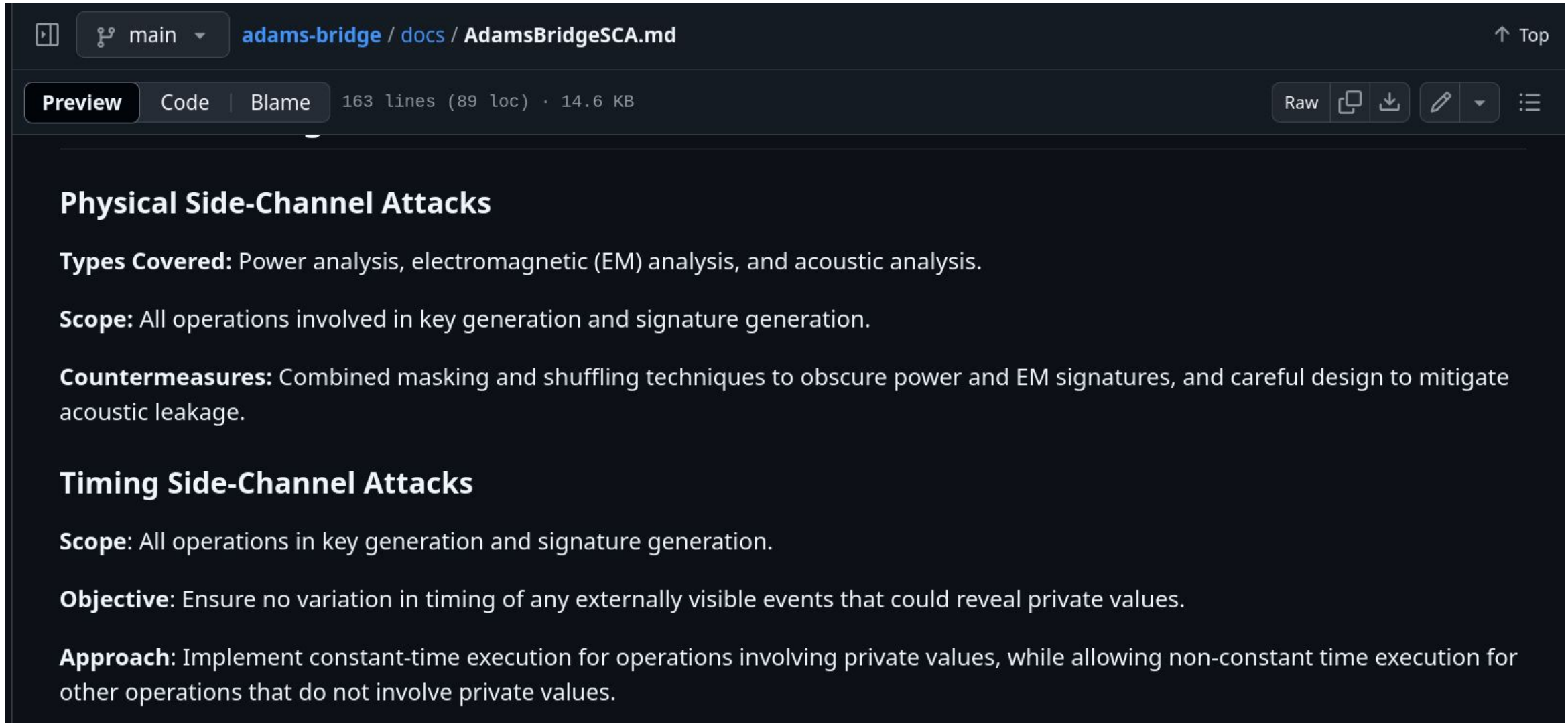
Note: At 330k LUTs the current design is too big to fit on Artix7 FPGA targets (CW305) anymore.

Probably fits on a $10k CW340 “Luna” Board which has Virtex UltraScale+ and allows glitching, etc.

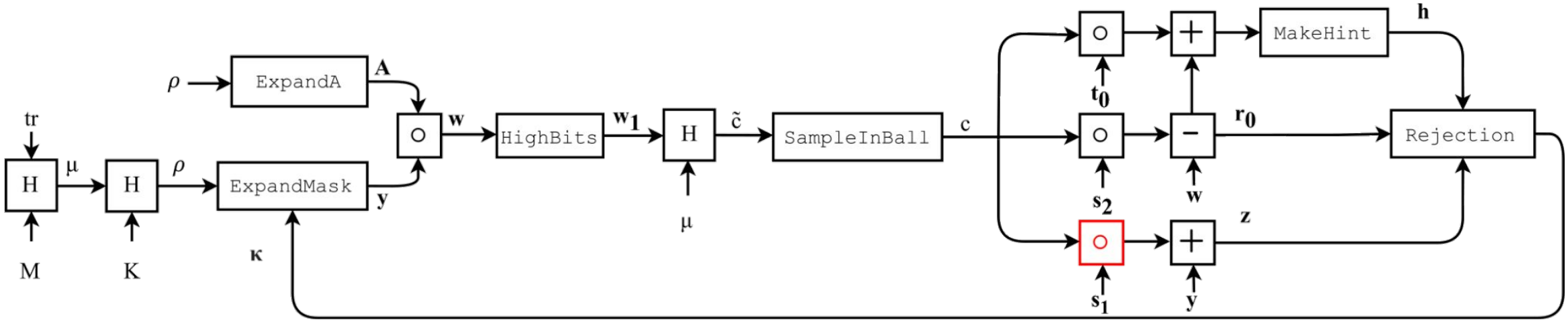
Or try a silicon target. We’ll keep tracking this.

Why Adam’s Bridge Leaks

Adams Bridge has bold SCA Claims



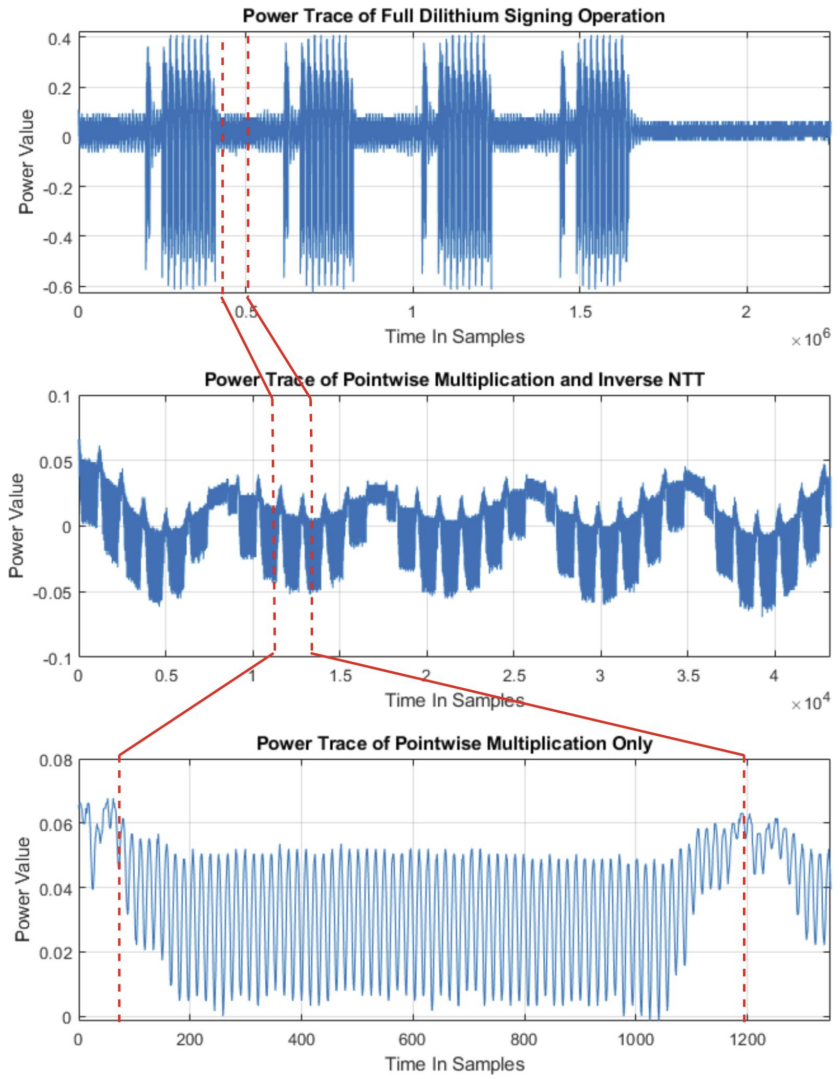
FAU paper on Adam's Bridge

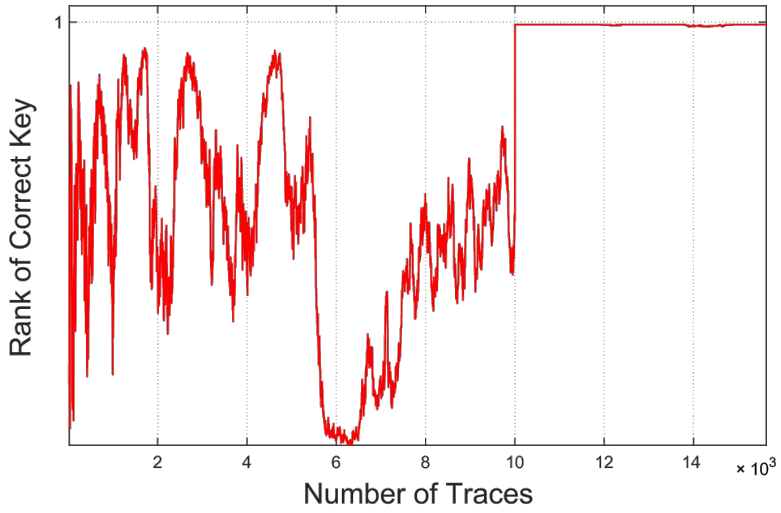


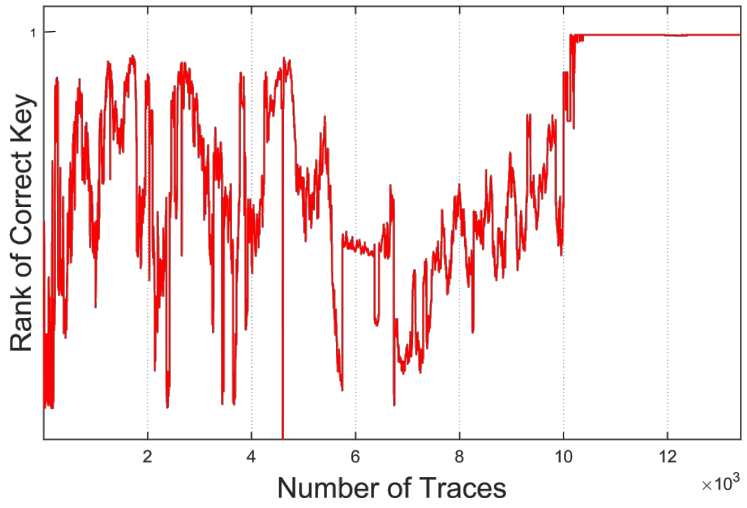
- A Correlation Power Analysis (CPA) attack on a **secret-key multiplication step**; late October '24 version of Adam's Bridge.
- 10,000 traces to recover the secret keys, CW305 A7 FPGA target.
- M. Karabulut, R. Azarderakhsh, *"Efficient CPA Attack on Hardware Implementation of ML-DSA in Post-Quantum Root of Trust."*

  (IEEE HOST '2025) https://ia.cr/2025/009

FPGA Target and FAU Key Extraction



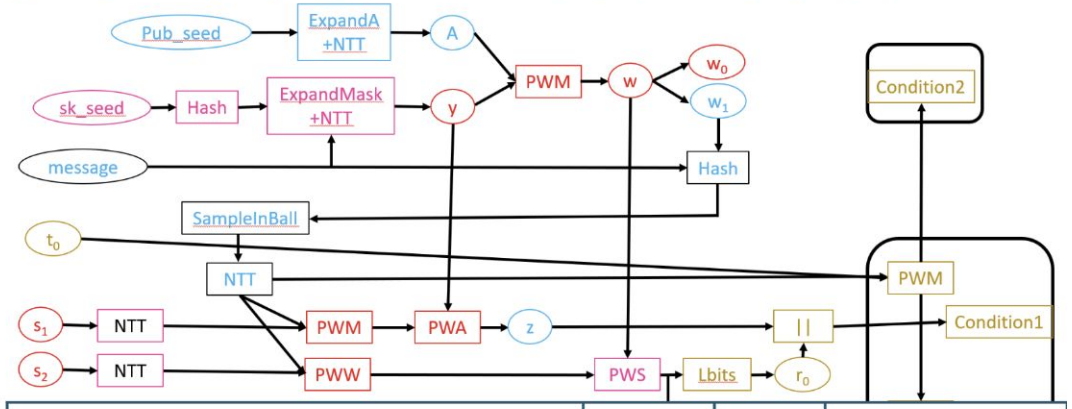




“Review attack literature” (only?)

Developing a Comprehensive SCA Threat Model

- Reviewing literature and listing existing SCAs
- Extending attack scope to include new and novel attacks
- Performing vulnerability assessment over data and control flows of our implementation
- Categorizing the attacks and setting up a priority list
- Revisiting our threat model after each RTL code review



Vectors	PWM	NTT	Protection
Does it have a secret input causing variation in the execution time?	No	No	Constant time design
Has it been exploited with profiled SCAs?	Yes	Yes	Hiding |Masking
Does it involve a sequential execution and have high signal-to-noise ratio?	Yes	Yes	Hiding |Masking
Has it been exploited with CPA/DPA?	Yes	No	Masking
Does it receive an observable public input while another operand is a fixed secret?	Yes	No	Masking

E. Karabulut, K. Upadhyayula, **"Side-Channel Countermeasures for the Adams Bridge Accelerator"**

..but advanced research exists

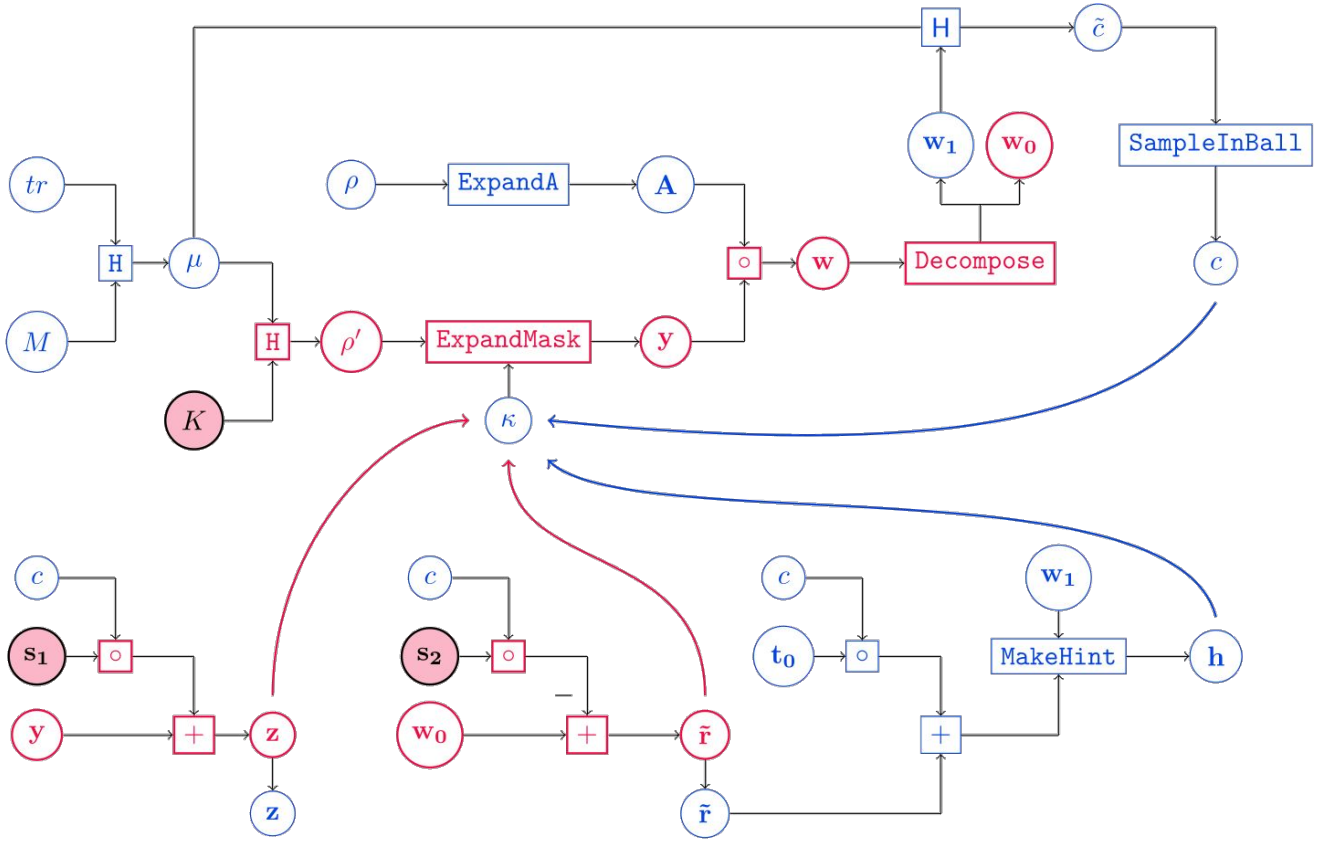
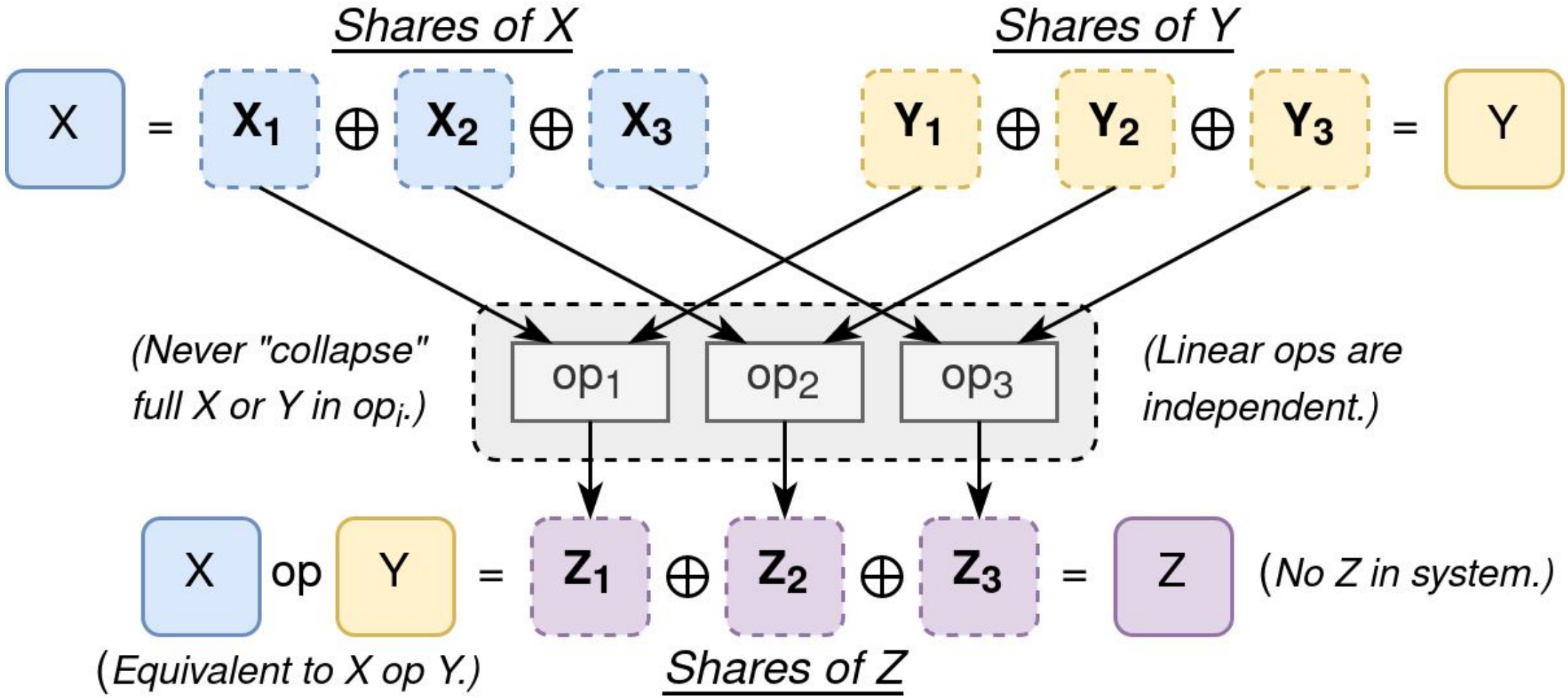


Figure 2: Graphical representation of the signature generation. Input: sk, M, Output: $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$. Curved arrows represent rejection checks. Red: sensitive. Blue: non-sensitive.

Azouaoui et al. *"Protecting Dilithium against Leakage: Revisited Sensitivity Analysis and Improved Implementations"* https://ia.cr/2022/1406

MASKING: SPLIT SECRETS INTO SHARES



ACTUALLY MASKING PQC ALGORITHMS

Type	Relationship	Algebraic Object
Algebraic / Prime Field	$X = X0 + X1 \pmod{q}$	Mod 3329 (Kyber) or 8380417 (Dilithium)
Algebraic / Power-of-2	$X = X0 + X1 \pmod{2^n}$	Some Lattice Crypto, SHA2, etc
Boolean / Binary Field	$X = X0 \oplus X1$	Nonlinear Functions, shifts, symmetric Crypto

- **Masking splits all secret variables into "shares."** Even perfect measurement of an individual share does not leak secret info.

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

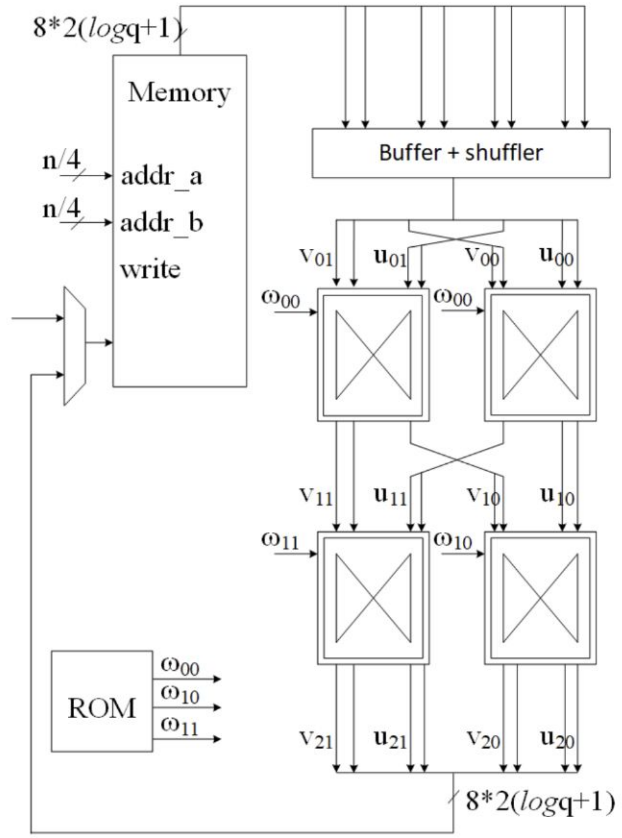
- **Why?** The algorithms are not homogenous like RSA or ECC but contain a number of dissimilar steps. In practice, one needs to design dozen different provable masking gadgets for one algorithm such as ML-DSA.

34

ABr: PW Mult and INTT (only)

Masking in Adam's Bridge – NTT, PWM

- Some PWM and INTT operations in CRYSTAL-DILITHIUM must have strong countermeasures to protect secrets
- Shuffling is not strong enough
- Masked BFU with 2 shares per input
- 4x memory overhead
- >4x NTT area overhead
- 4x latency overhead
- Very strong countermeasure



So, AbR is not really masked

- **Secret keys are not masked (the way we understand it.)**

  “Operations Protected with Masking: Point-wise multiplication and the first state of inverse NTT.”

- **Key generation is not protected at all**.

  “The key generation operation does not have a non-profiled attack vector since its nature is inherently secure against CPA-style attacks.” (well, CPA..)

- Well, there can’t be security proofs for this because secret variables are not protected everywhere..

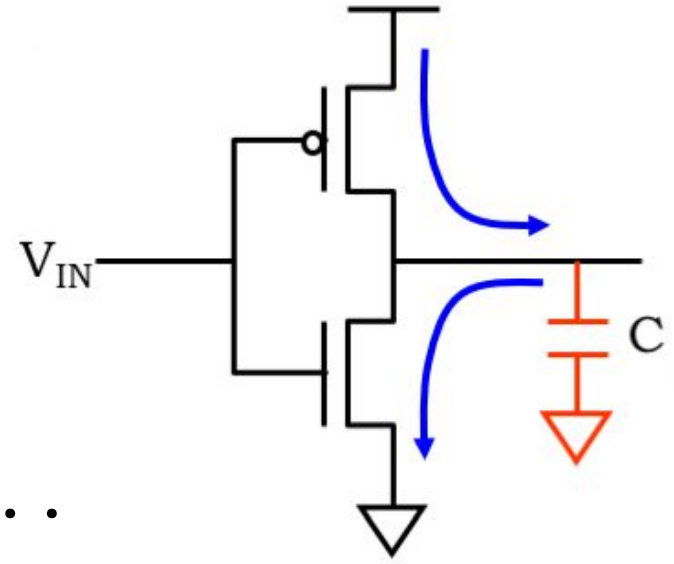
DPA/DEMA Fundamentals – “Toggling”

Logic changes (0 → 1 or 1 → 0) consume ***dynamic power***.
Toggling is used in **DPA** “Differential Power Analysis.”

There is also ***static power*** consumption, but unchanging bits generally don’t leak much side-channel information.

State changes also emanate on the ***electromagnetic spectrum*** (**DEMA**).

Programmable CPU registers and memory are *just one part* of the vast logic machinery that handles your secret bits..



OBSERVATIONS

- No control processor, but a hardcoded microcode “sequencer”, basically a finite state machine:

  mldsa_seq_prim.sv: 324 “instructions”

  mldsa_seq_sec.sv: 114 “instructions”

- RTL is reasonably fast to simulate with verilator:

  3200 cycles/sec (13.3sec/sign) without VCD dump

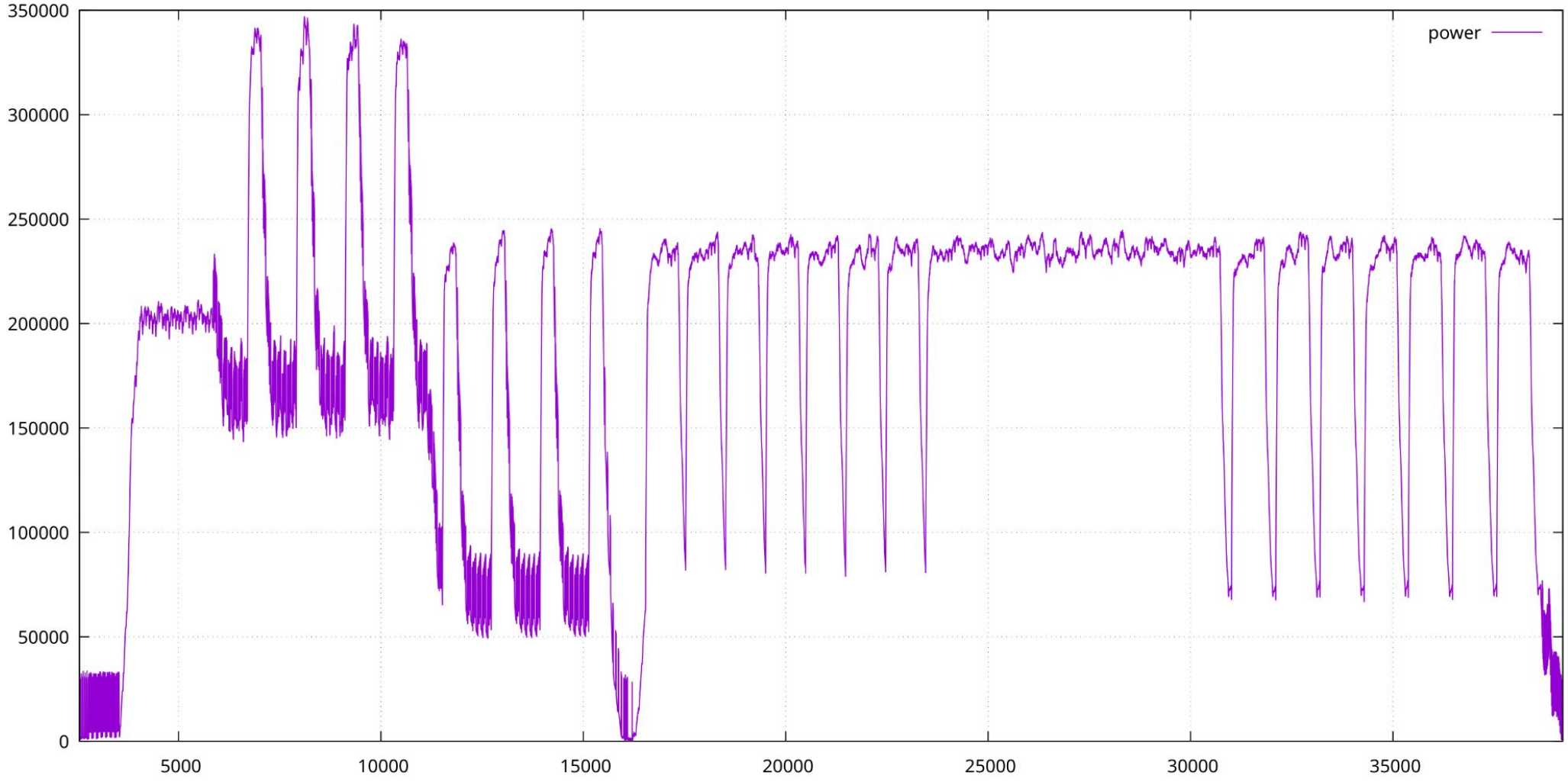
  670 cycles/sec (63.6sec/sign) with VCD dump

- Simulation repo: https://github.com/ml-dsa/abr-sim

Rough VCD Pre-Silicon Testing

- **verilator**: Produce VCD traces, DUT doing signing operations.
- **readvcd**: VCD-to-Trace program reads VCD file: Keeps track of all state bits and compute Hamming distance for each clock cycle. These “toggle counts” approximate power for DPA.
- Since the simulated signal is very “clean”, not nearly as many traces are required than from FPGA-oscilloscope setup.
- Better for locating leakage. Very precise; we get exact cycle of leak points and can check (from VCD) the names of wires and signals that were active and causing it.

“Toggle Trace” of ML-DSA Signing



PQC Leakage Assessment Strategies

1. **Fixed vs Random** (non-specific t-test)
- Trace set A: Fixed secret variable for every trace.
- Trace set B: New random secret variable for each trace.
- Statistically distinguishing of A from B is evidence of leakage.

2. **A/B Categorization** works with capture-then-analyze flow:
- Record traces with detailed test vector metadata.
- Traces are categorized ***after capture*** to A and B sets based on CSP selection criteria. Example: a specific secret key bit.
- The same trace data can be categorized repeatedly.

In both cases:

Set A and Set B statistically differentiable with t-test = **FAIL**.

Test Vector Leakage Assessment

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.

Dilithium Secret Key TVLA

- Basic TVLA fix-vs-random is really only suitable for symmetric ciphers..

- And a Dilithium secret key **has six components**, three ($K, \mathbf{s_1}, \mathbf{s_2}$) of which are actually secret for signing:

$$\mathbf{SK} = (\ \rho,\ K,\ tr,\ \mathbf{s_1},\ \mathbf{s_2},\ \mathbf{t_0}\ )$$

- The public parts, e.g. matrix A expansion from symmetric seed ρ do not need protection.

- How to create two sets of test vectors here?

ML-DSA Key Generation

Algorithm 6 ML-DSA.KeyGen_internal(ξ)

Generates a public-private key pair from a seed.

Input: Seed $\xi \in \mathbb{B}^{32}$

Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$
and private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen }(2\eta)+dk)}$.

1: $(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow \text{H}(\xi||\text{IntegerToBytes}(k,1)||\text{IntegerToBytes}(\ell,1),128)$
2: ▷ expand seed
3: $\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
4: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \text{ExpandS}(\rho')$
5: $\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$ ▷ compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \text{Power2Round}(\mathbf{t})$ ▷ compress $\mathbf{t}$
7: ▷ PowerTwoRound is applied componentwise (see explanatory text in Section 7.4)
8: $pk \leftarrow \text{pkEncode}(\rho, \mathbf{t}_1)$
9: $tr \leftarrow \text{H}(pk, 64)$
10: $sk \leftarrow \text{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$ ▷ K and tr are for use in signing
11: **return** (pk, sk)

Fixing (s1,s2) by inserting ρ'

Algorithm 6 ML-DSA.KeyGen_internal(ξ)

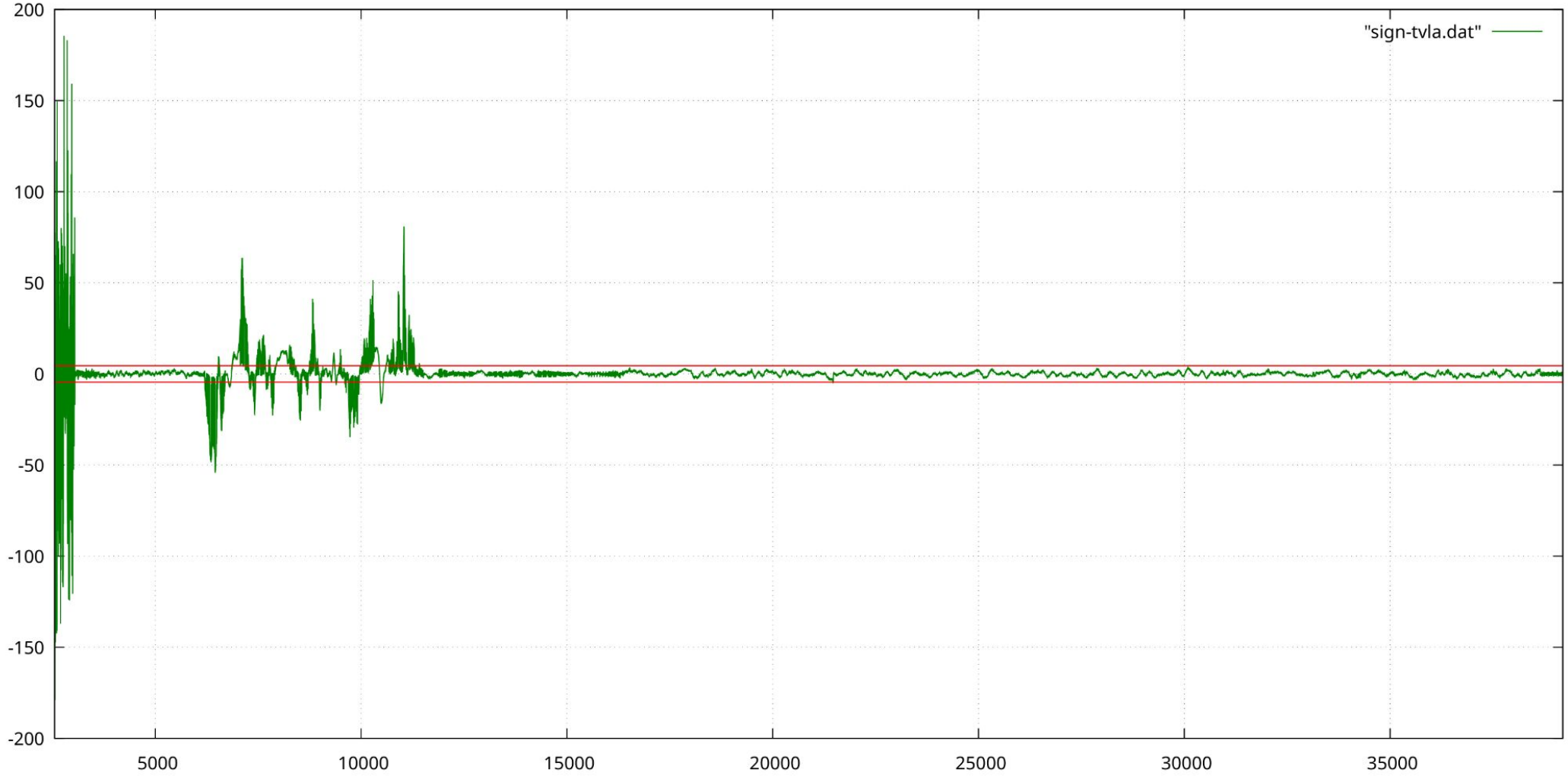
Generates a public-private key pair from a seed.

Input: Seed $\xi \in \mathbb{B}^{32}$

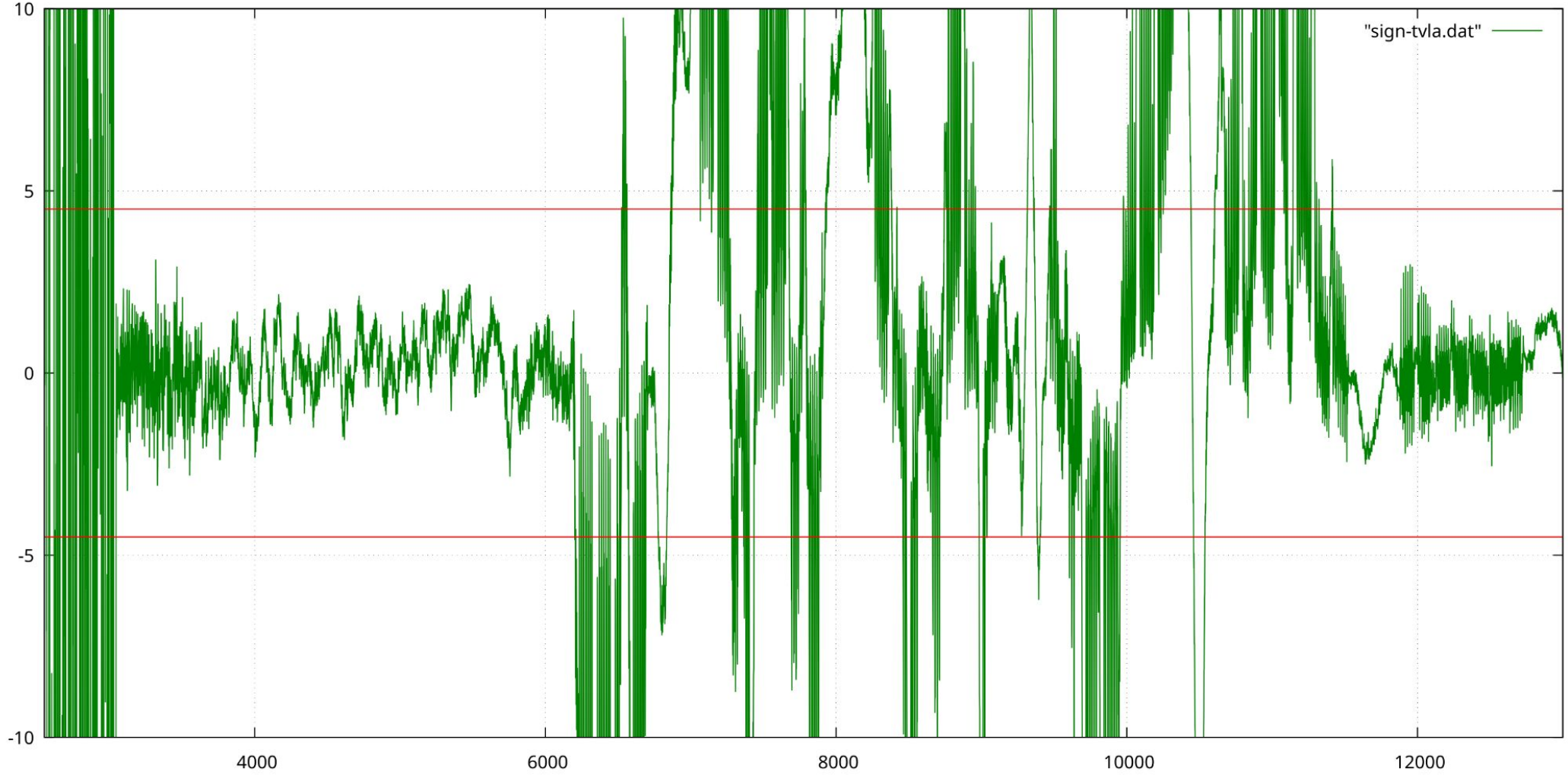
Output: Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen }(q-1)-d)}$
and private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen }(2\eta)+dk)}$.

1: $(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow \text{H}(\xi||\text{IntegerToBytes}(k,1)||\text{IntegerToBytes}(\ell,1),128)$
2: ▷ expand seed
3: $\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
4: $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \text{ExpandS}(\rho')$ override ρ' for the fixed set
5: $\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$ ▷ compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
6: $(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \text{Power2Round}(\mathbf{t})$ ▷ compress $\mathbf{t}$
7: ▷ PowerTwoRound is applied componentwise (see explanatory text in Section 7.4)
8: $pk \leftarrow \text{pkEncode}(\rho, \mathbf{t}_1)$
9: $tr \leftarrow \text{H}(pk, 64)$
10: $sk \leftarrow \text{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$ ▷ K and tr are for use in signing
11: **return** (pk, sk)

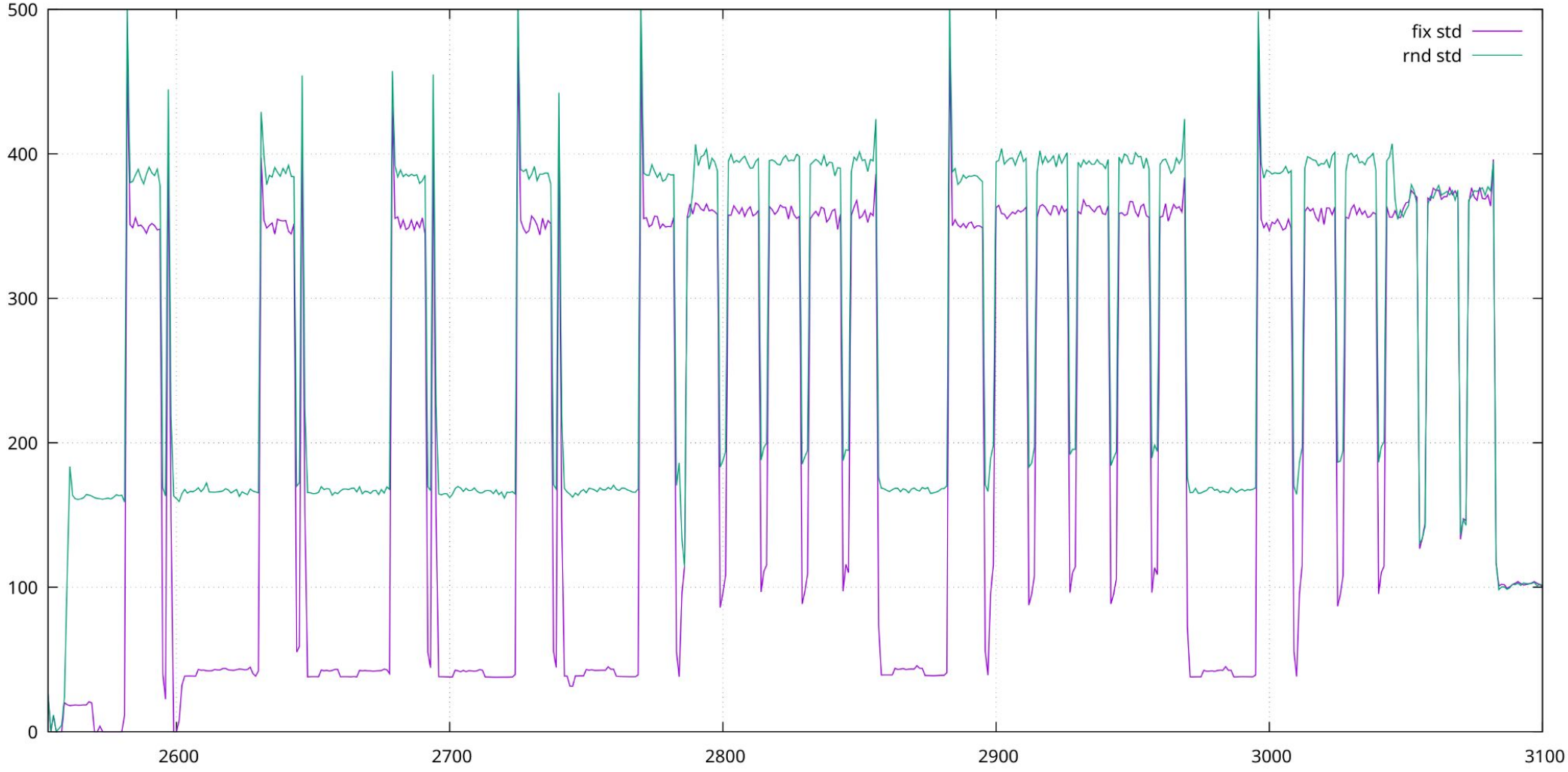
Fix-vs-Random TVLA: 11000 traces



Zoom into the beginning



Comparing std. devs (zoomed)



Examining Some Leak Points

Cycle	TVLA	Major Active signals
2783	+185.5	top0.skdecode_inst.s1s2_data

Biggest spikes are created in the beginning – the secret key is not masked and has to be deserialized as plaintext for NTT.

6456	-54.2	masked_bf_inst00.add_res_delay_inst.shift_reg
7118	+63.7	masked_bf_inst00.sub_inst_0.a2b_inst.x_reg
11042	+80.9	pwm_inst00.add_inst0.add_res_reg

There may be some bugs in the pointwise multiplication and NTT gadgets as they have leakage correlations with secret keys.

What can we say from Pre-Silicon

- **No surprise**: Leakage happens during early phases when the “plaintext” secret key is being moved about and transformed (NTT(s1), NTT(s2) ..)

- This would be automatically considered “broken” by the theory. However, leakage alone does not imply efficient key recovery or forgery attacks.

- Adam’s Bridge may be (in practice) saved by “being fast”; having wide data paths. A lot bits are moved in parallel so the attacker learns relatilvely little of individual bits.

- Follow-up questions: **Where do the keys come from?** etc

SUMMARY POINTS

- Expect to be working on PQC targets **a lot** in near future.
- Combining fault attacks with malformed signatures can help attacks against signature verification (learn how ML-DSA, SLH-DSA works!)
- Researchers know that many side-channel attacks work against Dilithium. Lattice countermeasure “theory” work has been going on for many years.
- Attack papers do not claim to describe all of the vulnerabilities, just what happened to be the low hanging fruit for specific targets.
- For countermeasures, take a **systematic masking approach** – must be complemented with other countermeasures, and with adversarial analysis.
- Masking and other countermeasures **impact architecture**. Don’t try to somehow “patch” countermeasures into an unprotected implementation.
- My ABr pre-silicon testing code: https://github.com/ml-dsa/abr-sim

Supplementary Material (consulting if needed)

Masking Elsewhere ..

WrapQ: Side-Channel Secure Key Management for Post-Quantum Cryptography

Markku-Juhani O. Saarinen, PQShield Ltd, Tampere University

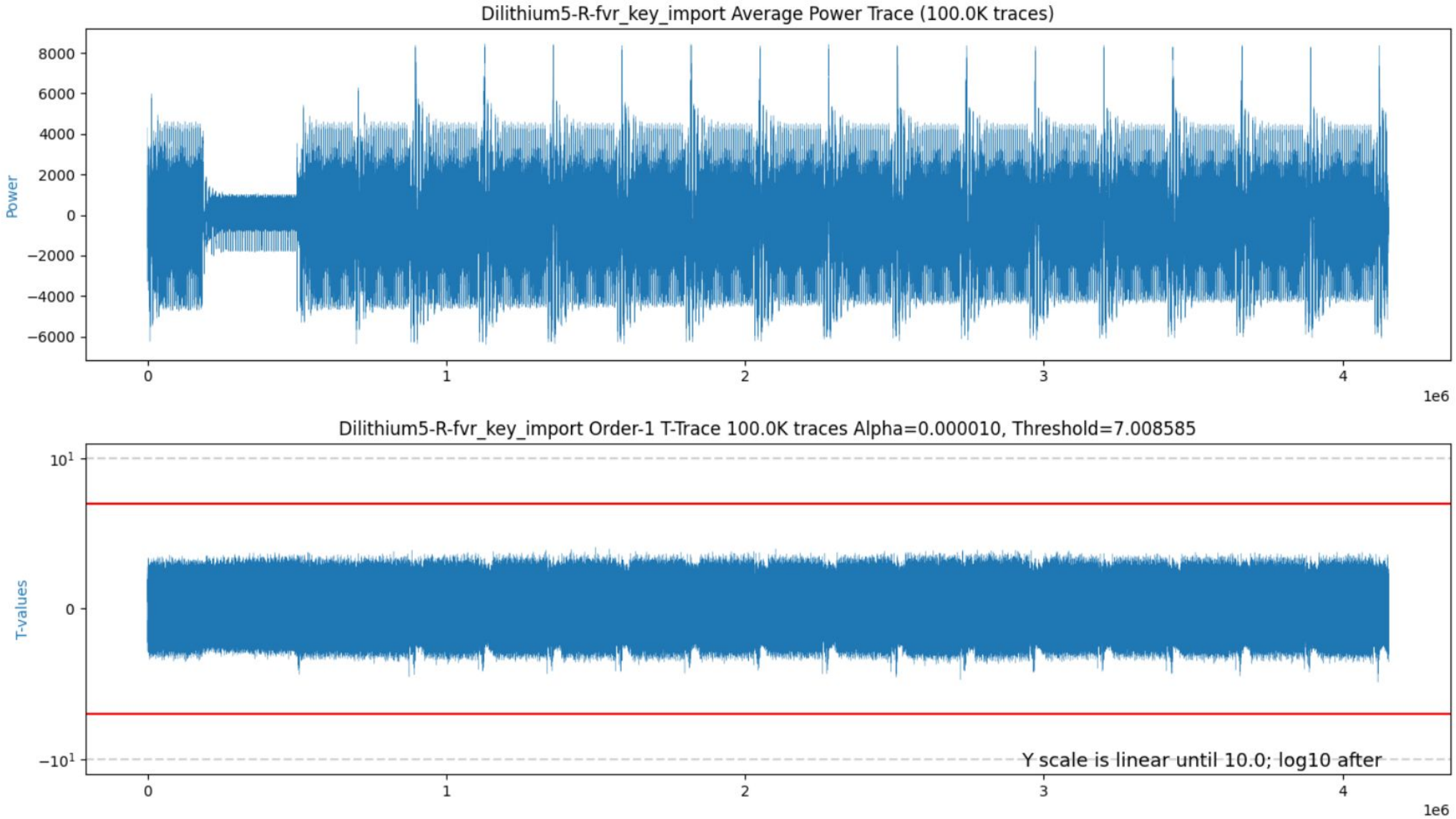
Abstract

Transition to PQC brings complex challenges to builders of secure cryptographic hardware. PQC keys usually need to be stored off-module and protected via symmetric encryption and message authentication codes. Only a short, symmetric Key-Encrypting Key (KEK) can be managed on-chip with trusted non-volatile key storage. For secure use, PQC key material is handled in masked format; as randomized shares. Due to the masked encoding of the key material, algorithm-specific techniques are needed to protect the side-channel security of the PQC key import and export processes. In this work, we study key handling techniques used in real-life secure Kyber and Dilithium hardware. We describe WrapQ, a masking-friendly key-wrapping mechanism designed for lattice cryptography. On a high level, WrapQ protects the integrity and confidentiality of key material and allows keys to be stored outside the main security boundary of the module. Significantly, its wrapping and unwrapping processes minimize side-channel leakage from the KEK integrity/authentication keys as well as the masked Kyber or Dilithium key material payload.

In this work, we study key handling techniques used in real-life secure Kyber and Dilithium hardware. We describe WrapQ, a masking-friendly key-wrapping mechanism designed for lattice cryptography. On a high level, WrapQ protects the integrity and confidentiality of key material and allows keys to be stored outside the main security boundary of the module. Significantly, its wrapping and unwrapping processes minimize side-channel leakage from the KEK integrity/authentication keys as well as the masked Kyber or Dilithium key material payload. (..)

https://eprint.iacr.org/2022/1499 (PQCrypto 2023)

Example 2: Dilithium5 WrapQ Key Import RvF CSP (#3)



On Certification of PQC Modules

FIPS 140-3 for PQC

- Required by U.S. Govt and industrial best practices.
- ACVP certifications have been issued since August 2024.
- Currently focuses only on functional (test vector) and “checklist compliance”.
- Random numbers: SP 800-90 testing still good for PQC.
- Slowly coming: “non-invasive” (ISO 17825) leakage assessment for FIPS 140-3 level 3+.

CC AVA_VAN and “Attack Potential”

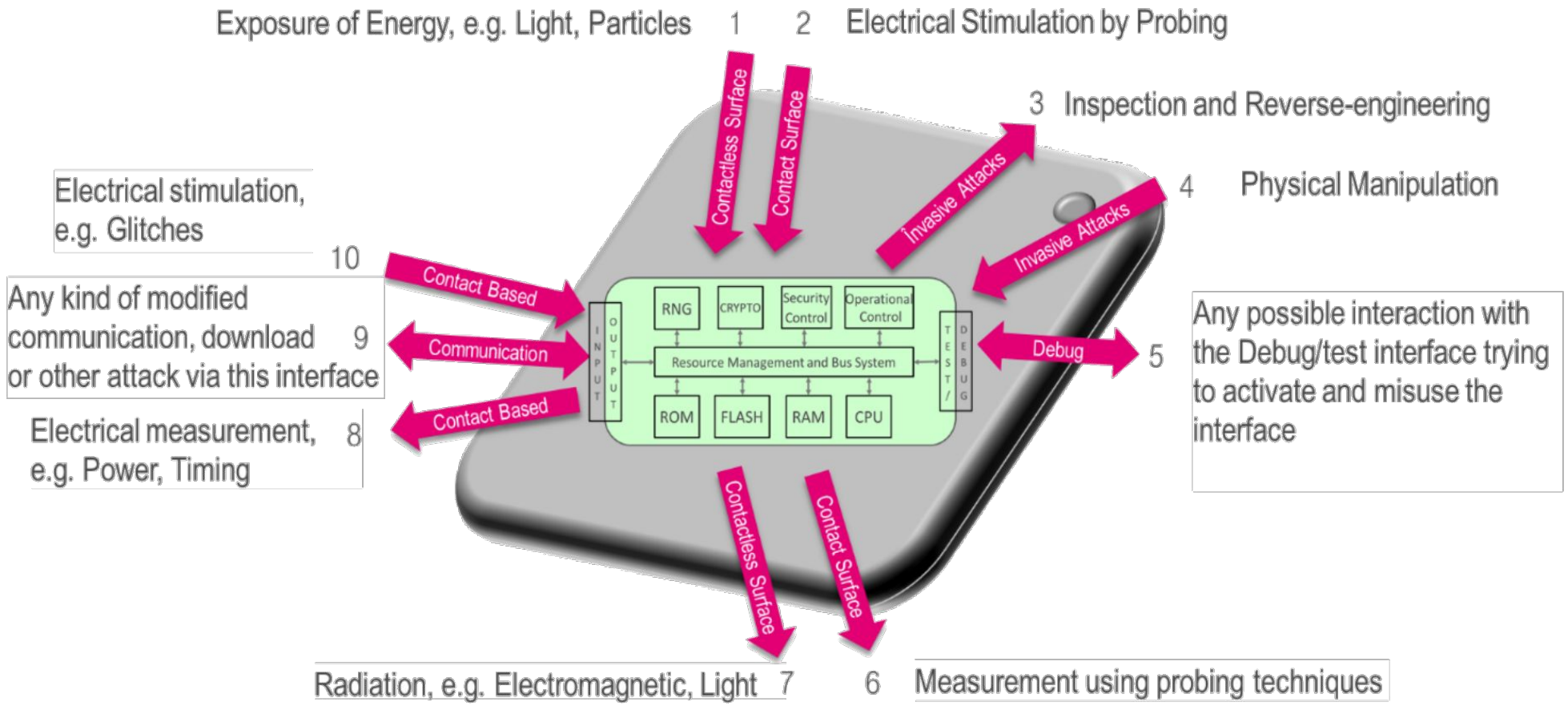
- High assurance level (EUCC: AVA_VAN.3+) is required for **Root of Trust IP**, **Smart Cards**, **Secure elements**, **IoT** (SESIP).
- AVA_VAN checks more of real-life security via a “penetration test.”
- AVA_VAN security level is determined by "attack potential": A score-based system measuring attack cost.
- Includes SCA, FA, etc.

(From PP-0117)

3.2 Threats

The threats described in this section are applicable to the base Protection Profile. For threats related to functional extensions see Chapter 7.

The following figure describes the attacks that are applicable to the TOE. The interactions related to the attacks are marked with red arrows.



https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Reporte/ReportePP/pp0117V2b_pdf.pdf

PQC Module Certification

- **FIPS 140-3**: Functional testing of ML-DSA implementations since Aug 2024. Bunch of modules have been certified.
- Most major vendors are upgrading their product lines, as they have to.

- **ANSSI**, **BSI** and other Europeans recommend ML-DSA to be combined with a classical algorithm (RSA or ECDSA).
- BSI had not received certification requests in January 2025.
- ANSSI has started working on procedures to test ML-DSA.

CC: Main Protection Profiles

AVA_VAN.3+ is a common requirement for Root of Trust and Security IC products. We assume that this will not change (much) with Post-Quantum Cryptography.

[JSADEN011] **“SESIP Profile for PSA Certified™ Level 3”**
PSA-RoT: 35 person-days of AVA_VAN.3 activities.

[PP-0084] **“Security IC Platform Protection Profile”**
EAL 4 augmented by AVA_VAN.5 and ALC_DVS.2

[PP-0117] **“Secure Sub-System in System-on-Chip (3S in SoC)”**
EAL 4 augmented by ATE_DPT.2, AVA_VAN.5, ALC_DVS.2, ALC_FLR.2

AVA_VAN: CC Vulnerability Analysis

Attack Potential is evaluated with a score-based system that roughly maps to the "**cost of attack**" (think $€£.)

Considers attack **Identification + exploitation**, with many factors:

- Elapsed time (hours–months)
- Attacker Expertise (multiple)
- Knowledge (how restricted)
- Access to the TOE (samples)
- Equipment (common/bespoke)

("Application of Attack Potential" docs.)

AVA_VAN.1 Vulnerability Survey
- TOE resistance against BASIC Attack Potential (0-15)

AVA_VAN.2 (Unstructured) Vuln. Analysis
- TOE resistance against BASIC Attack Potential (16-20)

AVA_VAN.3 Focused (Unstructured) Vuln. Analysis
- TOE resistance against ENHANCED-BASIC AP (21-24)

AVA_VAN.4 Methodical Vuln. Analysis
- TOE resistance against MODERATE AP (25-30)

AVA_VAN.5 Advanced Methodical Vuln. Analysis
- TOE resistance against HIGH Attack Potential (31-)

Attack Potential: Example

AP Component	Identification	Exploitation
Elapsed time	2 (< one week)	6 (< one month)
Expertise	5 (expert)	4 (expert)
Knowledge of the TOE	4 (sensitive)	0 (public)
Access to the TOE	0 (< 10 samples)	0 (< 10 samples)
Equipment	3 (specialized)	4 (specialized)
Open Samples	0 (public)	0 (public)
Total	28 (AVA_VAN.4 / moderate AP range)	

SOG-IS: “Application of Attack Potential to Smartcards and Similar Devices”

https://www.sogis.eu/documents/cc/domains/sc/JIL-Application-of-Attack-Potential-to-Smartcards-v3.2.1.pdf

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 $\mathbf{c'=cr^e}$ (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

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 **$O(nt^2)$** 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]] ⊕ [[B]]**
(*no share mixing!)*

X0 = A0 ⊕ B0
X1 = A1 ⊕ B1

SecAnd: **[[X]] = [[A]] ⋀ [[B]]**
(R is a random bit.)

X0 = (A0 ⋀ B0) ⊕ R ⊕ (A0 ⋀ B1)
X1 = (A1 ⋀ B0) ⊕ R ⊕ (A1 ⋀ 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, **s** and **e** are secret. No need to mask resulting **t** !

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. To generate them, Kyber and Dilithium require a lot of *mixing Arithmetic and Boolean masking*.

[detail] **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 **k** × **l** × **n.**
- Public key compression (bit dropping): d = 13 bits.
- Challenge distribution has τ non-zero ±1 coefficients and (n-τ) zero coefficients.
- The secret key distribution is *uniform* but in very short range [-η, +η].
- 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
(L2) **ML-DSA-44**	(4, 4)	39	2	2^{17}	88	78	88	4.3	2^{123}	2^{112}
(L3) **ML-DSA-65**	(6, 5)	49	4	2^{19}	32	196	55	5.1	2^{182}	2^{165}
(L5) **ML-DSA-87**	(8, 7)	60	2	2^{19}	32	120	75	4.0	2^{252}	2^{229}

[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. $\hat{\mathbf{A}}$ ← ExpandA(ρ) // *Public $\hat{\boldsymbol{A}}$ has size $k \times l \times R_q$, derived from ρ.*
04. $\mathbf{s_1}$ ← ExpandS(ρ', 0,2,..,l-1) // *Secret $\boldsymbol{s_1}$ has size $l \times R_q$, distribution $[-\eta, +\eta]$.*
    $\mathbf{s_2}$ ← ExpandS(ρ', l, ..,l+k-1) // *Secret $\boldsymbol{s_2}$ has size $k \times R_q$, distribution $[-\eta, +\eta]$.*
05. $\mathbf{t}$ ← $\mathbf{A} \cdot \mathbf{s_1} + \mathbf{s_2}$ // *All of $\boldsymbol{t}$ is secure. $\boldsymbol{A} \cdot \boldsymbol{s_1} = NTT^{-1}(\hat{\boldsymbol{A}} \circ NTT(\boldsymbol{s_1}))$.*
06. $(\mathbf{t_1}, \mathbf{t_0})$ ← Power2Round($\mathbf{t}$, d) // *Split $\boldsymbol{t}$; $\boldsymbol{t_1}$ high 13 bits, $\boldsymbol{t_0}$ 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. $\hat{\mathbf{A}} \leftarrow$ ExpandA(ρ) // *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(κ, μ)] // *[Use hash in deterministic signing.]*
13. while $(\mathbf{z}, \mathbf{h}) = \perp$ do: // — REJECTION LOOP —
14. | $\mathbf{y} \leftarrow$ ExpandMask(ρ', κ..) // ***y*** *is $l \times R_q$ sampled from $[-\gamma_1, +\gamma_1]$.*
15. | $\mathbf{w} \leftarrow \mathbf{A}*\mathbf{y}$ // *Compute as* ***w*** $= NTT^{-1}(\hat{\mathbf{A}} \circ NTT(\mathbf{y}))$.
16. | $\mathbf{w}_1 \leftarrow \text{HighBits}_q(\mathbf{w}, 2\gamma_2)$ // ***w_1*** *range is $(q-1)/2\gamma_2$ so [0,15] or [0,43].*
17. | $\tilde{c} \leftarrow H(\mu, \mathbf{w}_1)$ // $\tilde{c}$ *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. | $\mathbf{r_0} \leftarrow$ LowBits($\mathbf{w} - \mathbf{c}*\mathbf{s_2}, 2\gamma_2$) // *Range is basically* $\pm 2\gamma_2$
21. | if MaxAbs($\mathbf{z}$) $\geq \gamma_1 - \beta$ or MaxAbs($\mathbf{r_0}$) $\geq \gamma_2 - \beta$ then: ($\mathbf{z}$, $\mathbf{h}$) $\leftarrow \perp$ // *reject*
22. | else:
23. | $\mathbf{h} \leftarrow$ MakeHint($-\mathbf{c} * \mathbf{t_0}$, $\mathbf{w} - \mathbf{c}*\mathbf{s_2} - \mathbf{c} * \mathbf{t_0}, 2\gamma_2$) // $\mathbf{h} \in \{0,1\}^{kN}$
24. | if MaxAbs($\mathbf{c} * \mathbf{t_0}$) $> \gamma_2$ or CountOnes($\mathbf{h}$) $> \omega$ then: ($\mathbf{z}$, $\mathbf{h}$) $\leftarrow \perp$ // *reject*
25. | $\kappa \leftarrow \kappa + l$ // *For creating fresh* $\mathbf{y}$ *in next iteration*

end while

26. return Sig = ($\tilde{\mathbf{c}}$, $\mathbf{z}$, $\mathbf{h}$) // *no longer secret*

- Protecting just the ($\mathbf{s_1}$, $\mathbf{s_2}$) secret itself via masking is easy; NTT in shares.
- Leaking the one-time secret $\mathbf{y}$ 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):

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

31. if MaxAbs(**z**) < γ_1-β and **c̃** = H(μ || $\mathbf{w'}_1$) and CountOnes(**h**) ≤ ω then:
    | return T 👍 "Good signature"
    else:
    | return F 👎 "Fail!"