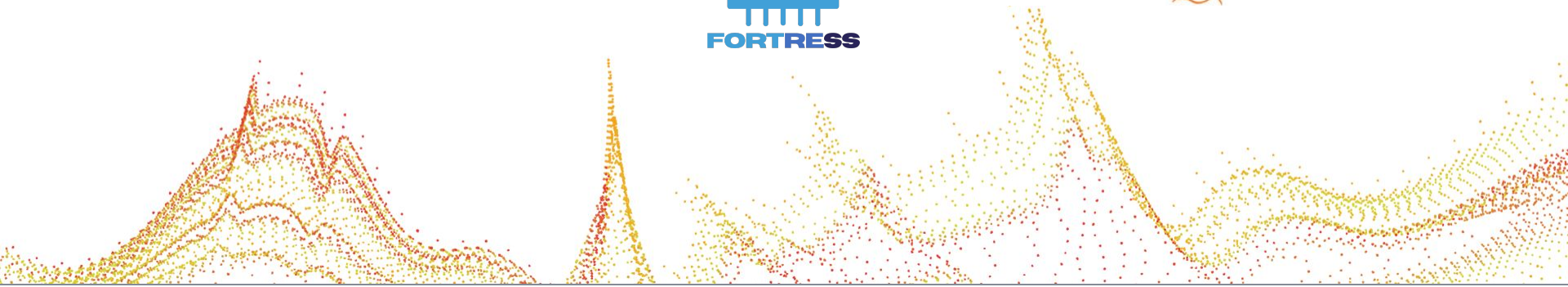
Apples, Oranges, and Signatures: Pitfalls and Methodology in ML-DSA Benchmarking

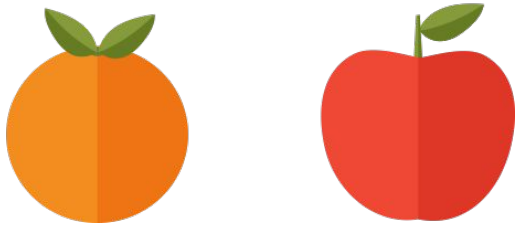
Sebastien Riou, Jong-Yeon Park, Liga Anwar, Axel Poschmann, and Michael Hutter



Problems with benchmarking in general

- Vendors vs Customers
  - Vendors want to show best numbers
  - Customers want to know real world performances
  - Customers want to compare several implementations
    - Pure Software
    - Pure Hardware
    - Hardware / Software mixed implementations

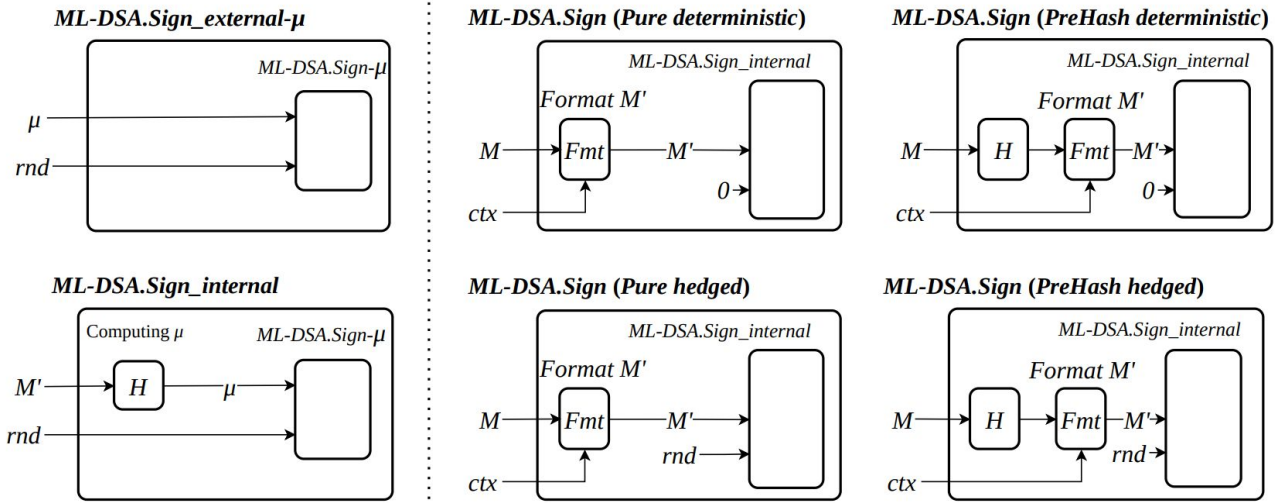
- Vendors tend to do their own thing, numbers are not comparable



- Developers need
  - Benchmarking is part of development iteration loop
  - Need to be short

ML-DSA sign overview

- 3 parameter sets: ML-DSA-44, ML-DSA-65, ML-DSA-87
- 2 modes: Hedged and Deterministic
- 4 interfaces: Pure, Pre-Hash, Internal and External-mu

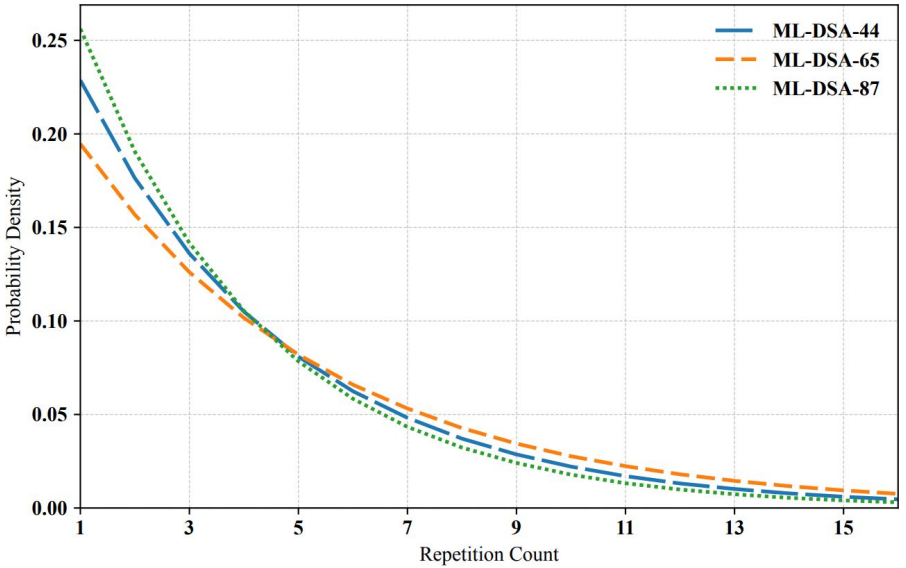


ML-DSA sign's repetition

- External-mu:
  - Decompress key
  - For **`repetition`** = 1 to 814:
    - Generate candidate signature
    - If candidate signature is secure
      - Break
  - Format signature and return it

- **`repetition`** average:
  - ML-DSA-44: 4.25
  - ML-DSA-65: 5.1
  - ML-DSA-87: 3.85

- Problem: **`repetition`** > 20 can happen



Problems with benchmarking ML-DSA sign

- Lot of variants→ambiguity about measurement boundaries
- Non deterministic execution time
  - Fixed key
  - Fixed input size
  - Use ‘DETERMINISTIC’ variant

  → you still get huge execution time variation (max/min > 20!)

- ML-DSA-87 repetitions:

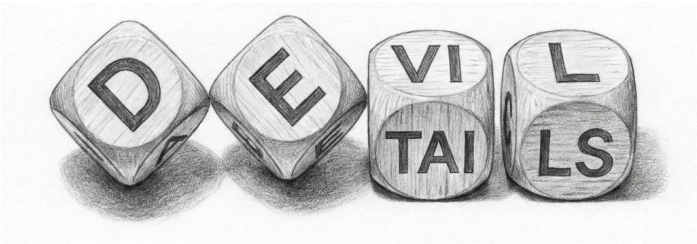
Sample size	Min. sum	Max. sum	Theoretical
10	12	108	38.5
100	270	579	385.0
1,000	3,543	4,293	3,850.0
10,000	37,979	40,143	38,500.0

How to benchmark it properly ?

- Test vectors tailor made for performance benchmarking
  - Include a best-case
  - Include high repetition count with specific occurrence probability
  - Add more to match average repetition count

- One caveat: it works only for DETERMINISTIC mode

- Benchmark key expansion and sign operation
  - Some computation can be shifted to key generation
- Benchmark all memory sizes
  - ML-DSA offers a lot of time-memory trade-offs
- Many other details explained in the paper



Generating test vectors

- Fork and instrument mldsa-native
- Low probability cases are hard to find:
  - Need consistent probability for all parameter sets
  - Search over 2 Trillions signature achieved only a probability of 2^{-37}

Algorithm	Occurrence Prob.			
	2^{-37}	2^{-39}	2^{-41}	2^{-43}
ML-DSA-44	96	-	106	-
ML-DSA-65	119	127	-	-
ML-DSA-87	86	-	-	100

- **Result: only 108 test vectors to cover the 3 parameter sets**
- Code is open source: BTVG-MLDSA
- We intend to search lower probability cases

Why 2^{-37} isn't low enough ?

- Systems with real time constraints
  - Timeout = Failure
- Automotive "V2X" for example
- Max signature/s limited by FIT budget

ASIL Level	PMHF
ASIL A	None
ASIL B	$\leq$ 100 FIT
ASIL C	$\leq$ 100 FIT
ASIL D	$\leq$ 10 FIT

- Example with 1% of FIT budget for 2^{-37} probability

ASIL Level	Max. signatures/s
ASIL B and C	0.0382
ASIL D	0.0038

Benchmarking ML-DSA on MCUs

- Open source code: crypto-benchmark
- Multi platform:
  - ARM Cortex-M3 and higher
  - RV32IMC
  - RV64IMC
  - Adding new ones is easy
- Companion repositories:
  - crypto-benchmark-stm32u5
  - crypto-benchmark-m5531
- Report as CSV and LATEX tables

- Measurements:
  - Cycle count
  - Stack size
  - Heap size
  - Static RAM size
  - Read only size
- Sizes for one parameter set
- Integration with Renode
  - Not the real cycle count
  - Good enough to compare implementations

Size results on STM32U5A5

PQSHIELD

Size for ML-DSA-87 key generation, deterministic sign and verify, KiB.

Implementation	Level	Stack	Heap	Static RAM	Read-only
pqcrystals-lowram	44	5.30	0.00	3.77	12.52
pqcrystals-lowram	65	6.80	0.00	5.84	11.96
pqcrystals-lowram	87	8.30	0.00	7.30	12.21
stm32pqc-lowram	44	1.20	0.00	14.88	17.20
stm32pqc-lowram	65	1.20	0.00	20.94	16.80
stm32pqc-lowram	87	1.20	0.00	26.41	17.04
wolfssl-lowram	44	1.98	28.01	20.32	25.94
wolfssl-lowram	65	1.98	47.01	20.32	25.94
wolfssl-lowram	87	1.98	79.01	20.32	25.94

Performance results on STM32U5A5

Performance of ML-DSA deterministic sign, 69 bytes message, 0 byte context, Million cycles.

Implementation	Level	Operation	Minimum	Average[a]	Worst observed[b]
pqcrystals-lowram	44	key-exp	3.55	3.65	3.70
pqcrystals-lowram	44	sign	6.99	22.01	446.48
pqcrystals-lowram	65	key-exp	6.90	6.90	6.90
pqcrystals-lowram	65	sign	11.50	41.12	860.33
pqcrystals-lowram	87	key-exp	11.29	11.44	11.74
pqcrystals-lowram	87	sign	18.56	53.92	1061.51
stm32pqc-lowram	44	key-exp	2.25	2.27	2.28
stm32pqc-lowram	44	sign	4.20	14.61	308.69
stm32pqc-lowram	65	key-exp	3.89	3.89	3.89
stm32pqc-lowram	65	sign	6.39	26.37	580.35
stm32pqc-lowram	87	key-exp	6.46	6.47	6.50
stm32pqc-lowram	87	sign	9.95	32.99	691.94
wolfssl-lowram	44	key-exp	2.93	2.96	2.97
wolfssl-lowram	44	sign	4.27	16.28	355.35
wolfssl-lowram	65	key-exp	5.11	5.11	5.11
wolfssl-lowram	65	sign	7.03	32.22	731.42
wolfssl-lowram	87	key-exp	8.54	8.56	8.61
wolfssl-lowram	87	sign	11.87	42.60	926.04

[a] Match long term average for `sign`. [b] Probability of occurrence is 2^{-37} for `sign`.

What about long messages ? (STM32U5A5)

PQSHIELD

Performance of ML-DSA-87 deterministic sign, 0 byte context, Million cycles.

Implementation	Level	Operation	Minimum	Average[a,c]	Worst observed[b,c]
pqcrystals-lowram	87	sign 69	18.56	53.92	1061.51
pqcrystals-lowram	87	sign 10K	20.08	55.44	1063.03
pqcrystals-lowram	87	sign 1M	171.47	206.83	1214.42
stm32pqc-lowram	87	sign 69	9.95	32.99	691.94
stm32pqc-lowram	87	sign 10K	11.01	34.05	693.01
stm32pqc-lowram	87	sign 1M	119.40	142.44	801.40
wolfssl-lowram	87	sign 69	11.87	42.60	926.04
wolfssl-lowram	87	sign 10K	13.53	44.27	927.71
wolfssl-lowram	87	sign 1M	183.19	213.92	1097.37

[a] Match long term average. [b] Probability of occurrence is 2^{-37}. [c] Extrapolated from 69-cycle baseline.

Conclusion

- ML-DSA + real time constraint is tricky
- Using dedicated test vectors is advantageous:
  - Get accurate average with low number of runs
  - Get best case
  - Get worst case for a given occurrence probability
- We made such test vectors publicly available
  - Python
  - C
  - Verilog
  - Json (following ACVP format)
- Open source code
  - Test vector search
  - Portable benchmarking framework
  - Demo with multiple SW libraries
- Help wanted: search for better test vectors (lower probability cases, need a LOT of CPU-hours)