Africacrypt 2023

Benchmarking and Analysing NIST PQC Lattice-Based Signature Scheme Standards on ARM Cortex M7

James Howe
Senior Research Scientist

SANDBOXAQ™

CONTENTS

01 Introduction and Motivation

02 Benchmark and Profile Results

03 Constant-Time Issues

SANDBOXAQ™

01

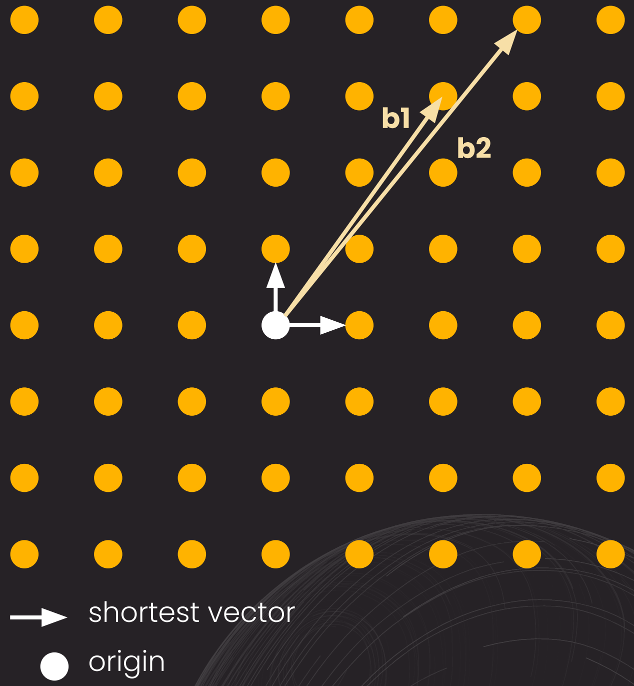
INTRODUCTION AND MOTIVATION

What are the PQC standards we have?

CRYSTALS-Kyber is the only KEM and CRYSTALS-Dilithium is the primary signature.

"The security of **Kyber** has been thoroughly analyzed [...] based on a strong framework of results in lattice-based cryptography. Kyber has excellent performance overall in software, hardware and many hybrid settings."

"Dilithium is a signature scheme with high efficiency, relatively simple implementation, a strong theoretical security basis, and an encouraging cryptanalytic history."



What are the PQC standards we have?

We also have two other PQ signatures

Falcon, also from lattices, different performance profile.

More complex implementation, emulates or uses FPU.

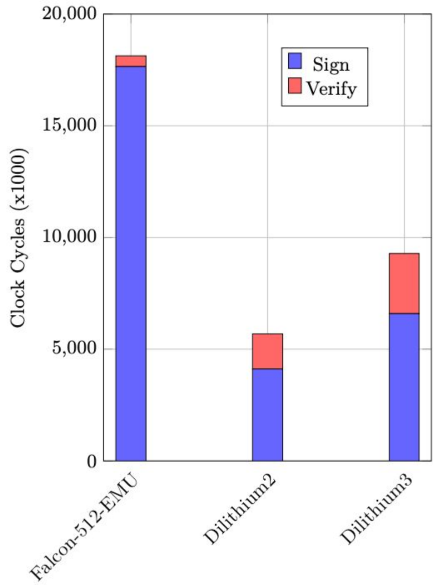
Offers significantly smaller signature sizes and fast verification.

> *“* ***Falcon*** *was chosen for standardization because NIST has confidence in its security (under the assumption that it is correctly implemented) and because its small bandwidth may be necessary in certain applications.*

The Premise

“NIST understands that some applications will not work as they are currently designed if the signature and the data being signed cannot fit in a single internet packet.”

“For this reason, NIST decided to standardize FALCON as well. Given FALCON’s overall better performance when signature generation does not need to be performed on constrained devices, many applications may prefer to use FALCON over Dilithium, even in cases in which Dilithium’s signature size would not be a barrier to implementation.”



Signature benchmarks of Dilithium and Falcon (tree) on ARM Cortex M4, using the template from [Fig. 7, AAC^{+}22].

Current State on ARM Cortex M4

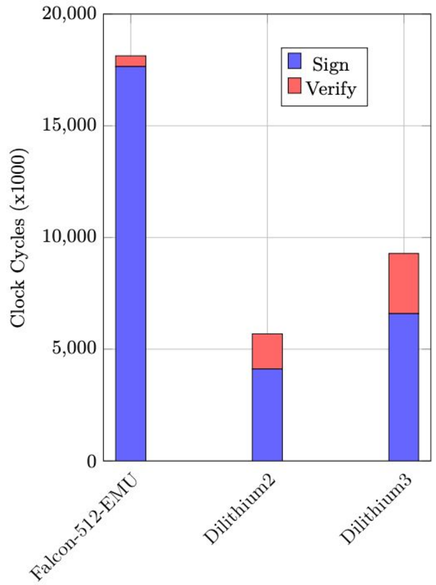
Without double precision, Falcon emulates floats.

Thus we get **performance profiles** like this ⟶

We wanted to **challenge the belief** that Falcon signing is much slower than Dilithium's.

Important decision in, e.g., RISC-V CPU and SoC implementations.

Can a full FPU implementations be constant time?



Signature benchmarks of Dilithium and Falcon (tree) on ARM Cortex M4, using the template from [Fig. 7, AAC⁺22].

What's the big deal?

Falcon's Constant-time and Correctness

01

Emulated **floating-point implementation** can be done

02

Only using integer operations with **uint32_t** and **uint64_t** types

03

This is **constant-time,** provided that the underlying platform offers constant-time opcodes for:

- Multiplication of two 32-bit unsigned integers into a 64-bit result.
- Left-shift or right-shift of a 32-bit unsigned integer by a potentially secret shift count in the 0...31 range.

Why the ARM Cortex M7?

NIST selected Cortex M4 as benchmark MCU;
and the Cortex M7 is a very similar core

Both have ARMv7-M architecture

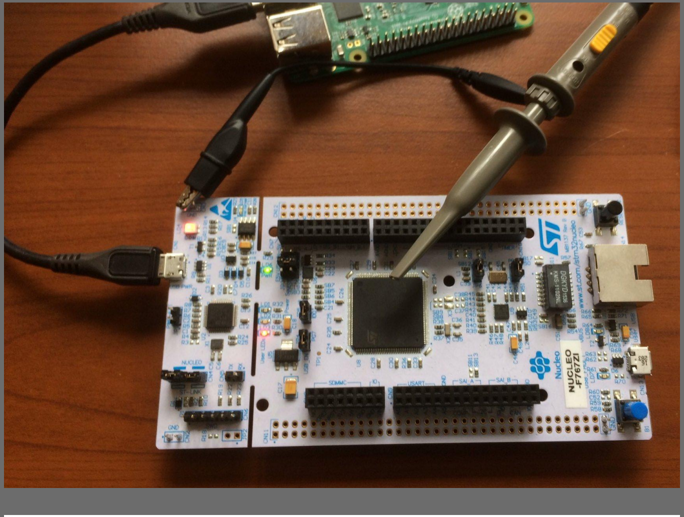
Cortex M7 has all ISA features available in the Cortex M4

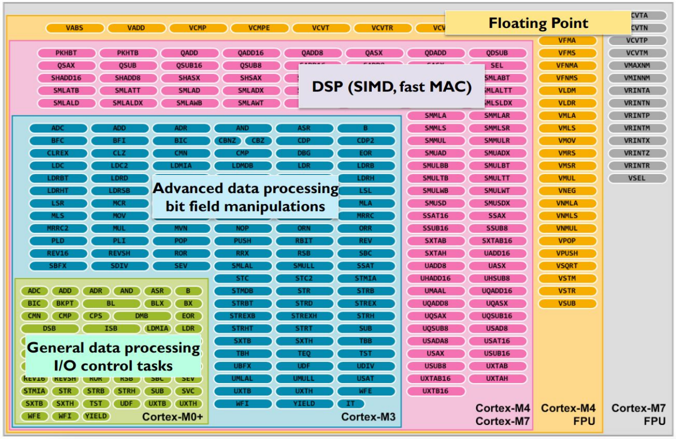
M7 has 6-stage pipeline (vs 3) and better memory features and branch predicting

M7 has 64-bit FPU, M4 has 32-bit

Falcon requires 53-bit floating-point precision

Using floating-points is rare in cryptography → side channels?





From: ARM® Cortex®-M for Beginners

02

BENCHMARKING AND PROFILING

Benchmarking Premise

- Benchmark Dilithium and Falcon on ARM Cortex M7.
- Taken from open source repos i.e., pqm4.
- Benchmarks averaged over 1000 runs.
- Presentation focus on clock cycles.
- Mainly use STM32F767ZI NUCLEO-144 board.
- Use GNU ARM embedded toolchain: v10.2.1

using -O2 -mcpu=cortex-m7 -march=-march=armv7e-m+fpv5+fp.dp

SANDBOXAQ™

Dilithium Benchmarking (M4 vs M7)

Overall, the performance of Dilithium wasn't interesting.

Improvements range between 1.09-1.19x

Essentially accounts for the slightly better MCU: Cortex M7 vs the Cortex M4.

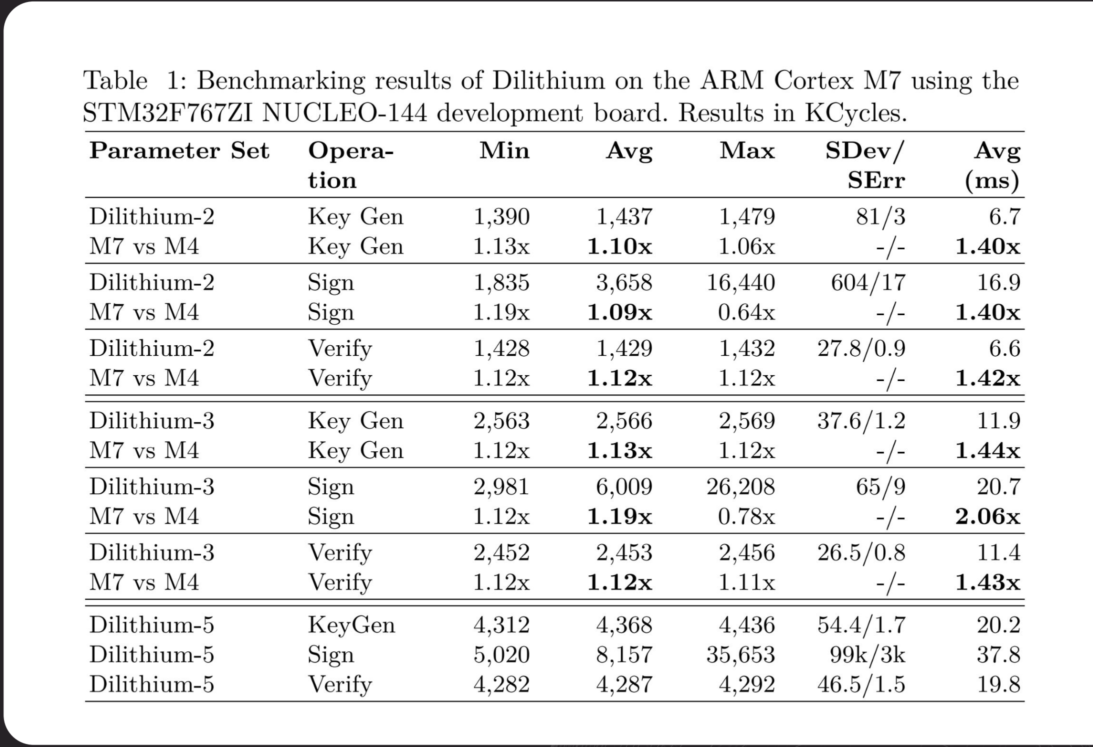


Table 1: Benchmarking results of Dilithium on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board. Results in KCycles.

Parameter Set	Operation	Min	Avg	Max	SDev/ SErr	Avg (ms)
Dilithium-2	Key Gen	1,390	1,437	1,479	81/3	6.7
M7 vs M4	Key Gen	1.13x	**1.10x**	1.06x	-/-	**1.40x**
Dilithium-2	Sign	1,835	3,658	16,440	604/17	16.9
M7 vs M4	Sign	1.19x	**1.09x**	0.64x	-/-	**1.40x**
Dilithium-2	Verify	1,428	1,429	1,432	27.8/0.9	6.6
M7 vs M4	Verify	1.12x	**1.12x**	1.12x	-/-	**1.42x**
Dilithium-3	Key Gen	2,563	2,566	2,569	37.6/1.2	11.9
M7 vs M4	Key Gen	1.12x	**1.13x**	1.12x	-/-	**1.44x**
Dilithium-3	Sign	2,981	6,009	26,208	65/9	20.7
M7 vs M4	Sign	1.12x	**1.19x**	0.78x	-/-	**2.06x**
Dilithium-3	Verify	2,452	2,453	2,456	26.5/0.8	11.4
M7 vs M4	Verify	1.12x	**1.12x**	1.11x	-/-	**1.43x**
Dilithium-5	KeyGen	4,312	4,368	4,436	54.4/1.7	20.2
Dilithium-5	Sign	5,020	8,157	35,653	99k/3k	37.8
Dilithium-5	Verify	4,282	4,287	4,292	46.5/1.5	19.8

Benchmarking Results (FPU vs EMU on M7)

Falcon expectedly sees a drastic speedup

Improvements range between >**6**-**8**x overall

Key generation is least impacted, >**1.5**x speedup overall.

Signing times show most improvements:

- Sign dynamic >**6**x speedup, close to Dilithium perf.
- Sign tree >**4.5**x speedup, comfortably faster than Dilithium

Verify not impacted, doesn't require floats.

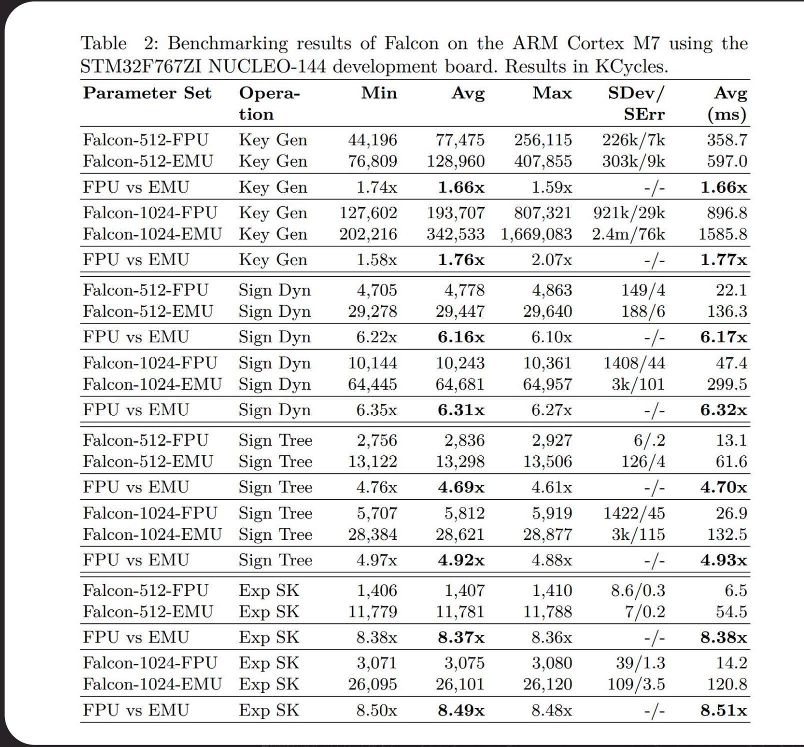


Table 2: Benchmarking results of Falcon on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board. Results in KCycles.

Parameter Set	**Operation**	**Min**	**Avg**	**Max**	**SDev/ SErr**	**Avg (ms)**
Falcon-512-FPU	Key Gen	44,196	77,475	256,115	226k/7k	358.7
Falcon-512-EMU	Key Gen	76,809	128,960	407,855	303k/9k	597.0
FPU vs EMU	Key Gen	1.74x	**1.66x**	1.59x	-/-	**1.66x**
Falcon-1024-FPU	Key Gen	127,602	193,707	807,321	921k/29k	896.8
Falcon-1024-EMU	Key Gen	202,216	342,533	1,669,083	2.4m/76k	1585.8
FPU vs EMU	Key Gen	1.58x	**1.76x**	2.07x	-/-	**1.77x**
Falcon-512-FPU	Sign Dyn	4,705	4,778	4,863	149/4	22.1
Falcon-512-EMU	Sign Dyn	29,278	29,447	29,640	188/6	136.3
FPU vs EMU	Sign Dyn	6.22x	**6.16x**	6.10x	-/-	**6.17x**
Falcon-1024-FPU	Sign Dyn	10,144	10,243	10,361	1408/44	47.4
Falcon-1024-EMU	Sign Dyn	64,445	64,681	64,957	3k/101	299.5
FPU vs EMU	Sign Dyn	6.35x	**6.31x**	6.27x	-/-	**6.32x**
Falcon-512-FPU	Sign Tree	2,756	2,836	2,927	6/.2	13.1
Falcon-512-EMU	Sign Tree	13,122	13,298	13,506	126/4	61.6
FPU vs EMU	Sign Tree	4.76x	**4.69x**	4.61x	-/-	**4.70x**
Falcon-1024-FPU	Sign Tree	5,707	5,812	5,919	1422/45	26.9
Falcon-1024-EMU	Sign Tree	28,384	28,621	28,877	3k/115	132.5
FPU vs EMU	Sign Tree	4.97x	**4.92x**	4.88x	-/-	**4.93x**
Falcon-512-FPU	Exp SK	1,406	1,407	1,410	8.6/0.3	6.5
Falcon-512-EMU	Exp SK	11,779	11,781	11,788	7/0.2	54.5
FPU vs EMU	Exp SK	8.38x	**8.37x**	8.36x	-/-	**8.38x**
Falcon-1024-FPU	Exp SK	3,071	3,075	3,080	39/1.3	14.2
Falcon-1024-EMU	Exp SK	26,095	26,101	26,120	109/3.5	120.8
FPU vs EMU	Exp SK	8.50x	**8.49x**	8.48x	-/-	**8.51x**

Profiling Falcon-512 (FPU vs EMU)

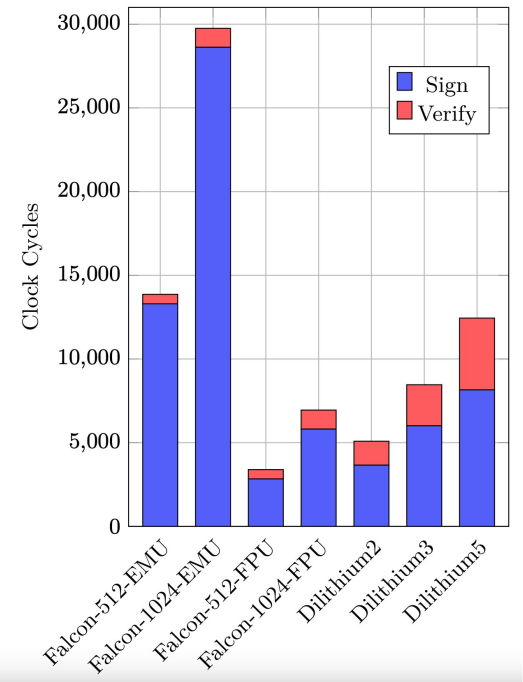
Performance improvements inside Falcon:

For key generation:

- NTRUSolve(·) improves by **1.7**x, 69m → 40m cycles.
- iFFT/FFT multiplication **16**x better, 10m → 0.5m cycles.
- FFT polynomial inversion **13**x better, 1.5m → 0.1m cycles.

Verify times were unchanged.

Expand private key improved 12x.
Going from 11m to <1m cycles.



SANDBOXAQ™

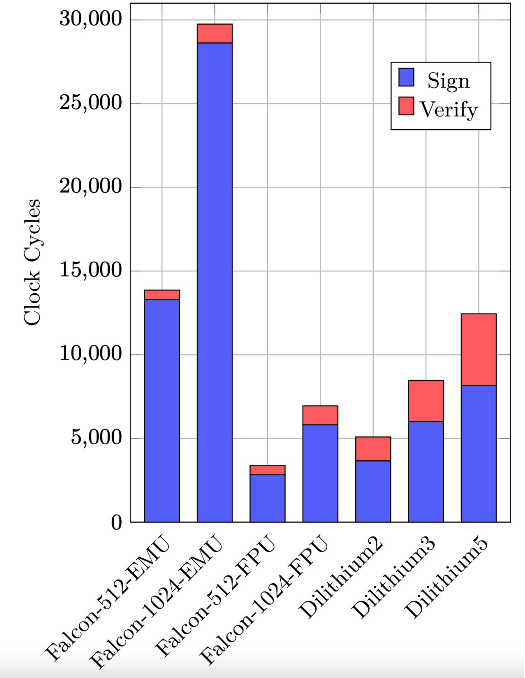
Profiling Falcon-512 (FPU vs EMU)

Performance improvements inside Falcon:

For (either) signing modes:

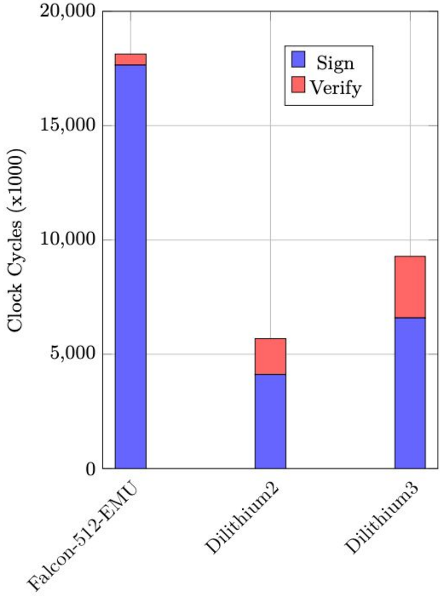
- Convert basis to FFT **16**x better, 4m → 0.2m cycles.
- FFT mult. for lattice basis **14**x better, 1.3m → 0.01m cycles.
- Fast Fourier sampling **5**x better, 16m → 3m cycles.
- Recompute basis matrix **15**x better, 4m → 0.2m cycles.
- Finding lattice point **8**x better, 3m → 0.3m cycles.

Almost all functions involve Fast Fourier.



SANDBOXAQ™

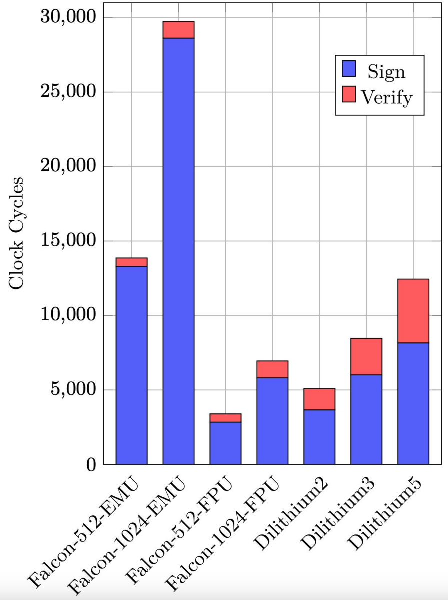
Benchmarking (Dilithium vs Falcon)



Falcon and Dilithium on the ARM Cortex M4.

Now we see a much different **performance profile**!

Falcon-512 & **Falcon-1024** signature generation now slightly faster than **Dilithium2** & **Dilithium5**!



Falcon and Dilithium on the ARM Cortex M7.

03

CONSTANT OR ISOCHRONOUS RUNTIME

Constant-Time Validation

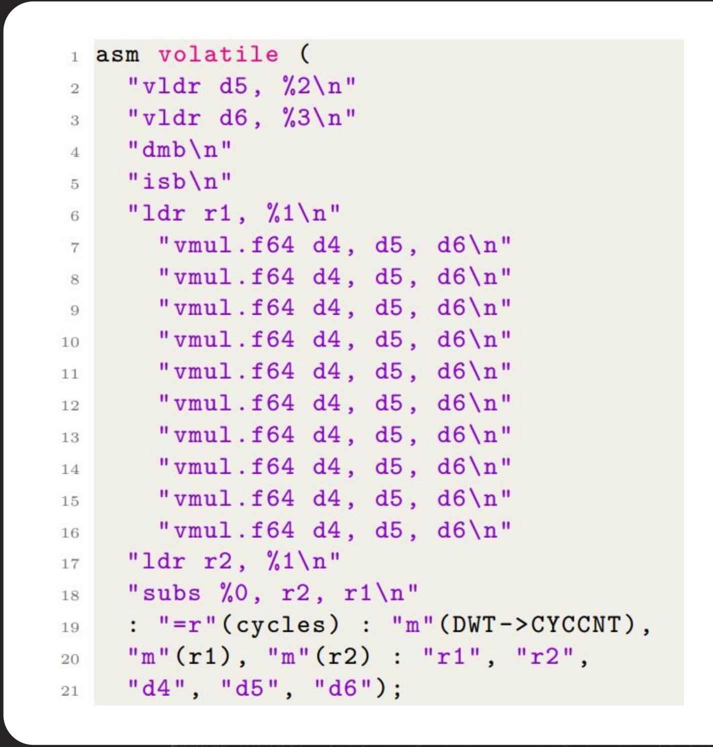
Floating-point arithmetic is rare in cryptography!
Thus we thought it was worth looking at…

We used inline assembly to

- Minimize the unwanted optimizations from the compiler / clobbered registers where necessary.
- This minimizes the effect of surrounding instructions on the operations of interest.
- Which occurred when we tried using C.
- Ensures that all execution is from cache.

This example is for double precision multiplication, i.e., vmul.f64, this is repeated for each instruction.

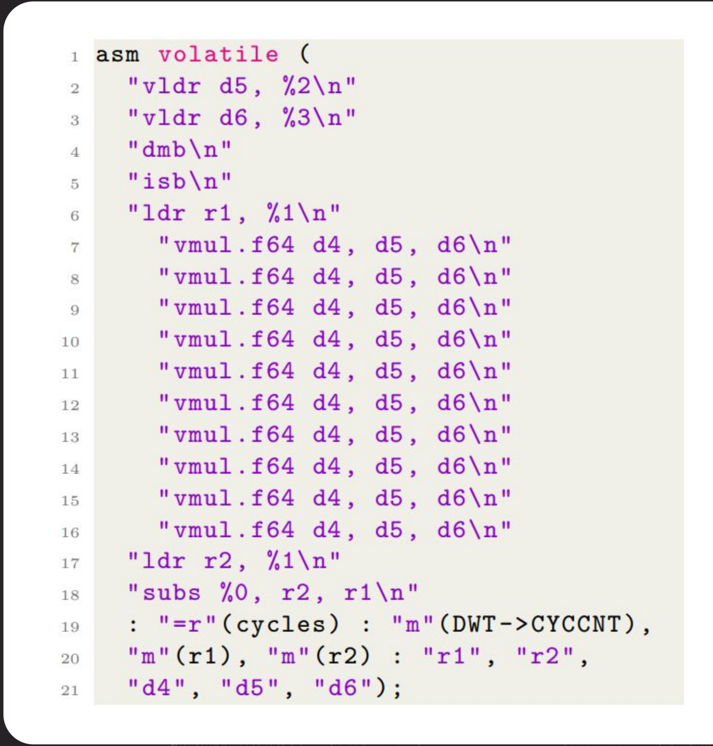
We tested 4 STM32 development boards.



```
asm volatile (
  "vldr d5, %2\n"
  "vldr d6, %3\n"
  "dmb\n"
  "isb\n"
  "ldr r1, %1\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
  "ldr r2, %1\n"
  "subs %0, r2, r1\n"
  : "=r"(cycles) : "m"(DWT->CYCCNT),
  "m"(r1), "m"(r2) : "r1", "r2",
  "d4", "d5", "d6");
```

Constant-Time Validation

- Assembly code uses **two random inputs** for each function.
- We found **timing issues** in all double precision FPU instructions across all 4 STM32 boards.
- In addition (vadd.f64) runtimes had **16 clocks on avg, standard deviation of 4.1.**
- Random values in same range (same exponents) had **constant runtime at 10 clock cycles.**
- From two different exponent ranges we observed **constant runtime at 19 clock cycles.**
- If one value is zero, **instruction was 'skipped'**.



```
asm volatile (
  "vldr d5, %2\n"
  "vldr d6, %3\n"
  "dmb\n"
  "isb\n"
  "ldr r1, %1\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
    "vmul.f64 d4, d5, d6\n"
  "ldr r2, %1\n"
  "subs %0, r2, r1\n"
  : "=r"(cycles) : "m"(DWT->CYCCNT),
  "m"(r1), "m"(r2) : "r1", "r2",
  "d4", "d5", "d6");
```

SANDBOXAQ™

Constant-Time Validation

Also tested the ARM Cortex A53 as a previous paper uses Raspberry Pi 3.

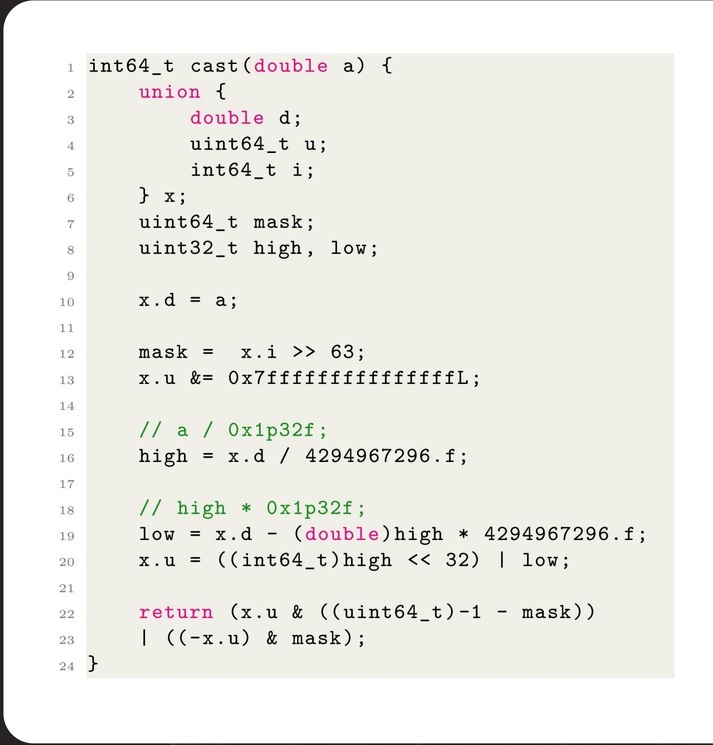
Issue found when casting from types **double** to **int64_t**, op rounds towards zero.

No native instruction to do this on ARMv7.

This can be non-constant time.

In LLVM, it isn't, and leaks the sign.

We reported this to the Falcon team and proposed the following fix shown on the right.



```
int64_t cast(double a) {
    union {
        double d;
        uint64_t u;
        int64_t i;
    } x;
    uint64_t mask;
    uint32_t high, low;

    x.d = a;

    mask =  x.i >> 63;
    x.u &= 0x7fffffffffffffffL;

    // a / 0x1p32f;
    high = x.d / 4294967296.f;

    // high * 0x1p32f;
    low = x.d - (double)high * 4294967296.f;
    x.u = ((int64_t)high << 32) | low;

    return (x.u & ((uint64_t)-1 - mask))
    | ((-x.u) & mask);
}
```

Takeaways

01 Falcon is super fast on the ARM Cortex M7.

02 Beware of timing issues, for all platforms.

03 Users should consider this thoroughly for all use cases.

For example
Cloudflare currently recommend using Falcon in offline situations.

SANDBOXAQ™

THANK YOU

Paper:
https://eprint.iacr.org/2022/405

GitHub:
https://github.com/jameshoweee/falcon-fpu