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

James Howe
SandboxAQ, Palo Alto, USA.
james.howe@sandboxaq.com

Bas Westerbaan
Cloudflare, Amsterdam, The Netherlands.
bas@westerbaan.name

***Abstract*—This paper presents a thorough analysis of the two lattice-based digital signature schemes, Dilithium and Falcon, which have been chosen by NIST for standardization, on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board. This research is motivated by the Cortex M7 device being the only processor in the Cortex-M family to offer a double precision (i.e., 64-bit) floating-point unit, making Falcon's implementations, requiring 53 bits of double precision, able to fully run native floating-point operations without any emulation. When benchmarking natively, Falcon shows significant speed-ups between 6.2-8.3x in clock cycles, 6.2-11.8x in runtime, however Dilithium does not show much improvement other than those gained by the slightly faster processor. We then present profiling results of the two schemes on the Cortex M7 to show their respective bottlenecks and operations where the improvements are and can be made, which show some operations in Falcon's procedures observe speed-ups by an order of magnitude. Finally, since Falcon's use of floating points is so rare in cryptography, we test the native FPU instructions on 4 different STM32 development boards with Cortex M7 and also a Raspberry Pi 3 which is used in some of Falcon's benchmarking results. We find constant-time irregularities in all of these devices, which should cause concern when using Falcon is certain use cases and on certain devices.**

I. Introduction

Since NIST began their Post-Quantum Cryptography (PQC) Standardization Project [1] there have been a number of instances where they have called for benchmarking and evaluations of the candidates on differing hardware platforms [2]–[4]. This prompted research into implementing these schemes on a variety of platforms in software (see PQClean [5], SUPERCOP [6], liboqs [7], and pqm4 [8]) and also in hardware [9]–[20].

In July 2022, NIST announced in their Round 3 status report [21] that their first set of PQC standards; one Key Encapsulation Mechanism (KEM) called CRYSTALS-Kyber [22], and three digital signature schemes called CRYSTALS-Dilithium [23], Falcon [24], and SPHINCS$^+$ [25], with three of the four of these being from the family of lattice-based cryptography.

In their Round 2 status report [4], NIST encouraged "more scrutiny of Falcon's implementation to determine whether the use of floating-point arithmetic makes implementation errors more likely than other schemes or provides an avenue for side-channel attacks". In this paper we look to bridge this gap by benchmarking, profiling, and analysing Falcon, as well as Dilithium on the ARM Cortex M7. We choose this specific microcontroller for two reasons. Firstly, as it is very similar to the ARM Cortex M4, which was chosen by NIST as the preferred benchmarking target to enable fair comparisons. Secondly, the ARM Cortex M7 is the only processor in the Cortex-M family to offer sufficient double floating-point instructions, via a 64-bit floating-point unit (FPU), useful to Falcon's key generation and signing procedures. We use publicly available[1] code from the Falcon submission package and we take the Dilithium implementation from pqm4.

Falcon's round 3 code, similar to the round 2 version [26], provides support for embedded targets (i.e., the ARM Cortex M4) which can use either emulated floating-point operations (`FALCON_FPEMU`) or native floating-point operations (`FALCON_FPNATIVE`). For Dilithium, we use the code available on the pqm4 repository (which performed better than the code on PQClean). Code designed for the Cortex M3 and Cortex M4 processors is compatible with the Cortex M7 processor as long as it does not rely on bit-banding [27].

A. Contributions

In Section III, we benchmark Dilithium and Falcon on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board, using 1,000 executions per scheme and providing minimum, average, and maximum clock cycles, standard deviation and standard error, and average runtime (in milliseconds). For Falcon, we provide benchmarks for key generation, sign dynamic, sign tree, verify, and expand private key operations. We provide these results for both native (double precision) and emulated floating-point operations and proving comparisons between these and those results publicly available on the ARM Cortex M4. We also provide results for Falcon-1024 sign tree, which does not fit on the Cortex M4.

For Dilithium, we benchmark the code from the pqm4 repository and in the same manner provide comparative results of Cortex M4 vs M7 performances. We also provide results for Dilithium's highest parameter set, which does not fit on the Cortex M4.

[1] See https://falcon-sign.info/

In Section IV, we profile Dilithium and Falcon to find their performance bottlenecks on the ARM Cortex M7, providing averages using 1,000 executions of each scheme. Specifically for Falcon, we provide what operations and functions benefit from using the board's 64-bit FPU the most. Indeed, we compare the profiling results using the Cortex M7's FPU against the profiling results on the same board where floating-point operations are emulated (as it does on the ARM Cortex M4). For Dilithium, we cannot compare this way (since it does not require floating points) and so we provide plain profiling results.

The code used in this paper is available at the following link: https://github.com/jameshoweee/falcon-fpu.

II. Background

Dilithium and Falcon are two of the three signature scheme chosen by NIST and PQC standards. Dilithium is the primary signature scheme and is based on the Fiat–Shamir with aborts paradigm, with its hardness relying on the decisional module-LWE and module-SIS problems. In the third round, Dilithium offered three parameter sets satisfying the NIST security levels 2, 3, and 5 for being at least as hard to break as SHA-256, AES-192, and AES-256, respectively. Dilithium benefits from using the same polynomial ring ($\mathbb{Z}_q[X]/(X^n + 1)$) with a fixed degree ($n = 256$) and modulus ($q = 8380417$) and only requires sampling from the uniform distribution, making its implementation significantly simpler than for Falcon. Dilithium's performance profile offers balance for the core operations (key generation, signing, and verifying) and also key and signature sizes. Furthermore, Dilithium can be implemented with a relatively small amount of RAM [28].

Falcon is based on the hash-then-sign paradigm over lattices, with its hardness relying on the NTRU assumption. In the third round, Falcon offered two parameter sets (for degree $n = 512$ and 1024) satisfying the NIST security levels 1 and 5 for being as hard to break as AES-128 and AES-256. Compared with Dilithium, Falcon is significantly more complex; relying on sampling over non-uniform distributions, with floating-point operations, and using tree data structures. However, Falcon benefits from having much smaller public key and signature sizes, while having similar signing and verification times. For more information on the details of these schemes, the reader is pointed to the specifications of Dilithium [23] and Falcon [24].

We benchmark Dilithium and Falcon on a 32-bit ARM Cortex M7 to mainly observe how much faster these signature schemes are on this device, compared to the ARM Cortex M4, and more specifically, to see the performances of Falcon using the ARM Cortex M7's 64-bit FPU. NIST decided on the ARM Cortex M4[2] as the preferred microcontroller target in order to make comparisons between each candidate easier. The ARM Cortex M4 and M7 are fairly similar cores; the M7 has all the ISA features available in the M4. However, the M7 offers additional support for double-precision floating point, a six stage (vs. three stage on the M4) instruction pipeline, and memory features like cache and tightly coupled memory (TCM). More specific differences are that the M7 will have faster branch predicting, plus it has two units for reading data from memory making it twice that of the M4.

The evaluation board we used for the benchmarking and profiling in this paper is the STM32 Nucleo-144 development board with STM32F767ZI MCU[3] which implements the ARMv7E-M instruction set. This is the extension of ARMv7-M that supports DSP type instructions (e.g., SIMD). The development board has a maximum clock frequency of 216 MHz, 2 MB of flash memory, 512 KB of SRAM. On the Cortex M7, the floating point architecture is based on FPv5, rather than FPv4 in Cortex-M4, so it has a few additional floating point instructions. We later utilize three more STM32 development boards (STM32H743ZI, STM32H723ZG, and STM32F769I-DISCO) and a Raspberry Pi 3 in order to check the constant runtime of Falcon more thoroughly.

All results reported in this paper used the GNU ARM embedded toolchain 10-2020-q4-major, i.e. GCC version 10.2.1 20201103, using optimization flags `-O2 -mcpu=cortex-m7 -march= -march=armv7e-m+fpv5+fp.dp`. All clock cycle results were obtained using the integrated clock cycle counter (`DWT->CYCCNT`).

III. Benchmarking on ARM Cortex M7

This section presents the results of benchmarking Dilithium (Table I) and Falcon (Table II) on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board. The values presented in the following tables are iterated over 1,000 runs of the operation. As noted previously, we provide results that are not available on the Cortex M4; Falcon-1024 sign tree and Dilithium for parameter set five.

The tables report minimum, average, and maximum clock cycles, as well as the standard deviation and standard error of the clock cycles, and the overall runtime in milliseconds clocked at 216 MHz. We run these benchmarks for each scheme's operation (e.g., verify) and for all parameter sets. Below each benchmarking row is a metric comparing the results on the Cortex M4 via pqm4 (where available). Specific in the Falcon benchmarking however is another comparison metric to illustrate the performance gains of its operations using the Cortex M7's native 64-bit FPU.

The remaining details provide stack usage (Tables III and V) and RAM usage (Tables IV and VI) of the two signature schemes.

A. Stack Usage and RAM Size

Tables III and V show stack usage of Dilithium and Falcon and Tables IV and VI show the RAM usage of Dilithium and Falcon on ARM Cortex M7. We calculate the stack usage by

[2] See the NIST PQC forum: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/cJxMq0_90gU/m/qbGEs3TXGwAJ

[3] https://www.st.com/en/evaluation-tools/nucleo-f767zi.html.

Table I: Benchmarking results of Dilithium on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board.

Parameter Set	**Operation**	**Min**	**Avg**	**Max**	**SDev/Err**	**Avg**
		(KCyc)	**(KCyc)**	**(KCyc)**	**(KCyc)**	**(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

using the `avstack.pl`[4] tool, adapted to the ARM toolchain, and RAM was calculated using `meminfo`. Note that the implementations we benchmarked weren't optimized for low memory usage. Dilithium, for one, can be used in much more memory constrained environments than these numbers here suggest [28].

IV. Profiling on ARM Cortex M7

This section presents the profiling results of Dilithium and Falcon on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board. Firstly, we provide Figures 1 and 2 profiling the acceptance rates of Dilithium's sign and Falcon's key generation procedures. Next, we profile the inner workings of Dilithium (Table VIII) and Falcon (Table VII).

A. *Rate of Acceptence in Dilithium and Falcon*

The following figures illustrate the effective 'rejection rates' or 'restart rates' of Dilithium's signing (Figure 1) and Falcon's key generation (Figure 2) procedures. Restart or rejection rates are shown in the figures' x-axis, with probabilities of acceptance shown in the y-axis.

B. *Profiling Results of Dilithium and Falcon*

The values presented in the following tables are iterated over 1,000 runs of the main operation (e.g., verify). As noted previously, for comparison, we provide profiling results for Falcon both with and without use of the FPU, and also provide the improvements over the results on the Cortex M4 provided in pqm4. For Dilithium, we only provide comparisons with pqm4 as it does not benefit at all from the FPU. Some lines of the tables will appear incomplete due to the fact that either that operation did not fit on the Cortex M4 (i.e., Falcon-1024 sign tree) or those results were not reported by pqm4 (i.e., Falcon's expand private key).

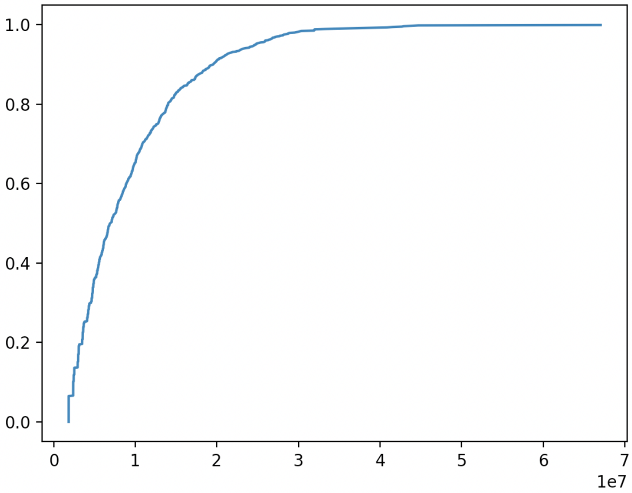

Figure 1: Dilithium's signing.

As expected, a significant amount of time is spent on the generation of uniform randomness in both scheme's key generation and signing procedures. In Dilithium, we see this in the `expand matrix` and in `sample vector` type operations, slightly increasing, as expected, as the parameter sets increase.

For Falcon, the `poly small mkgauss` and `ffsampling` similarly consume significant amounts of clock cycles for generating randomness. However, for `ffsampling` we see significant improvements using the FPU as this operation intensively uses floating-points for Gaussian sampling [29] used for randomization. The FPU also enables significant speedups in the FFT multiplier used in key generation and signing.

[4] https://dlbeer.co.nz/oss/avstack.html.

Table II: Benchmarking results of Falcon on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board.

Parameter Set	**Operation**	**Min (KCyc)**	**Avg (KCyc)**	**Max (KCyc)**	**SDev/Err (KCyc)**	**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**
M7 vs M4	Key Gen	2.32x	**2.21x**	2.26x	-/-	**2.84x**
Falcon-1024-FPU	Key Gen	127,602	193,707	807,321	921k/29k	896.8
Falcon-1024-EMU	Key en	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**
M7 vs M4	Key Gen	2.14x	**2.56x**	1.71x	-/-	**3.41x**
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**
M7 vs M4	Sign Dyn	8.24x	**8.16x**	8.07x	-/-	**11.66x**
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**
M7 vs M4	Sign Dyn	8.36x	**8.31x**	8.19x	-/-	**11.80x**
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**
M7 vs M4	Sign Tree	6.33x	**6.23x**	6.10x	-/-	**9.61x**
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**
Falcon-512-FPU	Verify	558	559	561	2.9/0.1	2.6
Falcon-512-EMU	Verify	561	565	570	98/3	2.6
FPU vs EMU	Verify	1.01x	**1.01x**	1.02x	-/-	**1.0x**
M7 vs M4	Verify	0.83x	**0.85x**	0.86x	-/-	**1.16x**
Falcon-1024-FPU	Verify	1,135	1,136	1,141	23/0.7	5.3
Falcon-1024-EMU	Verify	1,129	1,130	1,135	6.4/0.2	5.2
FPU vs EMU	Verify	0.99x	**0.99x**	0.99x	-/-	**0.98x**
M7 vs M4	Verify	0.85x	**0.86x**	0.87x	-/-	**1.16x**

Table III: Dilithium stack usage in bytes.

Parameter Set	**Key Gen**	**Sign**	**Verify**
Dilithium-2	38,444	52,052	36,332
Dilithium-3	60,972	79,728	57,836
Dilithium-5	97,836	122,708	92,908

Table IV: Dilithium RAM usage in bytes.

Parameter Set	**Key Gen**	**Sign**	**Verify**	**Overall**
Dilithium-2	9,627	13,035	9,107	13,035
Dilithium-3	15,259	19,947	14,483	19,947
Dilithium-5	24,475	30,699	23,251	30,699

Table V: Falcon stack usage in bytes.

Parameter Set	**Key Gen**	**Sign Dyn**	**Sign Tree**	**Verify**
Falcon-512-FPU	1,156	1,920	1,872	556
Falcon-1024-FPU	1,156	1,920	1,872	556
Falcon-512-EMU	1,068	1,880	1,824	556
Falcon-1024-EMU	1,068	1,880	1,872	556

Table VI: Falcon RAM usage in bytes.

Parameter Set	**Key Gen**	**Sign Dyn**	**Sign Tree**	**Verify**	**Overall (Dyn)**	**Overall (Tree)**
Falcon-512-FPU	18,512	42,488	85,512	6,256	63,384	133,048
Falcon-1024-FPU	36,304	84,216	178,440	12,016	125,976	273,464
Falcon-512-EMU	18,512	42,488	85,512	6,256	63,384	133,048
Falcon-1024-EMU	36,304	84,216	178,440	12,016	125,976	273,464

Table VII: Profiling Falcon on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board. All values reported are in KCycles.

Key Generation	**512-FPU**	**512-EMU**	**Vs.**	**1024-FPU**	**1024-EMU**	**Vs.**
total ntru gen	77,095 (99%)	127,828 (100%)	**1.66x**	186,120 (100%)	332,876 (100%)	**1.79x**
—poly small mkgauss	34,733 (45%)	34,805 (27%)	1.00x	56,509 (30%)	57,033 (17%)	1.00x
—poly small sqnorm	28 (0.04%)	29 (0.02%)	1.04x	94 (0.05%)	94 (0.03%)	1.00x
—poly small to fp	40 (0.05%)	306 (0.24%)	**7.65x**	132 (0.07%)	989 (0.30%)	**7.50x**
—fft multiply	609 (0.80%)	10,496 (8%)	**17.2x**	2,277 (1%)	38,681 (12%)	**17.00x**
—poly invnorm2 fft	110 (0.14%)	1,446 (1%)	**13.2x**	421 (0.22%)	4,777 (1%)	**11.00x**
—poly adj fft	23 (0.03%)	12 (0.01%)	0.52x	70 (0.04%)	43 (0.01%)	0.60x
—poly mulconst	69 (0.09%)	354 (0.28%)	**5.13x**	218 (0.12%)	1,168 (0.35%)	**5.36x**
—poly mul autoadj fft	63 (0.08%)	383 (0.30%)	**6.08x**	237 (0.13%)	1272 (0.38%)	**5.37x**
—ifft multiply	683 (0.90%)	10,666 (8%)	**15.6x**	2,544 (1.36%)	39,071 (12%)	**15.4x**
—bnorm/fpr add	14 (0.02%)	184 (0.14%)	**13.1x**	35 (0.02%)	424 (0.13%)	**12.1x**
—compute public key	383 (0.49%)	383 (0.30%)	1.00x	887 (0.50%)	887 (0.27%)	1.00x
—solve ntru:	40,337 (52%)	68,764 (54%)	**1.70x**	122,696 (66%)	188,438 (56%)	**1.54x**
encode priv key	26 (0.03%)	26 (0.02%)	1.00x	52 (0.03%)	52 (0.02%)	1.00x
recomp sk and encode	384 (0.50%)	385 (0.3%)	1.00x	815 (0.44%)	815 (0.24%)	1.00x
Signing Dynamic	**512-FPU**	**512-EMU**	**Vs.**	**1024-FPU**	**1024-EMU**	**Vs.**
sign start	4 (0.08%)	4 (0.01%)	1.00x	4 (0.04%)	4 (0.01%)	1.00x
decode/comp priv key	488 (11%)	489 (1.69%)	1.00x	1,040 (11%)	1,040 (2%)	1.00x
hash mess to point	<1 (0.01%)	<1 (0.00%)	0.10x	<1 (0.00%)	<1 (0.00%)	1.00x
signature encode	11 (0.26%)	11 (0.04%)	1.00x	22 (0.24%)	22 (0.03%)	1.00x
convert basis to fft	241 (6%)	3,885 (13%)	**16.1x**	549 (6%)	8,751 (14%)	**15.9x**
comp gram matrix	67 (2%)	628 (2%)	**9.37x**	167 (2%)	1,290 (2%)	**7.72x**
apply lattice basis	89 (2%)	1,250 (4%)	**14.0x**	207 (2%)	2,756 (4%)	**13.3x**
ffsampling	2,814 (66%)	16,190 (56%)	**5.75x**	6,009 (65%)	35,324 (56%)	**5.88x**
recomp matrix basis	258 (6%)	3,900 (14%)	**15.1x**	586 (6%)	8,787 (14%)	**15.0x**
get lattice point	314 (7%)	2,527 (9%)	**8.05x**	706 (8%)	5,564 (8%)	**7.88x**
Signing Tree	**512-FPU**	**512-EMU**	**Vs.**	**1024-FPU**	**1024-EMU**	**Vs.**
sign start	4 (0.08%)	4 (0.03%)	1.00x	4 (0.07%)	4 (0.07%)	1.0x
get deg/check params	<1 (0.00%)	<1 (0.00%)	1.00x	<1 (0.00%)	<1 (0.00%)	1.0x
hash mess to point	<1 (0.01%)	<1 (0.00%)	1.00x	<1 (0.00%)	<1 (0.00%)	1.0x
sig encode	11 (0.46%)	11 (0.09%)	1.00x	22 (0.44%)	22 (0.08%)	1.00x
apply lattice basis	89 (3.70%)	1,255 (10%)	**14.1x**	194 (4%)	2,746 (9.87%)	**14.1x**
apply ff sampling	1,975 (82%)	9,081 (70%)	**4.60**x	406 (82%)	4,094 (82%)	**10.1**x
get lattice point	314 (13%)	2,527 (20%)	**8.05x**	706 (14%)	5,564 (14%)	**7.88x**
compute signature	135 (6%)	23 (0.18%)	0.17x	272 (5%)	46 (0.17%)	0.17x
Verifying	**512-FPU**	**512-EMU**	**Vs.**	**1024-FPU**	**1024-EMU**	**Vs.**
verf start	<1 (0.06%)	<1 (0.06%)	1.00x	<1 (0.03%)	<1 (0.00%)	1.00x
get degree via pk	<1 (0.01%)	<1 (0.01%)	1.00x	<1 (0.00%)	<1 (0.00%)	1.00x
decode pub key	9 (1.6%)	9 (2%)	1.00x	18 (2%)	18 (2%)	1.00x
decode sign	12 (2%)	12 (2%)	1.00x	24 (2%)	24 (2%)	1.00x
hash mess to point	312 (55%)	311 (55%)	1.00x	595 (52%)	595 (52%)	1.00x
verify sign	231 (41%)	231 (41%)	1.00x	501 (44%)	501 (44%)	1.00x
Expand Private Key	**512-FPU**	**512-EMU**	**Vs.**	**1024-FPU**	**1024-EMU**	**Vs.**
get priv deg	<1 (0.00%)	<1 (0.00%)	1.00x	<1 (0.00%)	<1 (0.00%)	1.00x
decode priv	494 (35%)	494 (4%)	1.00x	1,040 (34%)	1,040 (34%)	1.00x
expand priv key	905 (65%)	11,281 (96%)	**12.5x**	2,018 (66%)	25,010 (96%)	**12.3x**

V. Constant-Time Validation of Falcon's Floating-Point Operations

This section presents the constant runtime analysis of Falcon on the ARM Cortex M7. Technical manuals for ARM development boards often report cycle counts for FPU instructions[5], however ARM does not appear to make this information public for the Cortex M7 core.

We are specifically interested in Falcon's use of double precision floating points and how it exploits the devices' 64-bit floating point unit (FPU). This has not been investigated before since the primary evaluation target used for post-quantum schemes, the ARM Cortex M4, only has a 32-bit FPU, which is not sufficient for the 53-bit floating-point precision required by Falcon.

The double precision FPU on the ARM Cortex M7 is compliant with the IEEE-754 standard as thus supports the binary64 type. The IEEE-754 standard defines all aspects of floating-point numbers (i.e., their sign, exponent, and mantissa) so that hardware/software interoperability can be ensured. Thus, most if not all modern CPUs offer compliance with this standard within their dedicated FPUs used to speed-up floating-point operations.

[5] For example, see the ARM Cortex-M4 Technical Reference Manual https://developer.arm.com/documentation/ddi0439/b/BEHJADED

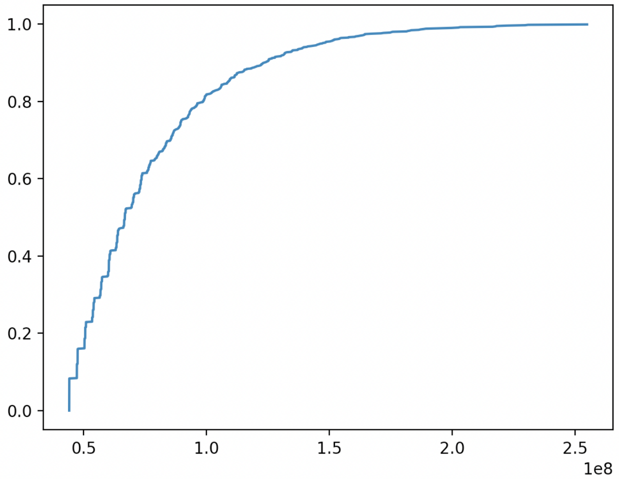

Figure 2: Falcon's key generation.

Table VIII: Profiling Dilithium on the ARM Cortex M7 using the STM32F767ZI NUCLEO-144 development board. All values reported are in KCycles.

Key Generation	**param2**	**param3**	**param5**
get randomness	13 (0.9%)	13 (0.5%)	13 (0.30%)
expand matrix	971 (68%)	1,826 (71%)	3,417 (78%)
sample vector	182 (13%)	317 (12%)	343 (8%)
matrix/vector mult	124 (9%)	190 (7%)	300 (7%)
add error	45 (0.34%)	7 (0.28%)	10 (0.23%)
expand/write pub key	16 (1%)	25 (1%)	33 (0.76%)
get h/comp priv key	125 (9%)	188 (7%)	247 (6%)
Signing	**param2**	**param3**	**param5**
compute crh	13 (0.39%)	13 (0.24%)	14 (0.17%)
exp mat/transf vecs	1,092 (32%)	1,993 (35%)	3,656 (47%)
sample y vector	1,001 (29%)	1,538 (27%)	1,688 (22%)
matrix/vector mult	516 (15%)	946 (17%)	1,178 (15%)
decomp w/ call RO	547 (16%)	710 (13%)	693 (9%)
compute z	137 (4%)	233 (4%)	269 (3%)
check cs2	62 (2%)	91 (2%)	123 (2%)
compute hint	70 (2%)	110 (2%)	149 (2%)
Verifying	**param2**	**param3**	**param5**
compute crh	124 (9%)	181 (8%)	235 (6%)
matrix/vector mult	1,174 (84%)	2,119 (88%)	3,859 (91%)
reconstruct w1	24 (2%)	28 (1%)	38 (0.90%)
call ro verify chall	78 (6%)	78 (3%)	100 (2%)

We investigate the timings on the device used in the previous sections, the STM32F767ZI NUCLEO-144 development board, and due to the issues found we extended this to three other STM32 development boards (the STM32H743ZI, STM32H723ZG, and STM32F769I-DISCO) in order to see if this issue affected other development boards. We found the same issues occured in all four development boards. We are aware of a similar experiment being run on the STM32H730[6]. We also further investigate timing issues on the Raspberry Pi 3, due to its use in evaluating the constant-time code of Falcon [26].

A. STM32 Development Boards

The issue discovered with the STM32 development boards was that the FPU operations were not fully constant time. We did not pursue ways to exploit this into an attack, but we felt this was worth reporting nonetheless. The code for testing this constant timeness is available on repository already provided.

For each floating-point instruction (e.g., `vmul.f64`), we wrote inline assembly of ten consecutive operations, given two random inputs, which we then averaged to find the required clock cycles. We used inline assembly to minimize the unwanted optimizations from the compiler, and clobbered registers where necessary. Using this approach minimizes the effect of surrounding instructions on the operations of interest, which for example would occur using C, and ensures that all execution is from cache. An example of this is shown in Listing 1 for the 64-bit floating point multiplication operation `vmul.f64`.

```
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");
```

Listing 1: Code snippet of the testing framework we used to test the constant timeness of the double precision FPU on the STM32 development boards.

The FPUs on the development boards typically provide two functions for each floating-point function; a 32-bit version (e.g., `vadd.f32`) and a 64-bit version (e.g., `vadd.f64`). Since we are concerned with Falcon which requires 53 bits of floating-point precision, we focus on the 64-bit (double-precision) floating-point functions. The IEEE 754 standard for floating-point binary representation is shown in Table IX for `float` and `double` types. The double-precision binary floating-point format (binary64) expresses floating point numbers using a 1-bit sign value in the most significant position, 11 bits for the exponent in positions 62-to-52, and 52 bits for the significand in positions 51-to-0.

We discovered variable timing behaviour in *all* double-precision floating-point functions on *all* the development boards we used in the experiments. We now focus on the double-precision floating-point addition (`vadd.f64`) function to illustrate and explain lower level timing irregularities.

[6] https://www.quinapalus.com/cm7cycles.html

Table IX: IEEE 754 standard format for single (32-bit) and double precision (64-bit).

Type/Precision	Sign	Exponent	Significand
`float` (32 bits)	31 (1 bit)	30:23 (8 bits)	22:0 (23 bits)
`double` (64 bits)	63 (1 bit)	62:52 (11 bits)	51:0 (52 bits)

The non-constant timeness was clearly observed when generating two random double-precision values for addition, with an average runtime of 16 clock cycles and standard deviation of 4.1. However, when we generated random values in the same range such they had the same exponents, the runtimes were constant and consistant at 10 clock cycles. Moreover, when we mixed randomness from two fixed exponent ranges we observed constant and consistant runtimes of 19 clock cycles.

B. Raspberry Pi 3

We also discovered a subtle issue with constant timeness on the Raspberry Pi 3, which itself has an ARM Cortex A53 core. This issue involves type casting, specifically, when casting a `double` to an `int64_t`, the operation rounds towards zero. There is no native instruction to do such a truncation on ARMv7. Thus instead, the compiler calls the runtime symbol `__fixdfi`, that is, `__aeabi_d2lz`. This may or may not be implemented in constant time. In LLVM it is not[7] and importantly it *leaks the sign*. This is the case for the Raspberry Pi 3 which they targeted in [26]. We reported this issue to the Falcon team and moreover proposed a constant time fix, which we show in Listing 2.

```
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;

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

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

Listing 2: The proposed fix for casting a `double` to an `int64_t` in LLVM.

VI. Results and Discussions

In Section III, we observe from the benchmarking of Dilithium in Table I that all procedures show a slight improvement, but not many of significance in comparison to those reported by pqm4. The performance improvements seen range from 1.09-1.19x which essential accounts for the slightly better performance of the Cortex M7 vs the Cortex M4 in general.

However, from the benchmarking of Falcon in Table II we observe that:

- Key generation does not drastically benefit from the FPU, showing a 1.66-1.76x improvement in compared to emulated floating points. We also see similar results compared to the Cortex M4, with improvements between 2.21-2.56x.
- Sign dynamic has a significant improvement using the FPU; showing an increase between 6.16-6.31x between the emulated code and between 8.16-8.31x compared to the Cortex M4.
- Sign tree also has a significant improvement using the FPU; showing an increase between 4.69-4.92x between the emulated code and 6.23x compared to the Cortex M4 for Falcon-512 parameters. As already stated, Falcon-1024 sign tree cannot fit on the Cortex M4, but has been implemented in this research on the Cortex M7.
- Expanding the private key also has a significant improvement using the FPU; showing an increase between 8.37-8.49x between the emulated code.
- Verify shows little to know changes by using the FPU, due to it not requiring floating-point operations, and the slight decrease is probably due to the larger instruction pipeline on the M7.

In Section IV, we provide profiling results of the two signature schemes, which can point to areas in which these schemes could be optimised in the future. The profiling results of Dilithium in Table VIII perhaps offer little novel insights into the bottlenecks of its implementation on the Cortex M7. Dilithium has a much simpler implementation complexity in comparison to Falcon and this can be observed by the much more compact table of results. However, we can observe the elegance of its design and performance when comparing the results across parameter sets; seeing that some values change little, and some increase proportional to the added computations required by the small change in each parameter set, afforded by fixing the polynomial ring and modulus.

In Section III-A we provided stack and RAM usage for Dilithium and Falcon. The most notable results are for Falcon which has a small increase (at most, 88 Bytes) in stack usage when the FPU is used.

We observe from the profiling of Falcon in Table VII the following. The FPU improves upon emulating floating-point operations in key generation by an order of magnitude, specifically in the following operations.

- Converting a small vector to floating point (`poly_small_to_fp`) improves by 7.5-7.65x, multiplying polynomials by a constant (`poly_mulconst`) and an adjoint (`poly_mul_autoadj_fft`) improves by 5.13-5.36x and 5.37-6.08x, respectively.

[7] see for example https://github.com/llvm-mirror/compiler-rt/blob/69445f095c22aac2388f939bedebf224a6efcdaf/lib/builtins/fixdfdi.c#L18

- Polynomial inversion to FFT format (`poly_invnorm2_fft`) saves between 11-13.2x.
- The normalisation step alongside `FPR` addition saves between 12.1-13.1x.
- FFT and iFFT operations improve by 15.4-17.2x, making this the biggest improvement of all operations in Falcon.

The FPU improves upon emulating floating-point operations in sign dynamic by an order of magnitude, specifically in the following operations.

- ffSampling improves by 5.75-5.88x, get lattice point and computing the Gram matrix (G) improves by 7.88-8.05x and 7.72-9.37x, respectively.
- Applying the lattice basis, recomputing the matrix basis, and converting the basis to FFT format save 13.3-14x, 15-15.1x, and 15.9-16.1x, respectively.
- Similar savings are noted for signing tree for applying the lattice basis, applying ffSampling, and getting the lattice point.to the tiny vector.
- Expanding the private key saves between 12.3-12.5x.

The FPU does not have any affect on Falcon's verification operation, this is essentially because it does not require floating-point operations and is a relatively computationally light procedure.

In Section V, we find constant time issues with Falcon on four different STM32 development boards using the ARM Cortex M7 and the Raspberry Pi 3. The issues we found on the STM32 development boards were where the devices' dedicated floating-point unit was used (which can significantly speed-up Falcon), specifically the double-precision functions, where all were shown to be non constant time. Analysing the double-preicision addition, we discovered the size of the significand influenced the runtime of this function.

We further investigated constant timeness on the Raspberry Pi 3, which uses the ARM Cortex A53, where we also found timing issues when casting from a `double` to an `int64_t`, and when implemented in LLVM, it is not constant time and leaks the sign of the value.

We reported these issues and our proposed fix to the Falcon team but we did not investigate how to exploit this for a timing attack.

Overall, this research shows that when implementing Falcon the platform and/or situation it is used in should play a major consideration. At the very least, the processor should be checked for constant timesness *if* the FPU is being used. A recent Cloudflare blog[8] took note of our results and is currently only considering uses for Falcon in an offline manner, as they "feel it's too early to deploy Falcon where the timing of signature minting can be measured".

[8] https://blog.cloudflare.com/nist-post-quantum-surprise/

References

[1] NIST, *Post-quantum cryptography*, https://csrc.nist.gov/projects/post-quantum-cryptography, 2015 (cit. on p. 1).

[2] ——, *Submission requirements and evaluation criteria for the post-quantum cryptography standardization process*, 2016 (cit. on p. 1).

[3] G. Alagic, G. Alagic, J. Alperin-Sheriff, *et al.*, *Status report on the first round of the NIST post-quantum cryptography standardization process*. 2019 (cit. on p. 1).

[4] G. Alagic, J. Alperin-Sheriff, D. Apon, *et al.*, "Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process," *NIST, Tech. Rep., July*, 2020 (cit. on p. 1).

[5] *PQClean: clean, portable, tested implementations of post-quantum cryptography*, https://github.com/PQClean/PQClean (cit. on p. 1).

[6] *SUPERCOP: system for unified performance evaluation related to cryptographic operations and primitives*, https://bench.cr.yp.to/supercop.html (cit. on p. 1).

[7] *liboqs: C library for prototyping and experimenting with quantum-resistant cryptography*, https://github.com/open-quantum-safe/liboqs (cit. on p. 1).

[8] *PQM4: Post-quantum crypto library for the ARM Cortex-M4*, https://github.com/mupq/pqm4 (cit. on p. 1).

[9] J. Richter-Brockmann and T. Güneysu, *Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices*, Cryptology ePrint Archive, Report 2020/897, 2020 (cit. on p. 1).

[10] J. Howe, T. Oder, M. Krausz, and T. Güneysu, "Standard lattice-based key encapsulation on embedded devices," *IACR TCHES*, no. 3, 2018 (cit. on p. 1).

[11] J. Howe, M. Martinoli, E. Oswald, and F. Regazzoni, "Exploring parallelism to improve the performance of FrodoKEM in hardware," *Journal of Cryptographic Engineering*, no. 4, 2021 (cit. on p. 1).

[12] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR TCHES*, no. 4, 2019 (cit. on p. 1).

[13] Y. Xing and S. Li, "A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021 (cit. on p. 1).

[14] S. Ricci, L. Malina, P. Jedlicka, *et al.*, *Implementing crystals-dilithium signature scheme on fpgas*, Cryptology ePrint Archive, Report 2021/108, 2021 (cit. on p. 1).

[15] A. Marotzke, "A constant time full hardware implementation of streamlined ntru prime," in *International Conference on Smart Card Research and Advanced Applications*, Springer, 2020 (cit. on p. 1).

[16] D. Kales, S. Ramacher, C. Rechberger, R. Walch, and M. Werner, "Efficient FPGA implementations of LowMC and Picnic," in *CT-RSA*, 2020 (cit. on p. 1).

[17] S. S. Roy and A. Basso, "High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware," *IACR TCHES*, no. 4, 2020 (cit. on p. 1).

[18] P. M. C. Massolino, P. Longa, J. Renes, and L. Batina, "A compact and scalable hardware/software co-design of SIKE," *IACR TCHES*, no. 2, 2020 (cit. on p. 1).

[19] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, *Efficient and fast hardware architectures for sike round 2 on fpga*, Cryptology ePrint Archive, Report 2020/611, 2020 (cit. on p. 1).

[20] B. Koziel, A. Ackie, R. El Khatib, R. Azarderakhsh, and M. M. Kermani, "Sike'd up: Fast hardware architectures for supersingular isogeny key encapsulation," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2020 (cit. on p. 1).

[21] G. Alagic, D. Apon, D. Cooper, *et al.*, “Status report on the third round of the nist post-quantum cryptography standardization process,” National Institute of Standards and Technology Gaithersburg, MD, Tech. Rep., 2022 (cit. on p. 1).

[22] P. Schwabe, R. Avanzi, J. Bos, *et al.*, “CRYSTALS-KYBER,” National Institute of Standards and Technology, Tech. Rep., 2019 (cit. on p. 1).

[23] V. Lyubashevsky, L. Ducas, E. Kiltz, *et al.*, “CRYSTALS-DILITHIUM,” National Institute of Standards and Technology, Tech. Rep., 2020 (cit. on pp. 1, 2).

[24] T. Prest, P.-A. Fouque, J. Hoffstein, *et al.*, “FALCON,” National Institute of Standards and Technology, Tech. Rep., 2020 (cit. on pp. 1, 2).

[25] A. Hulsing, D. J. Bernstein, C. Dobraunig, *et al.*, “SPHINCS+,” National Institute of Standards and Technology, Tech. Rep., 2019 (cit. on p. 1).

[26] T. Pornin, *New efficient, constant-time implementations of Falcon*, Cryptology ePrint Archive, Report 2019/893, 2019 (cit. on pp. 1, 6, 7).

[27] ARM, *Arm cortex-m7 processor: Technical reference manual*, Revision r1p2, 2018 (cit. on p. 1).

[28] D. O. C. Greconici, M. J. Kannwischer, and D. Sprenkels, *Compact dilithium implementations on cortex-M3 and cortex-M4*, Cryptology ePrint Archive, Report 2020/1278, 2020 (cit. on pp. 2, 3).

[29] J. Howe, T. Prest, T. Ricosset, and M. Rossi, “Isochronous gaussian sampling: From inception to implementation,” in *Post-Quantum Cryptography - 11th International Conference, PQCrypto*, 2020 (cit. on p. 3).