Fast Falcon Signature Generation and Verification Using ARMv8 NEON Instructions

Duc Tri Nguyen and Kris Gaj

George Mason University, Fairfax, VA, 22030, USA
{dnguye69,kgaj}@gmu.edu

Abstract. We present our speed records for Falcon signature generation and verification on ARMv8-A architecture. Our implementations are benchmarked on Apple M1 'Firestorm' and Raspberry Pi 4 Cortex-A72 chips. Compared with lattice-based CRYSTALS-Dilithium, our optimized signature generation is 2× slower, but signature verification is 3–3.9× faster than the state-of-the-art CRYSTALS-Dilithium implementation on the same platforms. Compared with stateful and stateless hash-based digital signatures, XMSS and SPHINCS$^+$, our optimized Falcon implementation has 10–950× faster signature generation and 23–31× faster signature verification. Faster signature verification may be particularly useful for the client side on constrained devices. Our Falcon implementation outperforms the previous work in both Sign and Verify operations. We achieve improvement in Falcon signature generation by supporting all possible parameters N of FFT-related functions and applying our compressed twiddle-factor table to reduce memory usage. We also demonstrate that the recently proposed signature scheme Hawk, sharing functionality with Falcon, offers 17% smaller signature sizes, 3.3× faster signature generation, and 1.6–1.9× slower signature verification when implemented on the same ARMv8 processors as Falcon.

Keywords: Number Theoretic Transform · Fast Fourier Transform · Falcon · lattice-based cryptography · Post-Quantum Cryptography · ARMv8 · NEON

1 Introduction

When large quantum computers arrive, Shor's algorithm [Sho94] will break almost all currently deployed public-key cryptography in polynomial time [CJL$^+$16] due to its capability to obliterate two cryptographic bastions: the integer factorization and discrete logarithm problems. While there is no known quantum computer capable of running Shor's algorithm with parameters required to break current public-key standards, the process of selecting, standardizing, and deploying new cryptographic algorithms has always been taking years, if not decades.

In 2016, NIST announced the Post-Quantum Cryptography (PQC) standardization process aimed at developing new public-key standards resistant to quantum computers. In July 2022, NIST announced the choice of three digital signature algorithms [AAC$^+$22]: CRYSTALS-Dilithium [BDK$^+$21], Falcon [FHK$^+$20] and SPHINCS$^+$ [BHK$^+$19] for a likely standardization within the next two years. Additionally, NIST has already standardized two stateful signature schemes XMSS [HBG$^+$18] and LMS [MCF19].

Compared to Elliptic Curve Cryptography and RSA, PQC digital signatures have imposed additional implementation constraints, such as bigger key and signature sizes (and thus increased bandwidth), higher memory usage, support for floating-point operations, etc. In many common applications, such as the distribution of software updates and the use of digital certificates, a signature is generated once by the server but verified over and over again by clients forming the network.

In this paper, we examine the PQC digital signatures' speed on ARMv8 platforms. ARMv8 (a.k.a. ARMv8-A) supports two instruction set architectures AArch64 (a.k.a. arm64) and AArch32 (a.k.a. armeabi). The associated instruction sets are referred to as A64 and A32, respectively. AArch32 and A32 are compatible with an older version of ARM called ARMv7-A. AArch64 and A64 support operations on 64-bit operands.

NEON is an alternative name for Advanced Single Instruction Multiple Data (ASIMD) extension, mandatory since ARMv7. NEON includes additional instructions that can perform arithmetic operations in parallel on multiple data streams. It also provides a developer with 32 128-bit vector registers. Each register can store two 64-bit, four 32-bit, eight 16-bit, or sixteen 8-bit integer data elements. NEON instructions can perform the same arithmetic operation simultaneously on the corresponding elements of two 128-bit registers and store the results in the respective fields of a third register. Thus, an ideal speed-up vs. traditional single-instruction single-data (SISD) ARM instructions varies between 2 (for 64-bit operands) and 16 (for 8-bit operands).

In today's market, there is a wide range of ARMv8 processors supporting NEON. They are developed by manufacturers such as Apple Inc., ARM Holdings, Cavium, Fujitsu, Nvidia, Marvell, Qualcomm, and Samsung. They power the majority of smartphones and tablets available on the market. The ARM Cortex-A72 is a core implementing the ARMv8-A 64-bit instruction set. It was designed by ARM Holdings' Austin design center. This core is used in Raspberry Pi 4 - a small single-board computer developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom. The corresponding SoC is called BCM2711. This SoC contains four Cortex-A72 cores. Apple M1 is an ARMv8-based system on a chip (SoC) designed by Apple Inc. for multiple Apple devices, such as MacBook Air, MacBook Pro, Mac Mini, iMac, and iPad Pro. The M1 has four high-performance 'Firestorm' and four energy-efficient 'Icestorm' cores supporting an extension of ARMv8 called ARMv8.4-A. Each of them supports NEON. Starting from version Armv8.3-A, and thus included in version Armv8.4-A, there is NEON support for operations on complex numbers.

In this work, we developed an optimized implementation of Falcon targeting ARMv8 cores. We then reused a significant portion of our Falcon code to implement Hawk – a new lattice-based signature scheme proposed in Sep. 2022 [DPPvW22]. Although this scheme is not a candidate in the NIST PQC standardization process yet, it may be potentially still submitted for consideration in response to the new NIST call, with the deadline in June 2023.

We then benchmarked our implementation and existing implementations of Falcon, Hawk, CRYSTALS-Dilithium, SPHINCS+, and XMSS using the 'Firestorm' core of Apple M1 (being a part of MacBook Air) and the Cortex-A72 core (being a part of Raspberry Pi 4), as these platforms are widely available for benchmarking. However, we expect that similar rankings of candidates can be achieved using other ARMv8 cores (a.k.a. microarchitectures of ARMv8).

Contributions. In this paper, we overcome the high complexity of the Falcon implementation and present a speed record for its **S**ignature generation and **V**erification on two different ARMv8 processors.

In a signature generation, we constructed vectorized scalable FFT implementation that can be applied to any FFT level greater than five. We compressed the twiddle factor table in our FFT implementation, inspired by the complex *conjugate* root of FFT. In particular, we reduced the size of this table from *16 Kilobytes* in the reference implementation down to *4 Kilobytes* in our new `ref` and `neon` implementations. The modified FFT implementation with $4\times$ smaller twiddle factor table is not specific to any processor. Thus, it can be used on any platform, including constrained devices with limited storage or memory.

In the **V**erify operation, we applied the best-known Number Theoretic Transform (NTT) implementation techniques to speed up its operation for Falcon-specific parameters. Additionally, we present the exhaustive search bound analysis applied to twiddle factors per NTT level aimed at minimizing the number of Barrett reductions in Forward and Inverse NTT.

We comprehensively compare three stateless and one stateful digital signature schemes selected by NIST for standardization – Falcon, CRYSTALS-Dilithium, SPHINCS$^+$, and XMSS – and one recently-proposed lattice-based scheme Hawk. We rank them according to signature size, public-key size, and Sign and Verify operations' performance using the best implementations available to date.

2 Previous Work

The paper by Streit et al. [SDS18] was the first work about a NEON-based ARMv8 implementation of the lattice-based public-key encryption scheme New Hope Simple. Other works implement Kyber [ZZH+21], SIKE [JAMK+19], and Frodo [KJK+21] on ARMv8. The most recent works on the lattice-based finalists NTRU, Saber, and CRYSTALS-Kyber are reported by Nguyen et al. [NG21a,NG21b]. The paper improved polynomial multiplication and compared the performance of vectorized Toom-Cook and NTT implementations. Notably, the work by Becker et al. [BHK+21] showed a vectorized NEON NTT implementation superior to Toom-Cook, introduced fast Barrett multiplication, and special use of multiply-return-high-only `sq[r]dmulh` instructions. The SIMD implementation of Falcon was reported by Pornin [Por19] and Kim et al. [KSS22]. On the application side, Falcon is the only viable option in hybrid, partial, and pure PQC V2V design described in the work of Bindel et al. [BMTR22]

In the area of low-power implementations, most previous works targeted the ARM Cortex-M4 [KRSS19]. In particular, Botros et al. [BKS19], and Alkim et al. [AABCG20] developed Cortex-M4 implementations of Kyber. Karmakar et al. [KBMSRV18] reported results for Saber. Chung et al. [CHK+21] on Saber and NTRU, and later work by Becker et al. [BMK+22] improved Saber by a large margin on Cortex-M4/M55. The latest work by Abdulrahman et al. [AHKS22] improved Kyber and Dilithium performance on Cortex-M4.

The most comprehensive Fast Fourier Transform (FFT) work is by Becoulet et al. [BV21][1]. FFTW and FFTS by Frigo et al. The publications [FJ12] and Blake [BWC13] describe the SIMD FFT implementations. Impressive results have been reported for AutoFFT on both ARM and x86 CPUs by Li et al. [LJZ+20]. However, we were unable to locate the corresponding source code.

3 Background

Table 1 summarizes values of parameters n and q for various NIST security levels. n is a parameter in the cyclotomic polynomial $\phi = (x^n + 1)$, and q is a prime defining a ring $\mathbb{Z}_q[x]/(\phi)$. The sizes of the public key and signature in bytes (**B**) are denoted with $|pk|$ and $|sig|$. The signature ratio $|sig|$ ratio is the result of dividing the signature size of other schemes by the signature size of FALCON512, DILITHIUM3, and FALCON1024.

3.1 Falcon

Falcon is a lattice-based signature scheme utilizing the 'hash-and-sign' paradigm. The security of Falcon is based on the hardness of the Short Integer Solution problem over

[1] https://github.com/diaxen/fft-garden

Table 1: Parameter sets, key sizes, and signature sizes for FALCON, HAWK, DILITHIUM, XMSS, and SPHINCS$^+$ in comparison with FALCON512, DILITHIUM3, FALCON1024 according to three security levels

	NIST level	n	q	|pk|	|sig|	|pk + sig|	|sig| ratio
FALCON512	I	512	12,289	897	652	1,549	1.00
HAWK512			65,537	1,006	542	1,548	0.83
DILITHIUM2	II	256	8,380,417	1,312	2,420	3,732	3.71
XMSS^{16}-SHA256		-	-	64	2,692	2,756	4.12
SPHINCS$^+$128s	I	-	-	32	7,856	7,888	12.05
SPHINCS$^+$128f		-	-	32	17,088	17,120	26.21
DILITHIUM3	III	256	8,380,417	1,952	3,293	5,245	1.00
SPHINCS$^+$192s		-	-	48	16,224	16,272	4.93
SPHINCS$^+$192f		-	-	48	35,664	35,712	10.83
FALCON1024	V	1024	12,289	1,793	1,261	3,054	1.00
HAWK1024			65,537	2,329	1,195	3,524	0.95
DILITHIUM5		256	8,380,417	2,592	4,595	7,187	3.64
SPHINCS$^+$256s		-	-	64	29,792	29,856	23.62
SPHINCS$^+$256f		-	-	64	49,856	49,920	39.53

NTRU lattices, and the security proofs are given in the random oracle model with tight reduction. Falcon is difficult to implement, requiring tree data structures, extensive floating-point operations, and random sampling from several discrete Gaussian distributions. The upsides of Falcon are its small public keys and signatures as compared to Dilithium. As shown in Table 1, the signature size of Falcon at the highest NIST security level is still smaller than that of the lowest security level of Dilithium, XMSS, and SPHINCS$^+$. Key generation in Falcon is expensive. However, a key can be generated once and reused later.

The signature generation (Algorithm 17 in Appendix A) of Falcon first computes hash value c from message m and salt r. Then, it uses (f, g, F, G) from the secret key components to compute two short values s_1, s_2 such that $s_1 + s_2 h = c \bmod (\phi, q)$. Falcon relies extensively on floating point computation during signature generation, used in Fast Fourier Transform (FFT) over the ring $\mathbb{Q}[x]/(\phi)$, and Gaussian and Fast Fourier Sampling (`ffSampling`) for Falcon tree T.

The signature verification (Algorithm 18 in Appendix A) checks if two short values (s_1, s_2) are in acceptance bound $\lfloor \beta^2 \rfloor$ using the knowledge from public key pk, and signature (r, s). If the condition at line 5 is satisfied, then the signature is valid; otherwise it is rejected. As opposed to signature generation, Falcon Verify operates only over integers.

Falcon only supports NIST security levels 1 and 5. A more detailed description of the underlying operations can be found in the Falcon specification [FHK$^+$20].

3.2 Dilithium

Dilithium is a member of the Cryptographic Suite for Algebraic Lattices (CRYSTALS) along with the key encapsulation mechanism (KEM) Kyber. The core operations of Dilithium are the arithmetic of polynomial matrices and vectors. Unlike 'hash-and-sign' used in Falcon, Dilithium applied Fiat-Shamir with Aborts [DFG$^+$,Lyu09] style signature scheme and bases its security upon the Module Learning with Errors (M-LWE) and Module Short Integer Solution (M-SIS) problems.

Compared with Falcon, Dilithium only operate over the integer ring $\mathbb{Z}_q[x]/(\phi)$ with

$\phi = (x^n + 1)$. Thus, it is easier to deploy in environments lacking floating-point units. Dilithium supports three NIST security levels: 2, 3, and 5, and its parameters are shown in Table 1. More details can be found in the Dilithium specification [BDK+21].

3.3 XMSS

XMSS [HBG+18] (eXtended Merkle Signature Scheme) is a stateful hash-based signature scheme based on Winternitz One-Time Signature Plus (WOTS+) [Hül13]. XMSS requires state tracking because the private key is updated every time a signature is generated. Hence, the key management of XMSS is considered difficult. Consequently, XMSS should only be used in highly controlled environments [CAD+20]. The advantages of XMSS over SPHINCS$^+$ are smaller signature sizes and better performance. XMSS is a single-tree scheme, with a multi-tree variant XMSS^{MT} also included in the specification. The security of XMSS relies on the security of collision search of an underlying hashing algorithm.

Single-tree XMSS has faster signature generation and verification than the multi-tree XMSS^{MT} and comes with three tree heights: $h = [10, 16, 20]$, which can produce up to 2^h signatures. In this paper, we select a single-tree variant of XMSS with a reasonable number of signatures, 2^{16}, and choose SHA256 as underlying hash functions. This variant is denoted as XMSS^{16}-SHA256 in Table 1.

3.4 SPHINCS$^+$

SPHINCS$^+$ [BHK+19] is a stateless hash-based signature scheme that avoids the complexities of state management associated with using stateful hash-based signatures. SPHINCS$^+$ security also relies on hash algorithms. The algorithm is considered a conservative choice, preventing any future attacks on lattice-based signatures. SPHINCS$^+$ provides *'simple'* and *'robust'* construction. The *'robust'* construction affects the security proof and runtime. In addition, small ('*s*') and fast parameters ('*f*') influence execution time. These parameter set variants are over 128, 192, and 256 quantum security bits.

Based on the performance provided in the specification of SPHINCS$^+$, we select the *'simple'* construction, and both '*s*' and '*f*' parameters for NIST security levels 1, 3, and 5, as shown in Table 1. Unlike XMSS, we select SHAKE as the underlying hash function.

3.5 Hawk

Hawk is a recent signature algorithm proposed by Ducas et al. [DPPvW22] based on the Lattice Isomorphism Problem (LIP). Hawk avoids the complexities of the floating-point discrete Gaussian sampling, which is a bottleneck in our optimized Falcon implementation. Hawk chooses to sample in a simple lattice $\mathbb{Z}^n$ [BGPS21, DvW22] with a hidden rotation.

An AVX2 implementation of HAWK1024 is faster than the equivalent implementation of FALCON1024 by 3.9× and 2.2×. With our optimized `neon` implementation of Falcon, we decided to port our optimized Falcon code to Hawk and investigated if such performance gaps between Hawk and Falcon still hold. In this work, we select HAWK512, HAWK1024 at NIST security levels 1 and 5.

4 Number Theoretic Transform Implementation

The Number Theoretic Transform (NTT) is a transformation used as a basis for a polynomial multiplication algorithm with the time complexity of $O(n \log n)$ [CT65]. In Falcon, the NTT algorithm is used for polynomial multiplication over the ring $\mathbf{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, where degree $n = [512, 1024]$ and $q = 12289 = 2^{13} + 2^{12} + 1$ with $q = 1 \bmod 2n$.

Complete NTT is similar to traditional FFT (Fast Fourier Transform) but uses the root of unity in the discrete field rather than in a set of real numbers. NTT and NTT^{-1} are forward and inverse operations, where $NTT^{-1}(NTT(f)) = f$ for all $f \in \mathbf{R}_q$.

$NTT(A) * NTT(B)$ denotes pointwise multiplication. Polynomial multiplication using NTT is shown in Equation 1.

$$C(x) = A(x) \times B(x) = NTT^{-1}(NTT(A) * NTT(B)) \tag{1}$$

4.1 Barrett multiplication

Barrett multiplication and reduction are summarized in Appendix B. In our Falcon implementation, Barrett multiplication is used extensively when one factor is known. As shown in Algorithm 1, b must be a known constant, and b' is derived from b. In fact, b and b' in NTT are from the precomputed table ω_i and ω'_i. An explanation of Barrett multiplication is shown in Equation 4 in Appendix B.

Algorithm 1: Signed Barrett multiplication with a known constant [BHK+21]

```
  Input: Any $|a| < R = 2^w$, constant $|b| < q$ and $b' = [(b \cdot R/q)/2]$
  Output: $c =$ barrett_mul$(a, b, b') = a * b \bmod q$, and $|c| < \frac{3q}{2} < \frac{R}{2}$
1 $t \leftarrow$ sqrdmulh$(a, b')$        ▷ hi(round($(2 \cdot a \cdot b')$))
2 $c \leftarrow$ mul$(a, b)$               ▷ lo$(a \cdot b)$
3 $c \leftarrow$ mls$(c, t, q)$            ▷ lo$(c - t \cdot q)$
```

4.2 Montgomery multiplication

The basic idea of Montgomery multiplication is summarized in Appendix B. Its use in our implementation is explained below.

First, Falcon Verify computes only one polynomial multiplication (as in line 4 of Algorithm 18). Two polynomials (s_2, h) are converted to NTT domain. Then, we perform pointwise multiplication between two polynomials in NTT domain. To efficiently compute modular reduction for two unknown factors during pointwise multiplication, the conventional way is to convert one polynomial to the Montgomery domain and perform Montgomery multiplication. Eventually, the multiply with a constant factor n^{-1} is applied at the end of Inverse NTT. We apply small tweak by embedding n^{-1} into Montgomery conversion during pointwise multiplication ($a_i b_i n^{-1}$ instead of $a_i b_i$ for $i \in [0, \dots, n-1]$) to avoid multiplications with n^{-1} in Inverse NTT.

The Montgomery n^{-1} conversion uses Barrett multiplication with a known factor $a_{mont} =$ `barrett_mul`(a, b, b'), with $b = R \cdot n^{-1} \bmod q$. Furthermore, it is beneficial at the instruction level to embed n^{-1} when n is a power of 2 in Montgomery conversion. In particular, when $(R, n) = (2^{16}, 2^{10})$ then $b = R \cdot n^{-1} = 2^{16} \cdot 2^{-10} = 2^6 \bmod q$. Hence, line 2 of Algorithm 1 can be replaced by a shift left `shl` instruction.

Secondly, we apply Algorithm 2 for pointwise multiplication. Another variant of Montgomery multiplication *via rounding* using only 4 instructions is considered, but it only works if one factor is odd (Section 3 in Becker et al. [BHK+21]). Thus, Montgomery multiplication *via doubling* is selected since it does not assume the parity of coefficients.

Algorithm 2: Signed Montgomery multiplication *via doubling* [BHK+21]

```
Input: $|a| < R$, and $|bR| < R$, $R = 2^w$ and $w = [16, 32]$
Output: $c = \texttt{mont\_mul}(a, bR) = a * (bR) * (R^{-1}) = a * b \bmod q$ and $-q \le c < q$
1 $t \leftarrow \texttt{mul}(b, q^{-1})$        ▷ $\texttt{lo}(b \cdot q^{-1})$
2 $c \leftarrow \texttt{sqdmulh}(a, b)$         ▷ $\texttt{hi}(2 \cdot a \cdot b)$
3 $t \leftarrow \texttt{mul}(a, t)$             ▷ $\texttt{lo}(a \cdot t)$
4 $t \leftarrow \texttt{sqdmulh}(t, q)$         ▷ $\texttt{hi}(2 \cdot t \cdot q)$
5 $c \leftarrow \texttt{shsub}(c, t)$           ▷ $(c - t)/2$
```

4.3 Minimizing the number of Barrett reductions

The Algorithm 3, derived from Equation 3 in Appendix B, consists of two multiplications and one rounding right-shift instruction. By utilizing signed multiply-return-high-only instruction, the total number of instructions is reduced from 6 (in general Barrett reduction [NG21a,NG21b]) to 3 (in Becker et al. [BHK+21]). However, this can only be applied in the NTT centered around 0 due to the signed arithmetic of `sqdmulh` instruction.

In Barrett multiplication (Algorithm 1), the bound of output c is in $-\frac{3q}{2} \le c < \frac{3q}{2}$. Details of the proof can be found in Becker et al. [BHK+21]. Given that $q = 12289, R = 2^{16}$, and center around 0, the maximum bound of signed arithmetic center around 0 is $2.6q \approx \frac{R}{2q}$ instead of $5.3q \approx \frac{R}{q}$ in unsigned arithmetic.

During Forward and Inverse NTT, we carefully control the bound of each coefficient by applying our strict Barrett multiplication bound analysis. The naive $2.6q$ bound assumption will lead to performing Barrett reduction after every NTT level in CT butterflies (Algorithm 4). To minimize the number of Barrett reductions, we validate the range of $c = \texttt{barrett\_mul}(b, \omega, \omega')$ for all unknown values of $|b| < \frac{R}{2}$ and $\omega \in \omega_i$ and $\omega' \in \omega'_i$ table according to each NTT level by exhaustive search. The bound output c of `barrett_mul` is increasing if $|b| < q$ and decreasing if $|b| \ge q$. For example, if $\frac{b}{q} \approx (\underline{0.5}, 1.0, 2.0, 2.5)$ then after Barrett multiplication, the obtained bounds are $\frac{c}{q} \approx (\underline{0.69}, 0.87, 1.25, 1.44)$.

As a result, we were able to minimize the number of reduction points in the Forward and Inverse NTT from after every one NTT level to every two NTT levels. In our case, an exhaustive search works in an acceptable time for the 16-bit space. However, it takes much longer for the 32- and 64-bit spaces. A formal, strict bound analysis instead of an exhaustive search approach is considered as future work.

Algorithm 3: Signed Barrett reduction [BHK+21] for prime $q = 12289$

```
Input: Any $|a| < R = 2^w$, constants $(q = 12289, w = 16, v = 5461, i = 11)$
Output: $c = \texttt{barrett\_mod}(a, q) \equiv a \bmod q$, and $-\frac{q}{2} \le c < \frac{q}{2}$
1 $t \leftarrow \texttt{sqdmulh}(a, v)$         ▷ $\texttt{hi}(2 \cdot a \cdot v)$
2 $t \leftarrow \texttt{srshr}(t, i)$           ▷ $\texttt{round}(t \gg i)$
3 $c \leftarrow \texttt{mls}(a, t, q)$          ▷ $\texttt{lo}(a - t \cdot q)$
```

4.4 Forward and Inverse NTT Implementation

Falcon uses NTT to compute polynomial multiplication in the Verify operation. To avoid a bit-reversal overhead, Cooley-Tukey (CT) butterflies (Algorithm 4) and Gentleman-Sande (GS) butterflies (Algorithm 5) are used for Forward and Inverse NTT, respectively.

Instead of vectorizing the original reference Falcon NTT implementation, we rewrite the NTT implementation to exploit cache temporal and spatial locality. Our NTT implementation is centered around 0 to use signed arithmetic instructions instead of the unsigned arithmetic approach used by default in Falcon. This choice of implementation significantly improved our work compared to Kim et al. [KSS22] due to special `sq[r]dmulh` instructions, which only work in signed arithmetic. We recommend utilizing multiply-return-high-only instruction for NTT implementation on any platform that supports it.

In Forward and Inverse NTT operations, `barrett_mul` is used extensively due to its compactness, thus yielding optimal performance, eliminating dependency chains by using only 3 instructions [BHK+21] rather than 9 instructions from Nguyen et al. [NG21a, NG21b]. At instruction level, based on Becker et al. [BHK+21] micro-architecture pipeline trick, we gather the addition and subtraction from multiple butterflies in a group and arrange multiple `barret_mul` together. Note that this behavior also appeared in modern compiler optimization. Since our implementation uses intrinsic instructions instead of assembly instructions, we confirmed that the output assembly code showed similar order of instructions as in intrinsic implementation. On the low-end Cortex-A72 ARMv8 processor, we achieved 10% performance improvement by grouping instructions compared with ungrouping multiply instructions. However, this improvement is negligible in the high-end Apple M1 processor.

In terms of storage, the Barrett multiplication requires twiddle factor table ω and an additional extra storage for precomputed table ω': $\omega'_i = [(\omega_i \cdot R/q)/2]$, where $\omega_i = \omega^i \bmod q$. We prepared the twiddle factor tables ω_i and ω'_i so that every read from such tables is in the forward direction, and each entry is loaded only once during the entire operation.

Our Forward and Inverse NTT consist of two loops with the constant-stride (cache-friendly) load and store into memory and the permutation following steps in Nguyen et al. [NG21a, NG21b]. As shown in Table 2, seven NTT levels are combined into the first loop, and the remaining two (resp. three) NTT levels are in the second loop for $n = 512$ (resp. 1024). Each coefficient is loaded and stored once in each loop. With two loops, our implementation can reach up to $n = 2048, R = 2^{16}$ or $n = 1024, R = 2^{32}$ with a minimal number of load and store instructions.

Algorithm 4: CT Butterflies

```
Input: $a, b, \omega, \omega' = [(\omega \cdot R/q)/2]$
Output: $(a, b) = (a + \omega b, a - \omega b)$
1 $t \leftarrow$ barrett_mul$(b, \omega, \omega')$
2 $b \leftarrow a - t$
3 $a \leftarrow a + t$
```

Algorithm 5: GS Butterflies

```
Input: $a, b, \omega, \omega' = [(\omega \cdot R/q)/2]$
Output: $(a, b) = (a + b, (a - b)\omega)$
1 $t \leftarrow a - b$
2 $a \leftarrow a + b$
3 $b \leftarrow$ barrett_mul$(t, \omega, \omega')$
```

5 Fast Fourier Transform Implementation

The Fast Fourier Transform (FFT) is a fast algorithm that computes a Discrete Fourier Transform from the time domain to the frequency domain and vice versa.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \text{ with } k \in [0, N-1] \tag{2}$$

The discrete Fourier transform in Equation 2 has time complexity of $O(n^2)$. FFT improves the transformation with the time complexity of $O(n \log n)$ [CT65].

The advantage of FFT over NTT for polynomial multiplication is that the root of unity $e^{i2\pi/N}$ always exists for arbitrary N. Additionally, FFT suffers precision loss caused by rounding in the floating-point-number computations, while polynomial multiplication using NTT guarantees correctness due to NTT operating in the integer domain.

Complex number arithmetic Let $a, b = (a_{re}, a_{im}), (b_{re}, b_{im})$ then $a \pm b = (a_{re} \pm b_{re}, a_{im} \pm b_{im})$ and $a * b = (a_{re}b_{re} - a_{im}b_{im}, a_{re}b_{im} + a_{im}b_{re})$. A counterclockwise rotation by 90 and 270 degree of a are $(ia) = (-a_{im}, a_{re})$ and $(-ia) = (a_{im}, -a_{re})$, respectively. Conjugate of a complex number $a = (a_{re}, a_{im})$ is $\hat{a} = conjugate(a) = (a_{re}, -a_{im})$.

5.1 Compressed twiddle factor table

In Falcon, each complex point utilizes 128 bits of storage. Reducing the required storage amount improves cache locality and minimizes memory requirements. Both improvements are especially important in constrained devices. When analyzing the twiddle factor table, we realized that the real and imaginary parts of complex points are repeated multiple times by the complex number negation, conjugation, and rotation. For example, complex roots of $x^8 + 1$ can be derived from a single complex root $a = (a_{re}, a_{im})$ to $[a, -a, -ia, ia]$ as the first half of the roots, and $[\hat{a}, -\hat{a}, -i\hat{a}, i\hat{a}]$ as the second half of the roots. It is notable that the second half is the *complex conjugate* of the first half, where $\hat{a} = conjugate(a) = (a_{re}, -a_{im})$. As a result, we only need to store $a = (a_{re}, a_{im})$ and use the add and subtract instructions to perform negation, conjugation, and rotation.

In summary, we fold the twiddle factor table by a factor of 2 by applying the *complex conjugate* to derive the second half from the first half. Furthermore, we use addition, subtraction, and rotation to derive the variants of complex roots within the first half, thus saving another factor of 2, as shown in Appendix D. However, the FFT implementation in reference implementation no longer works with our new twiddle factor table (`tw`). As a result, we rewrite our FFT implementation in `C` to adopt our newly compressed twiddle factor table.

In general, we can compress complex roots from n down to $\frac{n}{4}$. In particular, when $n = 512, 1024$, the default twiddle factor table size is $16n$ bytes. With compressed twiddle factors, we only need to store $128, 256$ complex roots, and the table size becomes $4n$ bytes. A special case in $x^4 + 1$, when $a = (a_{re}, a_{im}) = (\sqrt{2}, \sqrt{2})$, thus we exploit the fact that $a_{re} = a_{im}$ to save multiply instructions by writing a separate loop at the begin and end of Forward and Inverse FFT, respectively.

In Forward FFT, only the first half of the roots is used, while in Inverse FFT, only the second half is used.

5.1.1 Our Iterative SIMD FFT

Many FFT implementations prefer a recursive approach for high degree $N \geq 2^{13}$ [FJ12, BWC13], as it is more memory cache-friendly than the iterative approach. First, we decided to avoid using a vendor-specific library to maintain high portability. Secondly, we gave preference to an iterative approach to avoid function call overheads (since Falcon's $N \leq 1024$) and scheduling overheads for irregular butterfly patterns. Thirdly, we must support our compressed twiddle factor since the cost of deriving complex roots is free. Lastly, we focused on simplicity, so our code could be deployed and implemented on constrained devices and used as a starting point for hardware accelerator development.

In our literature search, we could not find either an FFT implementation or detailed algorithms fitting our needs. Hence, we wrote our own iterative FFT in `C`, then we vectorized our `C` FFT implementation. We are not aware of any published FFT implementation similar to our work.

5.2 Improved Forward FFT implementation

Similar to NTT, we use Cooley-Tukey butterflies in Algorithms 6 and 7 to exploit the first half of the roots in Forward FFT. We rewrote the Forward FFT implementation so each twiddle factor is always loaded once, and the program can take advantage of the

Algorithm 6: `CT_BF`

```
Input: $a, b, \omega$
Output: $(a, b) = (a + \omega b, a - \omega b)$
1 $t \leftarrow b * \omega$
2 $b \leftarrow a - t$
3 $a \leftarrow a + t$
```

Algorithm 7: `CT_BF_90`

```
Input: $a, b, \omega$
Output: $(a, b) = (a + i\omega b, a - i\omega b)$
1 $t \leftarrow b * (i\omega)$
2 $b \leftarrow a - t$
3 $a \leftarrow a + t$
```

Algorithm 8: In-place cache-friendly Forward FFT (`split` storage)

```
Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2} + 1)$, twiddle factor table tw
Output: $f =$ FFT($f$)
1  $\omega \leftarrow$ tw$[0][0]$
2  for $j = 0$ to $N/4 - 1$ do
3      CT_BF($f[j], f[j + N/4], \omega$)                ▷ exploit $\omega_{re} = \omega_{im}$
4      $j \leftarrow j + 1$
5  $level \leftarrow 1$
6  for $len = N/8$ to 1 do
7      $k \leftarrow 0$                                  ▷ reset $k$ at new $level$
8      for $s = 0$ to $N/2 - 1$ do
9          $\omega \leftarrow$ tw$[level][k]$            ▷ $\omega$ is shared between two loops
10         for $j = s$ to $s + len - 1$ do
11             CT_BF($f[j], f[j + len], \omega$)
12             $j \leftarrow j + 1$
13         $s \leftarrow s + (len \ll 1)$
14         for $j = s$ to $s + len - 1$ do
15             CT_BF_90($f[j], f[j + len], \omega$)
16             $j \leftarrow j + 1$
17         $s \leftarrow s + (len \ll 1)$
18         $k \leftarrow k + 1$                          ▷ increase by *one* point
19     $level \leftarrow level + 1$                      ▷ increase $level$
20     $len \leftarrow len \gg 1$                        ▷ half distance
```

Algorithm 9: `mergefft` for $N \geq 16$

```
Input: Polynomial $f_0, f_1 \in \mathbb{Q}[x]/(x^{N/4} + 1)$, twiddle factor table tw
Output: $f =$ mergefft($f_0, f_1$) $\in \mathbb{Q}[x]/(x^{N/2} + 1)$
1  $k \leftarrow 0$;    $level \leftarrow \log_2(N) - 2$
2  for $s = 0$ to $N/4 - 1$ do
3      $\omega \leftarrow$ tw$[level][k]$
4      $j \leftarrow s \ll 1$
5      $(f[j], f_1[j + 1]) \leftarrow$ CT_BF($f_0[s], f_1[s], \omega$)
6      $(f[j + 2], f[j + 3]) \leftarrow$ CT_BF_90($f_0[s + 1], f_1[s + 1], \omega$)
7      $k \leftarrow k + 1$                              ▷ increase by *one* complex point
8      $s \leftarrow s + 2$
```

Table 2: Summary of butterfly loops for NTT and FFT in Falcon $N = 512, 1024$

	$\log_2$(N)	Forward (CT butterfly)	Inverse (GS butterfly)
NTT	9	$2 + 7$	$7 + 2$
	10	$3 + 7$	$7 + 3$
FFT	5	5	5
	6	$1 + 5$	$5 + 1$
	7	$2 + 5$	$5 + 2$
	8	$1 + 2 + 5$	$5 + 2 + 1$
	9	$(2 \times 2) + 5$	$5 + (2 \times 2)$
	10	$1 + (2 \times 2) + 5$	$5 + (2 \times 2) + 1$

cache spatial and temporal locality with constant access patterns when load and store instructions are executed. All the butterflies in Algorithm 8 are computed in-place. Note that the two for loops from line 10 to 17 can be executed in parallel, which may be of interest when developing a hardware accelerator.

In a signature generation, at line 5 in Algorithm 17, Fast Fourier sampling (`ffSampling`) builds an FFT tree by traveling from top level $l = \log_2(N)$ to lower level $l-1, l-2, \dots 1$, where N is a total number of real and imaginary points. Hence, FFT implementation must support all FFT levels from $\log_2(N)$ to 1.

Our `C` FFT implementation supports all levels of Forward and Inverse FFT trivially. However, our vectorized FFT must be tailored to support all FFT levels. Instead of vectorizing $l-1$ FFT implementations, first, we determined the maximum number of coefficients that can be computed using 32 vector registers. Then, we select $l = 5$ as the baseline to compute FFT that minimizes load and store. In case $l < 5$, we unroll the FFT loop completely, and save instructions overhead. When $l \geq 5$, we use the base loop with $l = 5$ with 32 64-bit coefficients and implement additional two FFT loops. The second loop computes two FFT levels per iteration to save load and store instructions. The third loop is an unrolled version of a single FFT level per iteration. In short, using three FFT loops, we can construct arbitrary FFT level $l \geq 5$ by using the base loop with 5 levels, then a multiple of two FFT levels by the second loop. Finally, the remaining FFT levels are handled by the third loop, as shown in Table 2.

Merge FFT As shown in Algorithm 9, $f = \texttt{mergefft}(f_0, f_1) = f_0(x^2) + xf_1(x^2)$ is similar to subroutine of Forward FFT, used inside `ffSampling`. We rewrite the original implementation to share the twiddle factor ω between two butterflies. Instead of computing in-place, we write output of butterflies from (f_0, f_1) to f. The output polynomial f is double the size of the input polynomial as indicated by the name of this operation. When $N \leq 8$, we unroll the loop completely and exploit $\omega_{re} = \omega_{im}$ if possible. The Algorithm 9 works for $N \geq 16$.

5.3 Improved Inverse FFT implementation

The butterflies in Inverse FFT in Algorithm 10 and Algorithm 11 exploit complex *conjugate* by using add and subtract instructions. A tweak at no cost in line 1 of Algorithm 11 to avoid floating-point negation instruction `fneg` in line 3. Similar to Forward FFT, two loops from line 6 to 13 in Algorithm 12 can be executed in parallel and share the same twiddle factor ω. The last loop from line 17 to 21 multiplies $\frac{2}{N}$ by all coefficients of FFT, and exploits special case of twiddle factor $\omega_{re} = \omega_{im}$. We also employ three FFT loops setting described in Table 2 to implement vectorized multi-level l of Inverse FFT to maximize vector registers usage, hence improving execution time. Note that butterflies in Algorithm 12 are computed in-place.

Algorithm 10: `GS_BF`

```
Input: $a, b, \omega$
Output: $(a, b) = (a + b, (a - b)\hat{\omega})$
1 $t \leftarrow a - b$
2 $a \leftarrow a + b$
3 $b \leftarrow t * conjugate(\omega)$
```

Algorithm 11: `GS_BF_270`

```
Input: $a, b, \omega$
Output: $(a, b) = (a + b, (a - b)i\hat{\omega})$
1 $t \leftarrow b - a$                ▷ avoid negation
2 $a \leftarrow a + b$
3 $b \leftarrow t * conjugate(-i\omega)$
```

Algorithm 12: In-place cache-friendly Inverse FFT (`split` storage)

```
Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2} + 1)$, twiddle factor table tw
Output: $f =$ invFFT($f$)
1  $level \leftarrow \log_2(N) - 2$          ▷ tw index starts at 0, and N/2 re, im points
2  for $len = 1$ to $N/8$ do
3      $k \leftarrow 0$                        ▷ reset $k$ at new level
4      for $s = 0$ to $N/2 - 1$ do
5          $\omega \leftarrow$ tw[$level$][$k$]          ▷ $\omega$ is shared between two loops
6          for $j = s$ to $s + len - 1$ do
7              GS_BF($f[j], f[j + len], \omega$)
8              $j \leftarrow j + 1$
9          $s \leftarrow s + (len \ll 1)$
10         for $j = s$ to $s + len - 1$ do
11             GS_BF_270($f[j], f[j + len], \omega$)
12             $j \leftarrow j + 1$
13         $s \leftarrow s + (len \ll 1)$
14         $k \leftarrow k + 1$                ▷ increase by one point
15     $level \leftarrow level - 1$            ▷ decrease level
16     $len \leftarrow len \ll 1$              ▷ double distance
17 $\omega \leftarrow$ tw[0][0] $\cdot \frac{2}{N}$
18 for $j = 0$ to $N/4 - 1$ do
19     GS_BF($f[j], f[j + N/4], \omega$)       ▷ exploit $\omega_{re} = \omega_{im}$
20     $f[j] \leftarrow f[j] \cdot \frac{2}{N}$
21     $j \leftarrow j + 1$
```

Algorithm 13: `splitfft`(f) for $N \geq 16$

```
Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2} + 1)$, twiddle factor table tw
Output: $f_0, f_1 =$ splitfft($f$) $\in \mathbb{Q}[x]/(x^{N/4} + 1)$
1 $k \leftarrow 0$;    $level \leftarrow \log_2(N) - 2$
2 for $s = 0$ to $N/4 - 1$ do
3     $\omega \leftarrow$ tw[$level$][$k$]
4     $j \leftarrow s \ll 1$
5     $(f_0[s], f_1[s]) \leftarrow \frac{1}{2} \cdot$ GS_BF($f[j], f[j + 1], \omega$)
6     $(f_0[s + 1], f_1[s + 1]) \leftarrow \frac{1}{2} \cdot$ GS_BF_270($f[j + 2], f[j + 3], \omega$)
7     $k \leftarrow k + 1$                     ▷ increase by one point
8     $s \leftarrow s + 2$
```

Split FFT The Algorithm 13 computes $(f_0, f_1) = \texttt{splitfft}(f)$, where $f = f_0(x^2) + xf_1(x^2)$, and `splitfft` is the inverse function of `mergefft`. The output polynomials f_0 and f_1 are half the size of the input polynomial f. Similar to `mergefft`, the twiddle factor ω is shared between each butterfly unit. We unroll the loop when $N \leq 8$.

5.4 Floating-point complex instructions and data storage

ARMv8.3 supports two floating-point complex instructions: `fcadd` and `fcmla`. The floating-point complex `fcadd` instruction offers addition and counterclockwise rotation by 90 and 270 degrees: $(a + ib)$ and $(a - ib)$. The combination of `fmul` and `fcmla` instructions can perform complex point multiplication $(a*b), (a*\hat{b})$ as shown in lines 1 and 3 of Algorithms 6 and 10 by applying counterclockwise rotation by 90 and 270 degree, respectively. Note that `fcmla` has the same cycle count as `fmla, fmls`.

The floating-point complex instructions offer a convenient way to compute *single pair* complex multiplications. Conversely, the `fmla, fmls` instructions require at least *two pairs* for complex multiplication. The only difference between floating-point complex instructions and traditional multiplication instructions is the data storage of real and imaginary values during the multiplication.

Our first approach is to use floating-point complex `fmul, fcmla, fcadd` instructions, where real and imaginary values are stored adjacent in memory (`adj` storage). This data storage setting is also seen in other FFT libraries such as FFTW [FJ12], and FFTS [BWC13]. Complex multiplications using such instructions are demonstrated in Algorithms 14 and 15. The second approach uses default data storage in Falcon: real and imaginary values are split into two locations (`split` storage). The complex multiplication using `fmul`, `fmla`, and `fmls` instructions is shown in Algorithm 16. Visual demonstration of Algorithms 14–16 are in Appendix C.

To find the highest performance gain, we implemented vectorized versions of the first and second approaches mentioned above. The former approach offers better cache locality for small $l \leq 4$. However, it introduces a vector permutation overhead to compute

Algorithm 14: $c = a * b$ with `fmul, fcmla`

```
Input: $a = (a_{re}, a_{im}), b = (b_{re}, b_{im})$
Output: $c = a * b = (a_{re}b_{re} - a_{im}b_{im}, a_{re}b_{im} + a_{im}b_{re})$
1 $c \leftarrow$ fmul$(b, a[0])$                ▷ $(b_{re}a_{re}, b_{im}a_{re})$
2 $c \leftarrow$ fcmla_90$(c, a, b)$            ▷ $(b_{re}a_{re} - a_{im}b_{im}, b_{im}a_{re} + a_{im}b_{re})$
```

Algorithm 15: $c = a * conjugate(b) = a * \hat{b}$ with `fmul, fcmla`

```
Input: $a = (a_{re}, a_{im}), b = (b_{re}, b_{im})$
Output: $c = a * \hat{b} = (a_{re}b_{re} + a_{im}b_{im}, a_{im}b_{re} - a_{re}b_{im})$
1 $c \leftarrow$ fmul$(a, b[0])$                ▷ $(a_{re}b_{re}, a_{im}b_{re})$
2 $c \leftarrow$ fcmla_270$(c, b, a)$           ▷ $(a_{re}b_{re} + b_{im}a_{im}, a_{im}b_{re} - b_{im}a_{re})$
```

Algorithm 16: $c = a * b$ with `fmul, fmls, fmla`

```
Input: $a_{re}, a_{im}, b_{re}, b_{im}$
Output: $c = (c_{re}, c_{im}) = (a_{re}b_{re} - a_{im}b_{im}, a_{re}b_{im} + a_{im}b_{re})$
1 $c_{re} \leftarrow$ fmul$(a_{re}, b_{re})$              ▷ $a_{re}b_{re}$
2 $c_{re} \leftarrow$ fmls$(c_{re}, a_{im}, b_{im})$      ▷ $c_{re} = a_{re}b_{re} - a_{im}b_{im}$
3 $c_{im} \leftarrow$ fmul$(a_{re}, b_{im})$              ▷ $a_{re}b_{re}$
4 $c_{im} \leftarrow$ fmla$(c_{im}, a_{im}, b_{re})$      ▷ $c_{im} = a_{re}b_{im} + a_{im}b_{re}$
```

complex multiplication in lines 1 and 3 of Algorithms 7 and 11 of Inverse FFT using aforementioned floating-point complex instructions. Another disadvantage of the first approach is preventing the deployment of Falcon to devices that do not support ARMv8.3, such as Cortex-A53/A72 on Raspberry Pi 3/4, respectively. The latter approach can run on any ARMv8 platform. However, the second approach computes at least two complex point multiplications instead of one, and the pure C implementation is slightly faster compared to the original reference FFT implementation of Falcon. On the other hand, the pure C implementation of first approach is the fastest as shown in Appendix E. We recommend using the first approach (`adj` storage) for non-vectorized implementation.

Eventually, we chose the second approach as our best-vectorized implementation and kept the reference implementation of the first approach in our code base for community interest. For $l \leq 2$, we unroll the loop, so the multiplication is done by scalar multiplication. By rearranging vector registers appropriately using `LD2, LD4, ST2, ST4` instructions, when $l \geq 3$, the latter approach is slightly faster than the first approach when benchmarked on the high-end Apple M1 CPU.

5.5 Floating-point to integer conversion

Notably, both GCC and Clang can generate native floating-point to integer conversion instructions during compilation such as `fpr_floor, fpr_trunc` using rounding toward zero instruction `fcvtzs` except for `fpr_rint` function. As described in the 64-bit floating-point to 64-bit signed integer `fpr_rint` implementation (a constant time conversion written in `C`), the obtained assembly language code generated by Clang and GCC, respectively, is not consistent and does not address constant-time concerns described in Howe et al. [HW22]. In our implementation on a `aarch64` ARMv8, we use the rounding to the nearest with ties to even instruction `fcvtns` to convert a 64-bit floating-point to a 64-bit signed integer [2]. This single instruction, used to replace the whole `fpr_rint` implementation, costs 3 cycles on Cortex-A72/A78.

5.6 Rounding concern in Floating-point Fused Multiply-Add

Another concern while implementing vectorized code is floating-point rounding [ANB+18]. In `ref` implementation, when using the independent multiply (`fmul`) and add (`fadd, fsub`) instructions, the floating-point rounding occurs after multiplication, and after addition. In `neon` implementation, when we use Fused Multiply-Add instruction `(fmla, fmls)`, the rounding is applied only after addition.

When we repeat our experiment with `fpr_expm_p63` function used in Gaussian sampling to observe the rounding of `fmla` and `(fmul, fadd)`, the differences between them grow. In our experiment, we sample random values as input to `fpr_expm_p63` function on both Apple M1 and Cortex-A72, the differences are consistent in both CPUs [3], about 7,000 out of 100,000 output values of `fpr_expm_63` function with `fmla` are different from `(fmul, fadd)`. Example of bits flipped are shown at Table 9 in Appendix F.

We have carefully tested our implementation according to the test vectors and KATs (Known-Answer-Tests) provided by Falcon submitters. Although all tests passed, the security of the floating-point rounding differences in ARMv8 is unknown. Therefore, by default, our code uses the independent multiply (`fmul`) and add (`fadd, fsub`) instructions. We chose to optionally enable `fmla` instructions for benchmarking purposes only and observed negligible differences $3 \rightarrow 4\%$ between the two approaches in terms of the total execution time for the Sign operation, as shown in Table 3.

[2] https://godbolt.org/z/esP78P33b
[3] https://godbolt.org/z/613vvzh3Y

Table 3: Performance of Signature generation with Fused Multiply-Add instructions *enabled* (`fmla`), and *disabled* (`fmul, fadd`). Results are in *kc - kilocycles.*

CPU	`neon` Sign	`(fmul, fadd)`	`fmla`	`fmla/(fmul, fadd)`
Cortex-A72	`falcon512`	1,038.14	1,000.31	0.964
	`falcon1024`	2,132.08	2,046.53	0.960
`Apple M1`	`falcon512`	459.19	445.91	0.971
	`falcon1024`	914.91	885.63	0.968

6 Results

ARMv8 Intrinsics are used for ease of implementation and to take advantage of the compiler optimizers. The optimizers know how intrinsics behave and tune performance toward the processor features such as aligning buffers, scheduling pipeline operations, and instruction ordering [4]. In our implementation, we always keep vector register usage under 32 and examine assembly language code obtained during our development process. We acknowledge that the compiler occasionally spills data from registers to memory and hides load/store latency through the instructions supporting pipelining.

Falcon and Hawk The reference implementation of Hawk [5] uses fixed-point data type by default. Since our choice of processors supports floating-point arithmetic, we convert all fixed-point arithmetic to floating-point arithmetic and achieve a significant performance boost in reference implementation as compared to the default setting. Notably, this choice disables NTT implementation in Hawk Sign and Verify, while Falcon Verify explicitly uses integer arithmetic for NTT implementation by default. Although it is possible to vectorize NTT implementation in Hawk Verify, we consider this as future work. In both implementations, we measure the Sign operations in the dynamic signing - the secret key is expanded before signing, and we do not use floating-point emulation options.

Dilithium We select the state-of-the-art ARMv8 Dilithium implementation from Becker et al. [BHK+21].

XMSS, SPHINCS+ To construct a comprehensive digital signature comparison, we select the XMSS implementation [6] with the forward security by Buchmann et al. [BDH11], which limited the way one-time signature keys are computed to enhance security. We accelerate SHA-256 by applying the OpenSSL SHA2-NI instruction set extensions in our `neon` implementation. For SPHINCS+, we select *s, f*-variant and 'simple' instantiation to compare with lattice-based signatures. Recent work by Becker et al. [BK22] proposed multiple settings to accelerate Keccak-`f1600`, combine with scalar, `neon` and SHA3 instructions. To make sure our hash-based signatures is up-to-date, we measure and apply the best settings on Apple M1 to yield our best result for hash-based signature, as shown in Table 10 in Appendix G.

The optimized implementation of SPHINCS+ [7] already included high-speed Keccak-F1600×2 `neon` implementation. The choice of underlying THASH functions for XMSS and SPHINCS+ are indirect comparisons of SHA2 and SHA3, one can expect a similar speed-up ratio for Sign and Verify if their underlying hash functions are replaced, as shown in Table 11 in Appendix G. For both, the `ref` implementation is a pure `C` hash implementation.

[4] https://godbolt.org/z/zPr94YjYr
[5] https://github.com/ludopulles/hawk-sign/
[6] https://github.com/GMUCERG/xmssfs
[7] https://github.com/sphincs/sphincsplus

Constant-time treatment For operations that use floating-point arithmetic extensively, we use vectorized division instruction as in the reference implementation. In Falcon **V**erify, there is only integer arithmetic. Thus, the division is replaced by modular multiplication. In both operations, secret data is not used in branch conditions or memory access patterns.

Benchmarking setup Our benchmarking setup for ARMv8 implementations included MacBook Air with Apple M1 @ 3.2GHz and Raspberry Pi 4 with Cortex-A72 @ 1.8GHz. For AVX2 implementations, we used a PC based on Intel 11th gen i7-1165G7 @ 2.8GHz with Spectre and Meltdown mitigations disabled via a kernel parameter[8].

For cycle count on Cortex-A72, we used the `pqax`[9] framework . In Apple M1, we rewrote the work from Dougall Johnson[10] to perform cycle count[11]. On both platforms, we use Clang 13 with `-O3`, we let the compiler to do its best to vectorize pure C implementations, denoted as `ref` to fairly compare them with our `neon` implementations. Thus, we did not employ `-fno-tree-vectorize` option. We acknowledge that compiler enable Fuse Multiply-Add to improve performance.

We report the average cycle count of 1,000 and 10,000 executions for hash-based and lattice-based signatures, respectively. Benchmarking is conducted using a single core and a single thread to fairly compare results with those obtained for lattice-based signatures, even though hash-based signatures can execute multiple hash operations in parallel.

Results for FFT and NTT are summarized in Table 4 with FMA *enable*. When $N = 4$, we realized that our vectorized code runs slower than the C implementation due to the FFT vectorized code being too short. Therefore, we use the same `ref` implementation. Starting from $N \geq 8$, our FFT vectorized code achieved speed-up from 1.2× to 2.3× and 1.4× to 2.4× compared to the `ref` implementation of the Forward and Inverse FFT on Apple M1, and from 1.8× to 1.9× and 1.9× to 2.3× for both Forward and Inverse FFT on Cortex-A72. These speed-ups are due to our unrolled vectorized implementation of FFT, which supports multiple FFT levels and twiddle-factor sharing.

In Falcon Verify, our benchmark of Inverse NTT is without multiplication by n^{-1} because we already embed this multiplication during Montgomery conversion. For both FFT and NTT, our `neon` implementation in both Cortex-A72 and Apple M1 achieve better speed-up ratio than the AVX2 implementation, as shown in Table 4. There is no AVX2 optimized implementation of Falcon Verify, it is the same as pure REF implementation. Overall, our `neon` NTT implementation is greatly improved compared to `ref` implementation, which determines the overall speed-up in Falcon Verify.

The increasing and then decreasing trend in Cortex-A72 shows the limit of the cache on low-end devices, while high-end `neon` and AVX2 processors show a declining trend at $N = 1024$ for both Forward and Inverse FFT.

Comparison with previous work In Table 5, by comparing the cycles of `ref` implementations, we can see that our implementation outperforms the work by Kim et al. [KSS22] conducted on Jetson AGX Xavier. On Jetson AGX Xavier, the implementation improved from 1.17× to 1.69× as compared to the `ref` implementation in both **S**ign and **V**erify operations at security level 5. The corresponding improvements on Cortex-A72 and Apple M1 are from 1.51× to 2.10× and 1.49× to 2.06× as compared to the C implementation.

In a signature generation, as compared to the Kim et al. [KSS22] approach, we decided against replacing macro functions `FPC_MUL, FPC_DIV,...`. Our manual work of unrolled versions of the Forward and Inverse FFT, as well as `splitfft, mergefft, mulfft,...`,

[8]`mitigations=off` https://make-linux-fast-again.com/
[9]https://github.com/mupq/pqax/tree/main/enable_ccr
[10]https://github.com/dougallj
[11]https://github.com/GMUCERG/PQC_NEON/blob/main/neon/kyber/m1cycles.c

Table 4: Cycle counts for the implementation of FFT (with FMA *enabled*) and NTT with the size of N coefficients on Apple M1 and Cortex-A72 - `neon` vs. `ref`. On Intel i7-1165G7 - REF vs. AVX2

`Apple M1`	`Forward FFT`(*cycles*)			`Inverse FFT`(*cycles*)		
N	`ref`	`neon`	`ref/neon`	`ref`	`neon`	`ref/neon`
8	115	91	1.26	127	88	1.44
16	167	110	1.52	178	103	1.73
32	245	147	1.67	261	144	1.81
64	408	232	1.76	424	228	1.86
128	759	404	1.88	847	401	2.11
256	1,633	789	2.07	1,810	794	2.28
512	3,640	1,577	2.31	3,930	1,609	2.44
1024	7,998	3,489	2.29	8,541	3,547	2.41

	`Forward NTT`(*cycles*)			`Inverse NTT`(*cycles*)		
512	6,607	840	7.87	6,449	811^n	7.95
1024	13,783	1,693	8.14	13,335	$1,702^n$	7.83

Cortex-A72	`Forward FFT`(*cycles*)			`Inverse FFT`(*cycles*)		
N	`ref`	`neon`	`ref/neon`	`ref`	`neon`	`ref/neon`
8	100	54	1.85	116	49	2.37
16	232	92	2.52	259	88	2.94
32	516	221	2.33	570	217	2.63
64	1,132	540	2.10	1,249	543	2.30
128	2,529	1,155	2.19	2,799	1,216	2.30
256	5,474	2,770	1.98	6,037	2,913	2.07
512	11,807	5,951	1.98	13,136	6,135	2.14
1024	27,366	14,060	1.95	28,151	14,705	1.91

	`Forward NTT`(*cycles*)			`Inverse NTT`(*cycles*)		
512	22,582	3,561	6.34	22,251	$3,563^n$	6.25
1024	48,097	7,688	6.26	47,196	$7,872^n$	6.00

Intel i7-1165G7	`Forward FFT`(*cycles*)			`Inverse FFT`(*cycles*)		
N	REF	AVX2	REF/AVX2	REF	AVX2	REF/AVX2
8	50	50	1.00	53	52	1.02
16	98	73	1.34	103	77	1.34
32	192	130	1.48	206	156	1.32
64	405	270	1.50	418	273	1.53
128	787	481	1.64	873	499	1.75
256	1,640	966	1.70	1,798	1,024	1.76
512	3,486	2,040	1.71	3,790	2,138	1.77
1024	7,341	4,370	1.68	7,961	4,572	1.74

^{n} no multiplication with n^{-1} at the end of `Inverse NTT`

Table 5: Speed-up ratio comparison (with FMA *enabled*) with previous work from Kim et al. [KSS22] on Raspberry Pi 4, Apple M1, and Jetson AGX Xavier - *kc*-kilocycles

	Jetson AGX Xavier					
	`ref`(*kc*)		`neon`(*kc*)		`ref/neon`	
	S	**V**	**S**	**V**	**S**	**V**
`falcon512` [KSS22]	580.7	48.0	498.6	29.0	1.16	1.65
`falcon1024` [KSS22]	1,159.6	106.0	990.5	62.5	1.17	1.69
	Apple M1					
`falcon512` (Ours)	654.0	43.5	442.0	22.7	1.48	1.92
`falcon1024` (Ours)	1,310.8	89.3	882.1	42.9	1.49	2.08
	Raspberry Pi 4					
`falcon512` (Ours)	1,490.7	126.3	1,001.9	58.8	1.49	2.15
`falcon1024` (Ours)	3,084.8	274.3	2,048.9	130.4	1.51	2.10

contribute to greatly improving the performance of Falcon. We also modify the code logic to reduce calling `memcpy` functions during operation and sharing twiddle factors, which greatly reduces memory load and store overhead. In signature verification, our NTT speed-up is $6.2\times$ and $6.0\times$ with respect to `ref` implementation for Forward and Inverse NTT, as shown in Table 4, while Kim et al. [KSS22] only achieve less than $3\times$ speed-up. The significant speed-up is accomplished due to our signed-integer implementation of NTT, with values centered around 0, while previous work used unsigned integer NTT. Our signed integer NTT implementation utilized fast Barrett multiplication in 3 instructions by Becker et al. [BHK+21], which greatly reduces the number of multiply instructions, creating a shorter dependency chain and eliminating zip and unzip instructions.

Falcon and Dilithium In Table 6, we rank our implementations in respect to the state-of-the-art CRYSTALS-Dilithium implementation from Becker et al. [BHK+21] across all security levels. Please note that in the top rows, only DILITHIUM2, XMSS^{16}-SHA256 have security level 2, while the rest algorithms have security level 1. For all security levels of Falcon and Dilithium, when executed over messages of the size of 59 bytes, for **S**ignature generation, Falcon is comparable with Dilithium in `ref` implementations. However, the landscape drastically changes in the optimized `neon` implementations. Dilithium has an execution time $2\times$ smaller than Falcon at the lowest and highest security levels.

The speed-up ratio of Dilithium in ARMv8 is $3.3\times$ as compared to `ref` implementation. This result is due to the size of operands in the vectorized implementation. Dilithium uses 32-bit integers and parallel hashing SHAKE128/256. This leads to a higher speed-up ratio as compared to 64-bit floating-point operations, serial hashing using SHAKE256, and serial sampling in Falcon. Additionally, the computation cost of floating-point operations is higher than for integers. Hence, we only achieve $1.42\times$ speed-up (with FMA *disable*) compared to the `ref` implementation for the signature generation operation in Falcon. Although there are no floating-point computations in Falcon **V**erify, our speed-up is smaller than for Dilithium due to the serial hashing using SHAKE256. We believe that if Falcon adopts parallel hashing and parallel sampling algorithms, its performance could be further improved.

In **V**erification across all security levels, Falcon is consistently faster than Dilithium by $3.0\times$ to $3.9\times$ in both `ref` and `neon` implementations.

Hawk vs. Falcon and Dilithium At security levels 1 and 5, Hawk outperforms both Falcon and Dilithium in `ref` and `neon` implementations. The exception is the `neon`

Table 6: **S**ignature generation and **V**erification speed comparison (with FMA *disable*) over three security levels, signed 59 bytes message. `ref` and `neon` results for Apple M1. *kc*-kilocycles.

`Apple-M1` `3.2 GHz`	NIST level	`ref`(*kc*) S	`ref`(*kc*) V	`neon`(*kc*) S	`neon`(*kc*) V	`ref/neon` S	`ref/neon` V
FALCON512	I, II	654.0	43.5	459.2	22.7	1.42	1.92
HAWK512		138.6	34.6	117.7	27.1	1.18	1.27
DILITHIUM2[b]		741.1	199.6	224.1	69.8	3.31	2.86
XMSS^{16}-SHA256[x]		26,044.3	2,879.4	4,804.2	531.0	5.42	5.42
SPHINCS$^+$128s[s]		1,950,265.0	1,982.4	549,130.7	658.6	3.55	3.01
SPHINCS$^+$128f[s]		93,853.9	5,483.8	26,505.3	1,731.2	3.54	3.16
DILITHIUM3[b]	III	1,218.0	329.2	365.2	104.8	3.33	3.14
SPHINCS$^+$192s[s]		3,367,415.5	2,753.1	950,869.9	893.2	3.54	3.08
SPHINCS$^+$192f[s]		151,245.2	8,191.5	42,815.1	2,515.8	3.53	3.25
FALCON1024	V	1,310.8	89.3	915.0	42.9	1.43	2.08
HAWK1024		279.7	73.7	236.9	58.5	1.18	1.26
DILITHIUM5[b]		1,531.1	557.7	426.6	167.5	3.59	3.33
SPHINCS$^+$256s[s]		2,938,702.4	3,929.3	840,259.4	1,317.5	3.50	2.98
SPHINCS$^+$256f[s]		311,034.3	8,242.5	88,498.9	2,593.8	3.51	3.17

[b] the work from Becker et al. [BHK+21]
[s] our benchmark SPHINCS$^+$ *'simple'* variants with 1,000 iterations using Keccak-`f1600`
[x] our benchmark XMSS^{16}-SHA256 with 1,000 iterations using SHA2 Crypto instruction

Table 7: **S**ignature generation and **V**erification speed comparison (with FMA *disabled*) over three security levels, signing a 59-byte message. Ranking over the **V**erification speed ratio. `ref` and `neon` results for Cortex-A72. *kc*-kilocycles.

Cortex-A72 1.8 GHz	NIST Level	`ref`(kc) S	`ref`(kc) V	`neon`(kc) S	`neon`(kc) V	`ref/neon` S	`ref/neon` V	**V** ratio
FALCON512	I, II	1,553.4	127.8	1,044.6	<u>59.9</u>	1.49	2.09	<u>1.0</u>
HAWK512		400.3	127.1	315.9	94.8	1.26	1.34	1.6
DILITHIUM2[b]		1,353.8	449.6	649.2	272.8	2.09	1.65	4.5
DILITHIUM3[b]	III	2,308.6	728.9	1,089.4	447.5	2.12	1.63	-
FALCON1024	V	3,193.0	272.1	2,137.0	<u>125.2</u>	1.49	2.17	<u>1.0</u>
HAWK1024		822.1	300.0	655.2	236.9	1.25	1.27	1.9
DILITHIUM5[b]		2,903.6	1,198.7	1,437.0	764.9	2.02	1.57	6.1

[b] the work from Becker et al. [BHK+21]

implementation of Falcon Verify due to two polynomial multiplications in Hawk instead of one in Falcon. An optimized `neon` implementation of Hawk Verify is unlikely to be faster than Falcon Verify even if Hawk has optimized `neon` NTT implementation in its signature verification. The performance of Hawk versus Falcon and Dilithium in Sign are 3.9× and

1.9× faster and in Verify are 0.8× and 2.5× faster for the `neon` implementation at security level 1. Our `neon` implementation of Hawk achieves a similar speed-up ratio as in the case of AVX2 implementations reported in Ducas et al. [DPPvW22] (Table 1).

Lattice-based signatures vs. Hash-based signatures Notably, in Table 6, the execution times of Falcon, Dilithium, and Hawk lattice-based signatures are shorter than for XMSS and SPHINCS^{+} hash-based signatures by orders of magnitude in both `ref` and `neon` implementations. In **V**erification alone, Falcon is faster than XMSS and SPHINCS^{+} by 23.4 to 28.8×. Similarly, the Dilithium verification is faster than for XMSS and SPHINCS^{+} by 7.6 to 9.4×. The speed-up of hash-based signatures is higher than for lattice-based signatures, and it can be even greater if parallelism is fully exploited, e.g., through multithreading in CPUs or GPUs. These improvements are left for future work.

Lattice-based signature in constrained devices Aside from high-end CPUs, low-end ARMv8 CPUs are widely used in the market, from mobile devices to the Internet of Things. In normal use cases, such devices do not generate signatures frequently, but they verify signatures when receiving updates, connecting to a network, etc.

In Table 7, we rank the **V**erfication performance of Falcon, Dilithium, and Hawk. Notably, our Hawk Verify uses floating-point arithmetic while the signature **V**erifications of Falcon and Dilithium only require integer operations. We exclude hash-based signatures from this comparison due to their low performance already shown for high-speed processors in Table 6. Falcon Verify at security level 5 is faster than Dilithium at security levels 1 and 5 by 2.2× and 6.1×. Hawk outperforms Dilithium and is only slower than Falcon in Verify operation by 1.6× and 1.9× in the same security level. In combination with Table 1, it is obvious that Falcon is more efficient than Dilithium in terms of both bandwidth and workload.

7 Conclusions

Falcon is the only PQC digital signature scheme selected by NIST for standardization using floating-point operations. Unless significant changes are introduced to Falcon, floating-point instructions required in Key generation and Sign operations will continue to be a key limitation of Falcon deployment. Additionally, the complexity of serial sampling and serial hashing significantly reduces the performance of Falcon Key and Signature generation. We demonstrate that Hawk, outperforms Dilithium and has faster Signature generation than Falcon. Its performance and bandwidth may be interesting to the community.

In summary, we report the new speed record for Falcon Sign and Verify operations using NEON-based instruction on Cortex-A72 and Apple M1 ARMv8 devices. We present a comprehensive comparison in terms of performance and bandwidth for Falcon, CRYSTALS-Dilithium, XMSS, and SPHINCS^{+} on both aforementioned devices. We believe that in some constrained protocol scenarios, where bandwidth and verification performance matter, Falcon is the better option than Dilithium, and lattice-based signatures are a far better choice than hash-based signatures in terms of key size and efficiency.

Lastly, we present a 7% mismatch between the Fuse Multiply-Add instructions on ARMv8 platforms. We recommend disabling the Fuse Multiply-Add instruction to guarantee implementation correctness. Further security analysis of this behavior is needed.

References

[AABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-M4 optimizations for {R,M} LWE schemes. *TCHES*, 2020(3):336–357, June 2020.

[AAC+22] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Yi-Kai Liu, Carl Miller, Dustin Moody, et al. Status report on the third round of the nist post-quantum cryptography standardization process. 2022.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Daan Sprenkels. Faster kyber and dilithium on the cortex-M4. pages 853–871, 2022.

[ANB+18] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1369–1382, New York, NY, USA, 2018. Association for Computing Machinery.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography*, volume 7071, pages 117–129. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[BDK+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Version 3.1). 2021.

[BGPS21] Huck Bennett, Atul Ganju, Pura Peetathawatchai, and Noah Stephens-Davidowitz. Just how hard are rotations of $\mathbb{Z}^n$? algorithms and cryptography with the simplest lattice. Cryptology ePrint Archive, Report 2021/1548, 2021. https://eprint.iacr.org/2021/1548.

[BHK+19] Daniel J. Bernstein, Andreas Hülsing, Stefan Kölbl, Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. The SPHINCS+ Signature Framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, November 2019.

[BHK+21] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, Nov. 2021.

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid scalar/vector implementations of keccak and SPHINCS+ on AArch64. Cryptology ePrint Archive, Report 2022/1243, 2022. https://eprint.iacr.org/2022/1243.

[BKS19] Leon Botros, Matthias J. Kannwischer, and Peter Schwabe. Memory-Efficient High-Speed Implementation of Kyber on Cortex-M4. Technical Report 489, 2019.

[BMK+22] Hanno Becker, Jose Maria Bermudo Mera, Angshuman Karmakar, Joseph Yiu, and Ingrid Verbauwhede. Polynomial multiplication on embedded vector architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022.

[BMTR22] Nina Bindel, Sarah McCarthy, Geoff Twardokus, and Hanif Rahbari. Drive (quantum) safe! – towards post-quantum security for v2v communications. Cryptology ePrint Archive, Paper 2022/483, 2022.

[BV21] Alexandre Becoulet and Amandine Verguet. A depth-first iterative algorithm for the conjugate pair fast fourier transform. *IEEE Transactions on Signal Processing*, 69:1537–1547, 2021.

[BWC13] Anthony M Blake, Ian H Witten, and Michael J Cree. The fastest fourier transform in the south. *IEEE transactions on signal processing*, 2013.

[CAD$^+$20] David A Cooper, Daniel C Apon, Quynh H Dang, Michael S Davidson, Morris J Dworkin, Carl A Miller, et al. Recommendation for stateful hash-based signature schemes. *NIST Special Publication*, 800:208, 2020.

[CHK$^+$21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. *TCHES*, pages 159–188, February 2021.

[CJL$^+$16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on Post-Quantum Cryptography. Technical Report NIST IR 8105, National Institute of Standards and Technology, April 2016.

[CT65] James W Cooley and John W Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of computation*, 1965.

[DFG$^+$] Özgür Dagdelen, Marc Fischlin, Tommaso Gagliardoni, Özgür Dagdelen, Marc Fischlin, and Tommaso Gagliardoni. The Fiat–Shamir Transformation in a Quantum World. In *Advances in Cryptology - ASIACRYPT 2013*.

[DPPvW22] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel van Woerden. Hawk: Module LIP makes lattice signatures fast, compact and simple. Cryptology ePrint Archive, Report 2022/1155, 2022. https://eprint.iacr.org/2022/1155.

[DvW22] Léo Ducas and Wessel P. J. van Woerden. On the lattice isomorphism problem, quadratic forms, remarkable lattices, and cryptography. pages 643–673, 2022.

[FHK$^+$20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU: Specifications v1.2, 2020.

[FJ12] Matteo Frigo and Steven G Johnson. Fftw: Fastest fourier transform in the west. *Astrophysics Source Code Library*, pages ascl–1201, 2012.

[HBG$^+$18] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: eXtended Merkle Signature Scheme. RFC 8391, May 2018.

[Hül13] Andreas Hülsing. W-OTS+ - shorter signatures for hash-based signature schemes. pages 173–188, 2013.

[HW22] James Howe and Bas Westerbaan. Benchmarking and analysing the nist pqc finalist lattice-based signature schemes on the arm cortex m7. Cryptology ePrint Archive, Paper 2022/405, 2022.

[JAMK+19] Amir Jalali, Reza Azarderakhsh, Mehran Mozaffari Kermani, Matthew Campagna, and David Jao. Armv8 sike: Optimized supersingular isogeny key encapsulation on armv8 processors. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(11):4209–4218, 2019.

[KBMSRV18] Angshuman Karmakar, Jose Maria Bermudo Mera, Sujoy Sinha Roy, and Ingrid Verbauwhede. Saber on ARM. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):243–266, August 2018.

[KJK+21] Hyeokdong Kwon, Kyungbae Jang, Hyunjun Kim, Hyunji Kim, Minjoo Sim, Siwoo Eum, Wai-Kong Lee, and Hwajeong Seo. Armed frodo. In *Information Security Applications*, pages 206–217, Cham, 2021. Springer International Publishing.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. Pqm4 - Post-quantum crypto library for the {ARM} {Cortex-M4}. https://github.com/mupq/pqm4, 2019.

[KSS22] Youngbeom Kim, Jingyo Song, and Seog Chung Seo. Accelerating falcon on armv8. *IEEE Access*, 10:44446–44460, 2022.

[LJZ+20] Zhihao Li, Haipeng Jia, Yunquan Zhang, Tun Chen, Liang Yuan, and Richard Vuduc. Automatic generation of high-performance fft kernels on arm and x86 cpus. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1925–1941, 2020.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In *Advances in Cryptology – ASIACRYPT 2009*, volume 5912, pages 598–616, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[MCF19] David McGrew, Michael Curcio, and Scott Fluhrer. Leighton-micali hash-based signatures. Technical report, 2019.

[Mon85] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[NG21a] Duc Tri Nguyen and Kris Gaj. Fast neon-based multiplication for lattice-based nist post-quantum cryptography finalists. In *Post-Quantum Cryptography*, pages 234–254, Cham, 2021. Springer International Publishing.

[NG21b] Duc Tri Nguyen and Kris Gaj. Optimized software implementations of crystals-kyber, ntru, and saber using neon-based special instructions of armv8. In *Proceedings of the NIST 3rd PQC Standardization Conference (NIST PQC 2021)*, 2021.

[Ngu20] Duc Tri Nguyen. ARMv8-A implementation for Keccak-f1600. `https://github.com/cothan/NEON-SHA3_2x`, 2020.

[Por19] Thomas Pornin. New Efficient, Constant-Time Implementations of Falcon. September 2019.

[SDS18] Silvan Streit and Fabrizio De Santis. Post-Quantum Key Exchange on ARMv8-A: A New Hope for NEON Made Simple. *IEEE Transactions on Computers*, 67(11):1651–1662, November 2018.

[Sho94] P.W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Santa Fe, NM, USA, 1994. IEEE Comput. Soc. Press.

[Wes21] Bas Westerbaan. ARMv8.4-A implementation for Keccak-f1600. `https://github.com/bwesterb/armed-keccak`, 2021.

[ZZH+21] Lirui Zhao, Jipeng Zhang, Junhao Huang, Zhe Liu, and Gerhard Hancke. Efficient implementation of kyber on mobile devices. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 506–513, 2021.

A Pseudocode of Falcon

Algorithm 17: Falcon Sign

```
   Input: A message $m$, a secret key $sk$, a bound $[\beta^2]$
   Output: $sig = (r, s)$
 1 $r \leftarrow \{0,1\}^{320}$ uniformly    $c \leftarrow \texttt{HashToPoint}(r||m, q, n)$
 2 $t \leftarrow \left(-\frac{1}{q}\,\mathrm{FFT}(c) \odot \mathrm{FFT}(F), \frac{1}{q}\,\mathrm{FFT}(c) \odot \mathrm{FFT}(f)\right)$    ▷ $t \in \mathbb{Q}[x]/(\phi)$
 3 do
 4     do
 5         $z \leftarrow \texttt{ffSampling}_n(t, T)$
 6         $s = (t - z)\hat{B}$    ▷ $s \in$ Gaussian distribution: $s \sim D_{(c,0)+\Lambda(B),\sigma,0}$
 7     while $||s^2|| > \lfloor \beta^2 \rfloor$;
 8     $(s_1, s_2) \leftarrow \mathrm{invFFT}(s)$    ▷ $s_1 + s_2 h = c \bmod (\phi, q)$
 9     $s \leftarrow \texttt{Compress}(s_2, 8 \cdot sbytelen - 328)$
10 while $(s = \perp)$;
11 return $sig = (r, s)$
```

Algorithm 18: Falcon Verify

```
   Input: A message $m, sig = (r, s)$, $pk = h \in \mathbb{Z}_q[x]/(\phi)$, a bound $\lfloor \beta^2 \rfloor$
   Output: Accept or reject
 1 $c \leftarrow \texttt{HashToPoint}(r\|\mathrm{m}, q, n)$    $s_2 \leftarrow \texttt{Decompress}(\mathrm{s}, 8\cdot$ sbytelen $-328)$
 2 if $(s_2 = \perp)$ then
 3     reject
 4 $s_1 \leftarrow c - s_2 h \bmod (\phi, q)$    ▷ $|s_1| < \frac{q}{2}$ and $s_1 \in \mathbb{Z}_q[x]/(\phi)$
 5 if $\|(s_1, s_2)\|^2 \leq \lfloor \beta^2 \rfloor$ then
 6     accept ;
 7 else
 8     reject
```

B Modular Multiplication

Barrett reduction is used to compute modular reduction efficiently without actually performing costly division. For example, we want to find $z' = z \bmod q$:

$$z' = z - q \cdot \left\lfloor \frac{z}{q} \right\rceil = z - q \cdot \left\lfloor \frac{zR}{qR} \right\rceil = z - q \cdot \left\lfloor \frac{z}{R} \cdot \frac{R}{q} \right\rceil = z - q \cdot \left\lfloor \frac{z \cdot t}{R} \right\rceil \tag{3}$$

As shown in Equation 3, we can precompute $t = [R/q]$ as an integer approximate of $\frac{R}{q}$ since R and q are constants. If we choose $R = 2^w$, then the division of z/R is a simple shift right by w. We select $R = 2^{16}$ to fit machine natural width for a 14-bit prime $q = 12289$ in Falcon, which allows using convenient instructions in ARMv8 as shown in later sections.

Barrett multiplication is inspired by Barrett reduction. In our work, we use Barrett multiplication in Forward and Inverse NTT, since one factor of the multiplication is always a known constant. Hence, from Barrett reduction, we derive Barrett multiplication $z' = (x \cdot y) \bmod q$:

$$z' = (x \cdot y) - q \cdot \left\lfloor \frac{(x \cdot y)}{q} \right\rceil = z - q \cdot \left\lfloor \frac{x}{R} \cdot \frac{(y \cdot R)}{q} \right\rceil = z - q \cdot \left\lfloor \frac{x \cdot t'}{R} \right\rceil \tag{4}$$

As shown in Equation 4, this time we precompute $t' = [(y \cdot R)/q]$, since y, R and q are also constants. Exploiting known constants in Barrett multiplication yields shorter multiplication instructions, inspired by Becker et al. [BHK+21] as shown in Section 4.

Montgomery multiplication is used in point-wise multiplication in Equation 1, where two factors of the multiplication are unknown. Similarly to Barrett reduction, Montgomery reduction [Mon85] is another way to avoid costly division. To use Montgomery reduction, first, we have to convert operands to the Montgomery domain by multiplying with $q < R = 2^w$, where w is the register width, e.g., 16 or 32. We assume that one factor of multiplication is already in Montgomery domain. With $R = 2^w$, the Montgomery multiplication can be derived as follows: $z' = (xR \cdot y) \bmod q$:

$$z' = \texttt{hi}(x \cdot y + q \cdot \texttt{lo}(q' \cdot (x \cdot y))) \tag{5}$$

where `hi` and `lo` are lower and upper part of $2w$ bits, each is w-bit respectively.

C Visualizing complex point multiplication

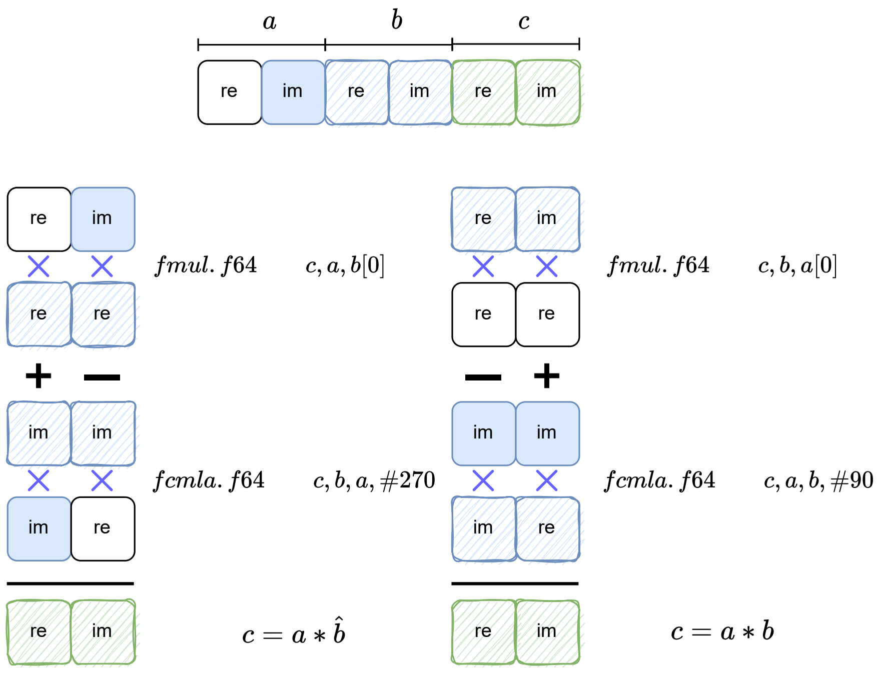

Figure 1: Single pair complex multiplication using `fmul, fcmla`. Real and imagine points are stored adjacently.

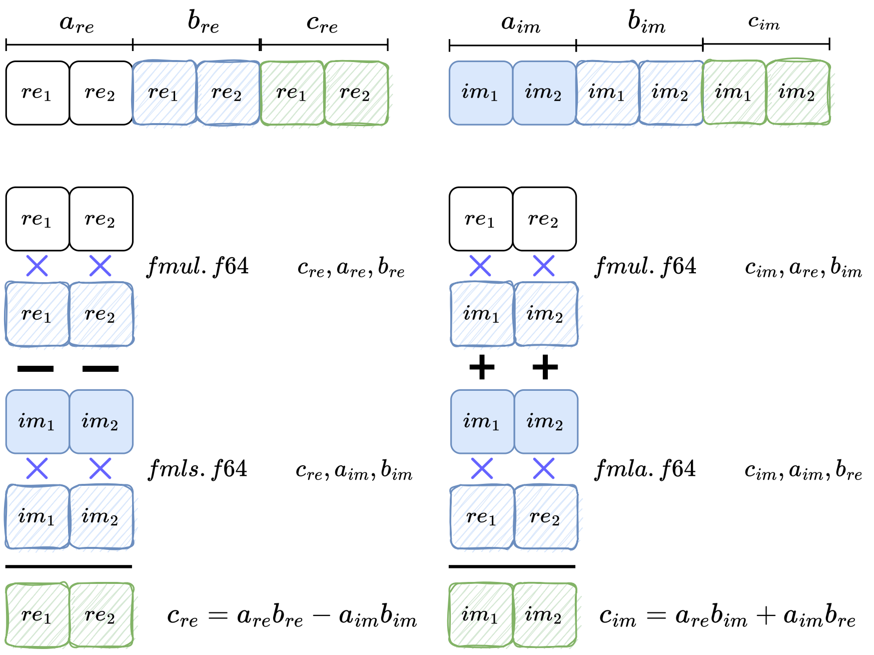

Figure 2: Two pairs complex multiplication using `fmul, fmls, fmla`. Real and imagine points are stored separately.

D Visualizing compressed twiddle factor

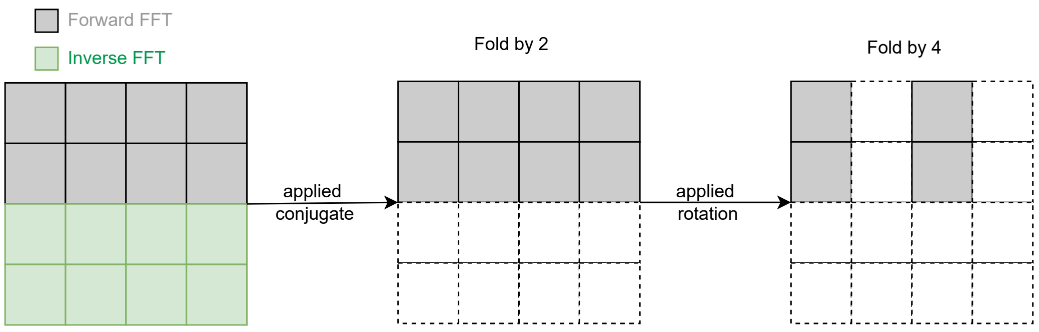

Figure 3: Deriving full twiddle factor table by applying complex conjugate and rotation.

E Compressed Fast Fourier Transform in C

Table 8: Comparison of reference (`ref`) and compressed FFT implementation written in C. `split` and `adj` are storage settings in Figure 3 and Figure 2; `-Os` optimization applied.

`Forward FFT`	Cortex-A72 (*cycles*)			`Apple M1` (*cycles*)		
N	`ref`	`split`	`adj`	`ref`	`split`	`adj`
4	28	19	17	82	78	79
8	70	56	55	99	103	102
16	162	142	137	153	147	146
32	369	346	335	216	214	213
64	849	825	804	368	365	359
128	2,022	1,983	1,954	734	735	726
256	4,709	4,606	4,512	1,576	1,561	1,540
512	10,518	10,434	10,276	3,542	3,515	3,395
1024	23,496	22,840	22,573	7,851	7,799	7,583

`Inverse FFT`	Cortex-A72 (*cycles*)			`Apple M1` (*cycles*)		
N	`ref`	`split`	`adj`	`ref`	`split`	`adj`
4	35	21	25	86	78	79
8	80	59	56	113	106	102
16	183	147	139	164	148	146
32	421	359	339	246	222	216
64	968	852	813	434	401	377
128	2,283	2,016	1,991	871	805	745
256	5,151	4,693	4,504	1,818	1,702	1,577
512	11,052	10,604	10,219	3,941	3,732	3,472
1024	24,867	23,456	23,141	8,526	8,201	7,681

F Fused Multiply-Add Rounding differences

Table 9: Floating point FMA rounding differences in our experiment of `fpr_expm_p63`. Two input, output pairs show that the bits are flipped at $(9, 11, 12)$ position.

Input	*float* value	*hex* value
(ccs, x)$_1$	0.9879566266, 0.7112028419	`0x3fef9d57371f9d57, 0x3fe6c22c7656c22c`
(ccs, x)$_2$	0.7176312343, 0.5160871303	`0x3fe6f6d5c736f6d6, 0x3fe083c9285083c9`

Output	*float* value	*hex* value
(a_1, b_1)	0.0000000015	`0x3e190af590033c00, 0x3e190af590034c00`
(a_2, b_2)	0.0000000000	`0x36d329d822c80400, 0x36d329d822c80e00`

Differences		*hex* value
$\lvert\mathrm{a}_1 - b_1\rvert$	0	`0x1000`
$\lvert\mathrm{a}_2 - b_2\rvert$	0	`0x0a00`

G Benchmark Keccak-f1600

Table 10: Benchmark Keccak-`f1600` implementation on Apple M1, with 2 to 5-ways in a single function call. On M1, Keccak-`f1600` 2-ways by [Wes21] yields the best performance. The ratio comparison between the fastest setting versus others. Results are in *cycles.*

Authors	Ways	Settings	Cycles (per hash)	Ratio
[Wes21]	2×	`neon` + SHA3	485 $\underline{(243)}$	$\underline{1.00}$
[Ngu20]	2×	`neon` + SHA3	548 (274)	1.13
[BK22]	3×	`neon` + SHA3 + scalar	1,052 (350)	1.44
[BK22]	4×	`neon` + SHA3 + scalar	2,094 (524)	2.15
[BK22]	5×	`neon` + SHA3 + scalar	3,184 (637)	2.64

Table 11: THASH-SHA256 and THASH-SHAKE speed comparison between the `C`, SHA2 Crypto instruction and 2-ways SHA3 instructions by Bas Westerbaan [Wes21] on Apple M1, measured with 64 bytes input and 32 bytes output. Results are in *cycles.*

	`C`	`neon` (per hash)	`ref/neon`
THASH-SHAKE	882	498 (249)	3.54
THASH-SHA256	1,534	268 (268)	5.72

H Forward and Inverse FFT for Adjacent storage setting

Algorithm 19: In-place cache-friendly Forward FFT (`adj` storage)

```
Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2}+1)$, twiddle factor table tw
Output: $f = \texttt{FFT}(f)$
$\omega \leftarrow \texttt{tw}[0][0]$
for $j = 0$ to $N/2 - 1$ do
    CT_BF$(f[j], f[j + N/2], \omega)$                ▷ exploit $\omega_{re} = \omega_{im}$
    $j \leftarrow j + 2$
$level \leftarrow 1$
for $len = N/4$ to 2 do
    $k \leftarrow 0$                                  ▷ reset $k$ at new $level$
    for $s = 0$ to $N - 1$ do
        $\omega \leftarrow \texttt{tw}[level][k]$     ▷ $\omega$ is shared between two loops
        for $j = s$ to $s + len - 1$ do
            CT_BF$(f[j], f[j + len], \omega)$
            $j \leftarrow j + 2$
        $s \leftarrow s + (len \ll 1)$
        for $j = s$ to $s + len - 1$ do
            CT_BF_90$(f[j], f[j + len], \omega)$
            $j \leftarrow j + 2$
        $s \leftarrow s + (len \ll 1)$
        $k \leftarrow k + 1$                          ▷ increase by *one* point
    $level \leftarrow level + 1$                      ▷ increase $level$
    $len \leftarrow len \gg 1$                        ▷ half distance
```

Algorithm 20: In-place cache-friendly Inverse FFT (`adj` storage)

```
Input: Polynomial $f \in \mathbb{Q}[x]/(x^{N/2}+1)$, twiddle factor table tw
Output: $f = \texttt{invFFT}(f)$
$level \leftarrow \log_2(N) - 2$          ▷ tw index starts at 0, and N/2 *re, im* points
for $len = 2$ to $N/4$ do
    $k \leftarrow 0$                                  ▷ reset $k$ at new $level$
    for $s = 0$ to $N - 1$ do
        $\omega \leftarrow \texttt{tw}[level][k]$     ▷ $\omega$ is shared between two loops
        for $j = s$ to $s + len - 1$ do
            GS_BF$(f[j], f[j + len], \omega)$
            $j \leftarrow j + 2$
        $s \leftarrow s + (len \ll 1)$
        for $j = s$ to $s + len - 1$ do
            GS_BF_270$(f[j], f[j + len], \omega)$
            $j \leftarrow j + 2$
        $s \leftarrow s + (len \ll 1)$
        $k \leftarrow k + 1$                          ▷ increase by *one* point
    $level \leftarrow level - 1$                      ▷ decrease $level$
    $len \leftarrow len \ll 1$                        ▷ double distance
$\omega \leftarrow \texttt{tw}[0][0] \cdot \frac{2}{N}$
for $j = 0$ to $N/2 - 1$ do
    GS_BF$(f[j], f[j + N/2], \omega)$                ▷ exploit $\omega_{re} = \omega_{im}$
    $f[j] \leftarrow f[j] \cdot \frac{2}{N}$
    $j \leftarrow j + 2$
```