Improving FALCON's Key Generation on ARMv8-A Platforms

Caio Teixeira[1], **Décio Luiz Gazzoni Filho**[2,3], **Julio César López Hernández**[3]

[1]Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas (UNICAMP) – Campinas, SP – Brazil

[2]Departamento de Engenharia Elétrica
Universidade Estadual de Londrina – Londrina, PR – Brazil

[3]Instituto de Computação
Universidade Estadual de Campinas (UNICAMP) – Campinas, SP – Brazil

caio@lasca.ic.unicamp.br, {decio.gazzoni,jlopez}@ic.unicamp.br

Abstract. *This short paper proposes two implementation techniques that may speed up the key generation routine of FALCON, a lattice-based digital signature scheme recently standardized by NIST. The first is a change in the set of primes used for splitting polynomials in NTT+RNS representation: by mixing 31-bit and 63-bit primes, we postulate that operations may be computed concurrently using both the scalar ALU (for 64-bit) and SIMD ALU (computing four 32-bit operations in a single 128-bit register). The second uses arbitrary-precision floating-point operations to compute the polynomial reduction step, which we prototype and benchmark on both Apple M1 SoC and Cortex-A72, improving on the original implementation for deeper recursion levels.*

1. Introduction

Classical cryptography has been under attack since the publishing of Shor's seminal work [Shor 1997], which describes a quantum algorithm that solves both discrete logarithm and integer factorization problems in polynomial time. *Post-Quantum Cryptography* (PQC) studies the design of cryptosystems whose security relies on problems that no quantum algorithm is expected to solve. The most prominent PQC cryptosystems are those based on structures called *lattices*, a claim confirmed by NIST's standardization process [NIST 2017], which has standardized four algorithms, three of them lattice-based.

One such standardized cryptosystem is FALCON, a lattice-based digital signature scheme. Its design focuses on minimizing communication cost – that is, the sum of both public key and signature sizes. This property is achieved by using NTRU lattices, which allow for compact representation of its bases using integer polynomials. However, post-quantum cryptosystems have higher algorithm execution time and higher communication cost than their classical counterparts. While the latter is mitigated by design, the former holds true for FALCON, and as such, efficient implementations are required to ensure the viability of switching from classical to post-quantum algorithms in practical applications.

Of the three main algorithms (key generation, signature generation and verification), FALCON's key generation is noticeably slow, even when compared to other standardized PQC cryptosystems. The most time-consuming subroutine is `NTRUSolve`,

an algorithm that operates on integer polynomials with large coefficients. To mitigate this cost, these polynomials are represented in a combination of the *Number Theoretic Transform* (NTT) and a *Residue Number System* (RNS), which speeds up operations [Pornin and Prest 2019]. Furthermore, its use of polynomial multiplications in a recursive fashion results in fast growth of coefficient bitsize. This increase is bounded by performing reductions at every level, through a generalization of Babai's nearest plane algorithm [Babai 1986], which we refer to as Babai's reduction. NTT+RNS arithmetic and Babai's reduction algorithm pose major challenges in speeding up key generation.

Performance issues due to the transition to PQC are even more critical on mobile devices, impacting energy consumption and communication latency. These devices predominantly employ the ARMv8-A architecture, transitioning to the backwards-compatible ARMv9-A on newer devices, making ARMv8-A a prominent target architecture for optimizations. Along with a 64-bit scalar ALU, the architecture also includes a separate, 128-bit-wide ALU for SIMD (*Single Instruction, Multiple Data*) operations, which may be leveraged during implementation to further improve performance. Improving FALCON's performance on this platform has been explored by Kim et al. (2022); however, to the best of our knowledge, this is the only contribution in the literature so far.

Our contributions. We present two techniques to speed up FALCON's key generation in software implementations, focusing on the `NTRUSolve` routine, and evaluate them on the ARMv8-A architecture. First, we propose using arbitrary-precision floating-point arithmetic to perform Babai's reduction in deeper levels of the routine's recursion. We provide experimental evidence of performance increase, compared to the reference implementation's technique of successive approximation of reduction factors using double-precision arithmetic, as coefficient size grows. We also propose a different layout of primes for the NTT+RNS representation of polynomials with large coefficients: by mixing 64-bit and 32-bit primes, we may leverage both scalar and SIMD ALUs in parallel.

2. Fundamentals

To contextualize the techniques we improve upon, we first define the class of NTRU lattices used by FALCON, which is the core component of key generation.

Definition 1 (NTRU Lattices)**.** *Let $n \in \mathbb{Z}$ be a power of two, q a prime, and $f, g \in \mathbb{Z}[x]/(x^n+1)$. We define two bases for the same lattice by computing different polynomials based on f, g. First, from the polynomial $h = g \cdot f^{-1} \mod q$, we define the basis $\mathbf{B}_{h,q}^{2\times 2} = [1 \;\; h \,|\, q \;\; 0]$. Furthermore, calculating $F, G \in \mathbb{Z}[x]/(x^n+1)$ that satisfy the NTRU equation*

$$f \cdot G - g \cdot F = q \bmod (x^n + 1), \tag{1}$$

the same lattice accepts another basis, namely $\mathbf{B}_{f,g}^{2\times 2} = [g \;\; -f \,|\, G \;\; -F]$.

FALCON's public key is h, and the private key is (f, g, F, G). To generate these keys, we first sample (f, g), following a discrete Gaussian distribution. Then, we solve the NTRU equation through the extended Euclid algorithm; however, performing this operation on polynomials is very costly. As FALCON employs a tower-of-fields structure [Pornin and Prest 2019], we may recursively map degree-2^m polynomials into degree-2^{m-1} polynomials through the *field norm* all the way to integers, where the equation is

easily solvable. Results are then lifted back up as the recursion unwinds, and reduced to smaller coefficient polynomials that still solve the equation through Babai's reduction.

We now define the *field norm*, used to map polynomials into smaller subfields. Then, we present two algorithms: `NTRUSolve`, which solves the NTRU equation; and `Reduce`, which performs Babai's reduction, as Algorithms 2.1 and 2.2. We refer to FALCON's documentation [Fouque et al. 2020, Section 3.8.2] for further explanations.

Definition 2 (Field Norm)**.** *The field norm $\mathcal{N}$ is a map of elements of a field $\mathbb{L}$ onto a subfield $\mathbb{K}$. We define it for a particular case of interest. Let $n \in \mathbb{Z}$ be a power of two, $\mathbb{L} = \mathbb{Q}[x]/(x^n + 1)$ and $\mathbb{K} = \mathbb{Q}[x]/(x^{n/2} + 1)$. The field norm, for this case, of a polynomial $f \in \mathbb{L}$, written as $f(x) = \sum_{i=0}^{n-1} a_i x^i$, is defined as*

$$\mathcal{N}(f) = f_0^2 - x f_1^2,$$

where $f_0(x) = \sum_{i=0}^{n/2-1} a_{2i} x^i$ and $f_1(x) = \sum_{i=0}^{n/2-1} a_{2i+1} x^i$, with $f_0, f_1 \in \mathbb{K}$. Equivalently,

$$\mathcal{N}(f)(x^2) = f(x) \cdot f(-x) \bmod (x^{n/2} + 1),$$

which is more convenient when f is represented in either FFT or NTT domains.

Algorithm 2.1 NTRUSolve$_{n,q}(f, g)$

Require: $f, g \in \mathbb{Z}[x]/(x^n + 1)$, where n is a power of two.
Ensure: Polynomials $F, G \in \mathbb{Z}[x]/(x^n + 1)$ satisfying Equation 1.

1: **if** $n = 1$ **then**
2: $(u, v, d) \leftarrow \texttt{xgcd}(f, g)$ ▷ $\texttt{xgcd}(f, g)$ finds $u, v, d \in \mathbb{Z}$ that solve $uf + vg = d$.
3: **if** $d \neq 1$ **then**
4: **return** $\perp$
5: **else**
6: $(F, G) \leftarrow (-vq, uq)$
7: **return** (F, G)
8: **else**
9: $f' \leftarrow \mathcal{N}(f)$ ▷ $\mathcal{N}(f)$ is the field norm, as per Definition 2,
10: $g' \leftarrow \mathcal{N}(g)$ ▷ and thus $f', g' \in \mathbb{Z}[x]/(x^{n/2} + 1)$.
11: $(F', G') \leftarrow \texttt{NTRUSolve}_{n/2,q}(f', g')$ ▷ $F', G' \in \mathbb{Z}[x]/(x^{n/2} + 1)$.
12: $F \leftarrow F'(x^2)g(-x)$
13: $G \leftarrow G'(x^2)f(-x)$
14: $(F, G) \leftarrow \texttt{Reduce}(f, g, F, G)$
15: **return** (F, G)

3. Implementation challenges

As the field norm is applied during `NTRUSolve`, the number of bits per coefficient grows considerably, from 4 bits up to 6320 bits on the lower recursion levels [Fouque et al. 2020, Section 4.4.3]. Furthermore, lifting operations further increase coefficient sizes, as a result of the multiplications in lines 12 and 13 of Algorithm 2.1; therefore, Babai's reduction is applied at every level, following Algorithm 2.2, so that the coefficient size of (F, G) is brought closer to (f, g) by calculating a reduction factor k, also an integer polynomial.

Algorithm 2.2 Reduce(f, g, F, G)

Require: $f, g, F, G \in \mathbb{Z}[x]/(x^n + 1)$, where n is a power of two.
Ensure: (F, G), reduced with respect to (f, g).

```
repeat
    k ← ⌊(Ff* + Gg*)/(ff* + gg*)⌉        ▷ (Ff* + Gg*)/(ff* + gg*) ∈ Q[x]/(x^n + 1), k ∈ Z[x]/(x^n + 1)
    F ← F − kf
    G ← G − kg
until k = 0
return (F, G)
```

Calculating k requires polynomial multiplication and inversion. This is simpler inside the *Fast-Fourier Transform* (FFT) domain, which maps polynomials into a set of complex numbers, represented as a double-precision floating-point number in the reference implementation. Due to the 53-bit precision of this representation, only about 30 bits of coefficient size are eliminated at each iteration, as k is only roughly approximated. As (F, G) are around triple the size of (f, g), a large number of iterations are required on deeper recursion levels – at worst, we reduce coefficients of 9500 bits to around 3100 bits.

Another challenge in handling these large integers is the choice of both representation and polynomial arithmetic algorithms. Efficient arithmetic may be performed using the *Number Theoretic Transform* (NTT), which maps an integer polynomial in $\mathbb{Z}_p[x]/(x^n + 1)$ to a set of n evaluations in $\mathbb{Z}_p$. However, to apply this transform, we must either choose a prime large enough to fit even the largest coefficients, or change primes at every recursion level, both of which introduce heavy representation overheads.

To solve this challenge, the authors of FALCON implement a Residue Number System (RNS), representing polynomial f as a set of polynomials $f_j = f \bmod p_j$, for distinct small primes p_j, which may be reversed by applying the Chinese Remainder Theorem. Furthermore, if every p_j is "NTT friendly", we may apply NTT for each split polynomial and NTT-domain arithmetic instead, greatly reducing cost. However, we note that calculating Babai's reduction's k may not be performed in NTT+RNS due to the incompatibility of division in $\mathbb{Q}[x]$ followed by rounding to $\mathbb{Z}[x]$ and arithmetic in $\mathbb{Z}_p[x]$, and so polynomials must be reconstructed before reduction, introducing some overhead.

By upper bounding coefficient sizes at each level, we may split polynomials using the minimum number of primes required to fit them; then, as recursion deepens, we increase the number of primes with little overhead, calculating new moduli only for the required new primes. In the reference implementation, all primes are 31 bits in size, as to make computations easy with pure integer arithmetics [Fouque et al. 2020, Section 4.4.3].

4. Proposed techniques

We propose two distinct techniques to speed up key generation. The first consists of modifying the set of primes used for the NTT+RNS representation. Recall that the reference implementation uses 31-bit primes for the RNS, amounting to 520 primes in the worst case. The large prime count slows down the splitting and reconstruction of polynomials in NTT+RNS representation, so reducing the number of primes may speed up the algorithm. Furthermore, we may leverage more parallelism by using both scalar and SIMD

ALUs at the same time, greatly speeding up computations. Thus, we propose splitting the prime set into 63-bit primes and 31-bit primes, keeping the productory bitsize the same. Polynomials in 63-bit moduli are operated on by the scalar ALU, while the SIMD ALU processes 31-bit moduli for 4 polynomials in parallel using 4×32-bit vector instructions, as NEON does not provide multiplication instructions for 64-bit lanes.

For the second technique, we replace the double-precision approximation of k in line 2 of Algorithm 2.2 by an exact computation using arbitrary precision in the deeper recursion levels. We now sketch why this is promising from a computational complexity perspective. Let b be the bitsize of the largest coefficients in (F, G), and d be the polynomial degree at a given recursion level. Each iteration in using double-precision removes a fixed number of bits from the polynomials (F, G); thus, the number of iterations in the loop of Algorithm 2.2 is $O(b)$. Lines 3 and 4 perform, for fixed d, a fixed number of operations on integers of b bits multiply kf; this cost is dominated by the cost of b-bit coefficient multiplications, which we denote by $M(b)$. Since the reference implementation uses the CPU's native double-precision arithmetic instructions, with assumed unit cost, computing line 2 costs $O(1)$ for fixed d. Thus, the cost of Algorithm 2.2 in the reference implementation is $O(b\,M(b))$. Our technique computes line 2 in arbitrary-precision floating-point arithmetic; for fixed d, this requires a fixed number of arbitrary-precision operations, each costing $O(M(b))$. The cost of lines 3 and 4 remain unchanged at $O(M(b))$. Since we compute k exactly, a single execution of lines 2, 3 and 4 is sufficient; therefore, the cost of our algorithm is $O(M(b))$, a factor of b better than the reference implementation's.

5. Experimental results

To prototype our second proposal, we implemented two versions of `Reduce` (Algorithm 2.2) using FLINT [Hart 2010], a C library that provides polynomial, as well as arbitrary-precision integer and floating-point, arithmetic. The first implements the algorithm using a double-precision floating point variables for FFT when calculating k, following the technique used by the reference implementation. The second uses arbitrary-precision floating-point variables for FFT, finishing the whole reduction in a single step.

We benchmarked the results using Google Benchmark on both the Apple M1 system-on-chip and a Cortex-A72 processor. In Table 1, we present the timings of each version, for each recursive call depth, and the average bitsize of the polynomials (f, g) and (F, G) before reduction. The depths range from 0 to 9 (where 10 is where the GCD is computed, requiring no reduction), following the `Falcon1024` parameter set.

We notice a trend in Table 1: double precision performs better than arbitrary precision in the initial stages of recursion, while the roles are reversed in the bottom stages. The former can be attributed to the overhead of arbitrary-precision arithmetic for small coefficients and emulation of floating-point using integer operations, while double precision arithmetic has native hardware support. In deeper recursion levels with larger coefficients, more efficient algorithms can be used for arbitrary precision, while double precision arithmetic requires more iterations of the costly reduction steps. This corroborates our complexity analysis, and suggests a fruitful avenue for further research. However, we also note that while our proposed techniques do not impact the scheme's theoretical security, libraries used in our prototype may not perform constant-time operations; imposing such a requirement may have performance implications.

Table 1. Comparison of key generation's reduction step using 53-bit precision (`double` C type) and arbitrary-precision (using FLINT)

Recursion depth	# of coeffs.	Avg. Bitsize (f, g)	Avg. Bitsize (F, G)	Apple M1 double precision	Apple M1 arbitrary precision	Cortex-A72 double precision	Cortex-A72 arbitrary precision
0	1024	4.00	19.61	$\mathbf{1178}\mu$s	23173μs	$\mathbf{5801}\mu$s	137958μs
1	512	10.99	39.82	$\mathbf{594}\mu$s	11132μs	$\mathbf{2941}\mu$s	64942μs
2	256	24.07	78.20	$\mathbf{561}\mu$s	5402μs	$\mathbf{2715}\mu$s	32831μs
3	128	50.37	153.65	$\mathbf{534}\mu$s	3071μs	$\mathbf{2580}\mu$s	18529μs
4	64	101.62	303.49	$\mathbf{496}\mu$s	1833μs	$\mathbf{2715}\mu$s	10848μs
5	32	202.22	599.81	$\mathbf{686}\mu$s	921μs	$\mathbf{3662}\mu$s	5915μs
6	16	400.67	1188.68	704μs	$\mathbf{522}\mu$s	3879μs	$\mathbf{3725}\mu$s
7	8	794.17	2361.84	773μs	$\mathbf{279}\mu$s	4437μs	$\mathbf{1809}\mu$s
8	4	1576.87	4703.30	982μs	$\mathbf{189}\mu$s	6870μs	$\mathbf{1206}\mu$s
9	2	3138.35	9403.29	1486μs	$\mathbf{125}\mu$s	10024μs	$\mathbf{802}\mu$s

With regards to future work, we believe using lower-overhead techniques, such as double-double and quad-double arithmetic, might further speed up reduction in intermediate recursion levels. As for NTT+RNS, we must investigate the concrete gains of our proposed parallelization, reimplementing and benchmarking operations. We also note that the balance of primes may depend on the target microarchitecture – especially the number of pipelines and instruction throughput –, which requires more investigation. Furthermore, Intel architectures may also benefit from the same techniques due to architecture similarities in terms of scalar and SIMD ALUs.

References

Babai, L. (1986). On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13.

Fouque, P.-A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., and Zhang, Z. (2020). Falcon: Fast-Fourier lattice-based compact signatures over NTRU. Submission to the NIST Post-Quantum Cryptography Standardization Project. `https://falcon-sign.info/falcon-round3.zip`.

Hart, W. B. (2010). Fast Library for Number Theory: An Introduction. In *Proceedings of the Third International Congress on Mathematical Software*, ICMS'10, pages 88–91, Berlin, Heidelberg. Springer-Verlag. `https://flintlib.org`.

Kim, Y., Song, J., and Seo, S. C. (2022). Accelerating Falcon on ARMv8. *IEEE Access*, 10:44446–44460.

NIST (2017). Post-Quantum Cryptography. `https://csrc.nist.gov/Projects/post-quantum-cryptography/`.

Pornin, T. and Prest, T. (2019). More efficient algorithms for the NTRU key generation using the field norm. In Lin, D. and Sako, K., editors, *Public-Key Cryptography – PKC 2019*, pages 504–533, Cham. Springer International Publishing.

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509.