# Improved (Again) Key Pair Generation for Falcon, BAT and Hawk

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

3 July, 2025

**Abstract.** In this short note, we describe some further improvements to the key pair generation process for the Falcon and Hawk lattice-based signature schemes, and for the BAT key encapsulation scheme, in a fully constant-time way and without any use of floating-point operations. Our new code is slightly faster than our previous implementation, and, more importantly for small embedded systems, uses less RAM space.

## 1 Solving the NTRU Equation

The Falcon signature scheme[5], currently in the process of being standardized by NIST under the name "FN-DSA", is based on a NTRU lattice. The Hawk signature scheme[3,2] is part of the NIST process for post-quantum cryptography standardization[6] (additional signature schemes); its key pairs use NTRU lattices very similar to those of Falcon key pairs. The BAT key encapsulation mechanism[4] again uses such NTRU lattices. Most of the runtime cost of generating a key pair for any of these three schemes lies in the step known as *solving the NTRU equation*. An algorithmically efficient method was presented in[8]; an optimized implementation was reported in[7]. This note describes an improvement on the latter, resulting in gains in both speed and temporary RAM usage.

### 1.1 Notations

We reuse the notations of [7]; see [8] for a more complete mathematical treatment. For a degree $n = 2^\ell$, we consider the ring $\mathcal{R}_n = \mathbb{Z}[X]/(X^n + 1)$, i.e. the polynomials with integer coefficients, taken modulo the cyclotomic polynomial $\phi_n = X^n + 1$. For all three schemes, the main degrees $n$ of interest are $n = 512$ and $n = 1024$, with $n = 256$ being also investigated as a lesser-security setting which may be appropriate for low-power, short-term applications. We can condiser $\mathcal{R}_n$ as a sub-ring of $\mathcal{Q}_n = \mathbb{Q}[X]/(X^n + 1)$, which is a field (since $X^n + 1$ is irreducible over $\mathbb{Q}[X]$).

For any $f \in \mathcal{R}_n$, we can define $f^\diamond(X) = f(-X)$ (i.e. we negate coefficients with an odd exponent); the product $ff^\diamond$ then only has coefficients with even exponents, and can be expressed as an expansion of a half-degree polynomial: $ff^\diamond = N(f)(X^2)$, with $N(f) \in \mathcal{R}_{n/2}$. If we consider $\mathcal{Q}_n$ as a degree-2 field extension of $\mathcal{Q}_{n/2}$, then $f^\diamond$ is the *Galois conjugate* of $f$, and $N(f)$ if the *field norm* of $f$.

Any $f \in \mathcal{Q}_n$ can be interpreted geometrically as a representation of a matrix $\mathcal{C}_n(f) \in \mathbb{C}^n \times \mathbb{C}^n$, with row $i$ consisting of the coefficients of $x^i f \bmod \phi_n$. Sums and products of polynomials in $\mathcal{Q}_n$ then map to sums and products of matrices. Over $\mathbb{C}^n \times \mathbb{C}^n$, we can define the

*Hermitian adjoint* $f^*$ with $f^*(\gamma) = \overline{f(\gamma)}$ for all roots $\gamma$ of $\phi_n$. With $\phi_n = X^n + 1$, we have $f^* = f(1/X)$ (because coefficients of $f$ are real, and $\overline{\gamma} = 1/\gamma$ for all roots $\gamma$ of $\phi_n$), which in turn means that $f^* = f_0 - \sum_{i=1}^{n-1} f_i X^{n-i}$. Note that if $f \in \mathcal{R}_q$, then $f^* \in \mathcal{R}_q$ as well.

## 1.2 NTRU Equation

For all three schemes, the private key includes two polynomials $f, g \in \mathcal{R}_q$, with coefficients chosen with a given Gaussian or binomial distribution centred on zero. Solving the *NTRU equation* consists in finding two other polynomials $F, G \in \mathcal{R}_q$ with small coefficients (in practice, coefficients of $F$ and $G$ are all in $[-127, +127]$) such that $fG - gF = q \bmod \phi_n$ for a given, fixed integer $q$ ($q = 12289$ for Falcon; $q = 1$ for Hawk and BAT). The solution is in general not unique, and solutions cannot necessarily be easily computed from each other; moreover, no solution is really better than any other. In lattice terms, $(f, g)$ and $(F, G)$ together are a short (and private) basis of the lattice of degree $2n$ whose public basis is $(1, h)$ and $(0, q)$, for $h = g/f \bmod \phi_n \bmod q$.

If $(F, G)$ is a solution to the NTRU equation, possibly with large coefficients, then a *reduced* solution can be obtained through Babai's round-off algorithm[1], leveraging the inner product and norm corresponding to the Hermitian adjoint. We compute the polynomial $k$ with integer coefficients:

$$k = \left\lfloor \frac{Ff^* + Gg^*}{ff^* + gg^*} \right\rceil \tag{1}$$

(coefficients of the division result, nominally in $\mathbb{Q}$, are rounded to the nearest integer), then replace $(F, G)$ with $(F - kf, G - kg)$. Heuristically, the obtained reduced solution has small enough coefficients for our key generation goal. The reduction of $(F, G)$ relatively to $(f, g)$ is algorithm Reduce from [8].

## 1.3 Algorithm Outline

We present here the solving algorithm, as used in [7], following algorithm TowerSolverR from [8]:

1. If $n = 1$, then $f$ and $g$ are really integers. We can compute their extended GCD $\delta$, yielding integers $u$ and $v$ such that $uf - vg = \delta$, $0 \leq u < g$ and $0 \leq v < f$. If $\delta$ is not a divisor of $q$, then there is no solution to the NTRU equation. Otherwise, a solution is $(F, G) = (vq/\delta, uq/\delta)$.
2. Otherwise, we have $n = 2^\ell$ for $\ell \geq 1$. We then do the following:
   (a) Compute the Galois norms $f' = N(f)$ and $g' = N(g)$, and invoke TowerSolverR recursively on $(f', g')$ and degree $n/2$; this yields $(F', G')$ such that $f'G' - g'F' = q \bmod \phi_{n/2}$.
   (b) Define $F = F'(X^2)g^\diamond$ and $G = G'(X^2)f^\diamond$. Since $f'(X^2) = ff^\diamond$ and $g'(X^2) = gg^\diamond$, this implies that $(F, G)$ is a solution to the NTRU equation.
   (c) Use Reduce to ensure that coefficients of $(F, G)$ are as small as possible, and return that reduced solution.

The use of Reduce is crucial to the efficiency of the algorithm: while $N(f)$ and $N(g)$ have half the degree of $f$ and $g$, their coefficients are about twice longer; at the deepest level of

the reduction, when $n = 1$, $f$ and $g$ are big integers that use thousands of bits. The unreduced $(F, G)$ obtained at step 2b have coefficients typically 2.5 times longer than those of $(f, g)$ at that recursion depth. If we did not apply Reduce at each recursion level, then RAM requirements for intermediate values would balloon to several megabytes, and runtime costs would be similarly high.

## 1.4 Costs

The main implementation costs of the algorithm above are the following:

- At low recursion depths, polynomials have high degree. In order to speed up polynomial multiplication, computations (other than Reduce) can be done in the field $\mathbb{Z}_p$ of integers modulo a small prime $p = 1 \bmod 2n$, in which $X^n + 1$ splits and the number theoretic transform (NTT) can be applied.
- At deeper levels, a simpler (quadratic) multiplication has better performance, but since coefficients are very large at these levels, we must work modulo several distinct moduli $p_i$; in other terms, we use a residue number system (RNS). We need to use up to 308 distinct moduli (for Falcon, at $n = 1024$). For code footprint reasons, it is not practical to store precomputed tables of roots of $\phi_n$ for each modulus $p_i$, as used by the NTT; these tables must be reconstructed dynamically.
- Since coefficients grow during the recursion descent, we must compute each coefficient modulo *additional* moduli $p_i$; this entails rebuilding the coefficient out of RNS and into a big integer in some appropriate basis (we use $2^{31}$ as convenient basis), so that the integer can be further reduced modulo each new $p_i$. This rebuilding uses the Chinese remainder theorem (CRT). Similarly, when going up the recursion, Reduce must work with big integers, again involving the CRT. In practice, most of the cost is spent switching between normal, RNS, and RNS+NTT representations of polynomials.
- Reduce involves a division of polynomials; computing it exactly would be quite expensive (megabytes of RAM, and several seconds worth of CPU time). Instead, we must use some approximation, with a floating-point or fixed-point representation. For better support of low-power embedded devices with no hardware FPU, we use a 64-bit fixed-point representation (32 fractional bits). The fast Fourier transform (FFT) is used to compute the division and products of polynomials. Since approximations are used, only a few bits are "shaved off" the coefficients of $(F, G)$, and the process must be applied repeatedly until $(F, G)$ coefficients are about the same size as those of $(f, g)$.
- The description above is recursive (TowerSolverR invokes itself with a half-degree); on small embedded systems, this involves some non-negligible stack space usage (maximum recursion depth is $\log n$), which can be a problem in practice since stack space is usually very restricted on such systems. Our implementation is really iterative, in a way similar to algorithm TowerSolverI from [8]: at each recursion level, $(f, g)$ are recomputed if they could not be stored in a temporary area (typically, this happens for the 3 or 4 outermost recursion levels). Such recomputations imply a runtime overhead of about 15%, but help with saving RAM space.

A less obvious cost is implied by the *failure rate*. In a secure implementation, all these steps should be *constant-time*, i.e. have a runtime cost and memory access pattern that are

independent of the actual secret values. In particular, the exact lengths (in bits) of all coefficients of $f$, $g$, $F$ and $G$, at every recursion level, are secret. All computations must therefore work by assuming that the involved integers have sizes ranging within some limits obtained by measuring averages over many key pair generation instances. Similarly, we must assume that each Reduce pass successfully shrank $(F, G)$ coefficients by some number of bits; we cannot perform extra passes conditionally on the actual values of these coefficients, since that would leak secret information through timing-based side channels. All these heuristics may fail, in which case the final output is non-valid. We thus have to check in some way that the final result is correct (e.g. by verifying the NTRU equation), and if a failure occurred then we must start the key pair generation again, with a new $(f, g)$ pair.

Such failures are unproblematic from a security point of view: if a $(f, g)$ pair failed, then we discard it, and while an outsider may observe that the pair failed through side channels, this is not a problem since the discarded pair, by being discarded, becomes non-secret. This hinges on the idea that candidate $(f, g)$ pairs are obtained through a sampling process that feeds on a cryptographically secure random source, so that even complete information on the output of the source for many candidate pairs yields no usable information about the next pair that will be generated. Discarding $(f, g)$ pairs does imply, mechanically, a reduction of the space of possible key pairs, but that reduction is slight (e.g. if half of all valid candidates are discarded through some heuristic failure, then this can only entail, at worst, a loss of 1 bit of security). A high failure rate, though, increases the runtime cost of key pair generation.

All the discussion above is about the implementation described in [7]. In the next section, we will describe the new improvements.

## 2 Improved Implementation Techniques

Consider the reduction step in algorithm TowerSolverR: we have an unreduced $(F, G)$ and we want to compute $k$, as per equation 1. We know (or at least heuristically assume, in a constant-time implementation) that $(F, G)$ is an unreduced solution to the NTRU equation $fG - gF = q$. We can thus write:

$$\begin{aligned} k &= \left\lfloor \frac{Ff^* + Gg^*}{ff^* + gg^*} \right\rceil \\ &= \left\lfloor \frac{Fff^* + (q + gF)g^*}{(ff^* + gg^*)f} \right\rceil \\ &= \left\lfloor \frac{F}{f} + \frac{qg^*}{(ff^* + gg^*)f} \right\rceil \end{aligned}$$

The second fraction in that last expression is, in practice, negligible most of the time (the intuition is that the denominator is "three times larger" than the numerator, hence that fraction is very small and usually has little or no impact on the actual value of $k$). We can thus use:

$$k \approx \left\lfloor \frac{F}{f} \right\rceil \tag{2}$$

This expression does not use either $G$ nor $g$, which opens the possibility of not actually computing $G$ in all intermediate steps. We define ReducePartial as a variant of Reduce which only receives $f$ and $F$, and returns $F - \lfloor F/f \rceil f$.

This leads to a modified recursive algorithm; let's call it TowerSolverPartialR. The algorithm receives $(f, g)$ as input, and returns only $F$ ("mostly" reduced), leaving $G$ implicit:

1. If $n = 1$, then, as in TowerSolverR, an extended GCD computation $uf - vg = \delta$ is performed, then $F = vq/\delta$ is returned (if $\delta$ does not divide $q$, then a failure is reported).
2. Otherwise, $n \geq 2$, and the following is performed:
   (a) Compute the Galois norms $f' = N(f)$ and $g' = N(g)$, and invoke TowerSolverPartialR recursively on $(f', g')$ and degree $n/2$; this yields $F'$ such that $q + g'F'$ is a multiple of $f'$ (i.e. there implicitly exists a $G'$ such that $(F', G')$ is a solution to the NTRU equation relatively to $(f', g')$).
   (b) Define $F = F'(X^2)g^\diamond$. $g$ can be discarded after this step.
   (c) Use ReducePartial on $f$ and $F$ to get a mostly reduced $F$. Since the value of $k$ computed in each step of ReducePartial is approximate, this might not yield the exact same $F$ and Reduce would, but it is still heuristically good enough.

For computing the complete solution $(F, G)$, the actual value of $G$ must be obtained, using $G = (q - gF)/f \bmod \phi_n \bmod p$, working modulo a suitable small prime $p$ (we choose $p = 1 \bmod 2n$ so that the NTT can be applied). Once $G$ has been recomputed, we can apply a final round of Reduce (not ReducePartial) to ensure a fully reduced result. These steps can be merged with the top-level call to TowerSolverPartialR: once we get $F'$ from the first recursive call (at degree $n/2$), we compute $F$ and then immediately $G$ (working modulo $p$), and we apply Reduce.

**Final Validation.** If all coefficients of $f$, $g$, $F$ and $G$ are at most 127 (in absolute value), then coefficients of $fG - gF \bmod \phi_n$ cannot exceed $127^2 \cdot 2n$ in absolute value. With $n \leq 1024$, this is less than $2^{25}$. It follows that if $p > 2^{26}$, then the fact that $fG - gF = q \bmod \phi_n \bmod p$ implies that the same relation holds over $\mathbb{Z}$ (without reduction modulo $p$). If we compute $G$ modulo $p$, then call Reduce to get a value of $k$ that we apply by working modulo $p$, and finally we check that all coefficients of $F$ and $G$ are in the expected range, then we already know that the solution is valid over the integers, and we do not need any further validation step. The result is verified to be correct at no extra cost.

**Intermediate Validation.** While the final steps of the computation inherently validate the result, it is beneficial to abort the processing earlier in case of a failure due to an unmet heuristic. At the deepest level, we can still check that the GCD is a multiple of $q$. For intermediate recursion levels, the implementation in [7] used a check modulo a small prime $p$: given intermediate $f$, $g$, $F$ amd $G$ (with possibly large coefficients), it was checked that $fG - gF = q \bmod \phi_n \bmod p$; if the reduction had failed and a value was truncated, with high probability the relation would not hold modulo $p$ and the processing of that candidate $(f, g)$ could be abandoned immediately. In TowerSolverPartialR, we do not have $G$ and do not keep $g$, preventing the use of that check. However, we have another, simpler and in fact more efficient check available: when $k$ is computed in ReducePartial and applied (by subtracting $kf$ from $F$), the process assumes that $F$ has been reduced by some bits; whenever the assumed length of $F$ becomes such that we can remove the top 31-bit limb from all coefficients of $F$, we can check that the removal is valid, i.e. that indeed the top limb of each coefficient is redundant with the sign bit of the limb immediately below it. If the reduction goes astray in some

way, then, with very high probability, subsequent reduction steps will not be able to reduce $F$ properly, and this will be reliably detected by this simple limb check. Note that this check is for performance only, and as such does not need to always work; the final validation will still ensure that the returned final result is valid (see previous discussion).

**Cost Reduction.** TowerSolverPartialR uses fewer computations than TowerSolverR, which improves speed and, more importantly, somewhat reduces storage requirement for temporary values. On the other hand, the approximate nature of the computation of $k$ tends to diminish the reduction efficiency, leading to a higher failure rate, which increases the runtime cost of the overall key generation process[1]. To reduce that failure rate, we reduce the number of bits assumed to be removed by each iteration of ReducePartial: this increases the number of needed iteration and thus the reduction cost, but reduces the overall failure rate; the optimal (or close to optimal) parameters were selected through benchmarking. Since the relative costs of the various steps may vary depending on the implementation platforms, modifying this parameter (`reduce_bits` field in the `ntru_profile` structure in our implementation) might yield some slight speed improvements on specific platforms.

For small embedded systems, RAM requirements are often a stricter constraint than computation speed, because application timing requirements usually have a somewhat flexible nature: in many cases, the ultimate limit is the human user's patience, which can be somewhat taxed to some extent, while RAM limitations come from the hardware itself and cannot be exceeded at all without changing the hardware. In [7], RAM requirements were $26n$ bytes (for all three schemes), already an improvement over previous implementations (which required $34n$, $31n$ or $48n$ bytes, for BAT, Falcon and Hawk, respectively); in this new code, this space usage was reduced to $22n$ bytes (again for all three algorithms).

To be fair, the RAM usage reduction might not matter much in the case of Falcon, since the system that generates the key pair will often also be used to compute signatures, and the Falcon signature generation process already requires more RAM[2]. However, it certainly helps for BAT and Hawk, in which the other private key operations (decapsulation and signature generation, respectively) have smaller RAM requirements than key pair generation.

# 3 Benchmarks

Our new implementation was pushed to:

`https://github.com/pornin/ntrugen`

The older version (corresponding to [7]) can be obtained by using the tag `2023/290` (matching the ePrint reference of that paper).

The new implementation performance on x86 and ARM Cortex M4 is summarized in table 1.

[1] This effect seems to be greater for Falcon, because it uses a larger value of $q$, making the approximation of $k$ "less exact" than in the case of Hawk or BAT.

[2] Implementation at `https://github.com/pornin/c-fn-dsa` currently uses $59n$ bytes for generating a signature.

| Algorithm | x86 w/ AVX2 (Mcyc) | | | ARM Cortex M4 (Mcyc) | | |
|---|---|---|---|---|---|---|
| | Old | New | Gain | Old | New | Gain |
| BAT-128-256 | 3.48 | 2.81 | ×1.24 | 23.16 | 19.18 | ×1.21 |
| BAT-257-512 | 11.24 | 9.20 | ×1.22 | 69.19 | 57.84 | ×1.20 |
| BAT-769-1024 | 44.91 | 41.94 | ×1.07 | 233.66 | 212.45 | ×1.10 |
| Falcon-256 | 3.67 | 3.03 | ×1.21 | 24.60 | 23.10 | ×1.06 |
| Falcon-512 | 11.58 | 9.48 | ×1.22 | 73.84 | 66.74 | ×1.11 |
| Falcon-1024 | 50.26 | 46.39 | ×1.08 | 287.94 | 286.48 | ×1.01 |
| Hawk-256 | 1.89 | 1.59 | ×1.19 | 18.66 | 15.99 | ×1.17 |
| Hawk-512 | 7.68 | 6.11 | ×1.26 | 47.92 | 38.56 | ×1.24 |
| Hawk-1024 | 44.22 | 39.69 | ×1.11 | 226.35 | 192.31 | ×1.18 |

**Table 1:** Performance of the new implementation of key pair generation on x86 (Intel i5-8259U "Coffee Lake", 2.3 GHz, 64-bit mode) and on ARM Cortex M4.
Compiler on x86 is Clang-18.1.3 with flags: `-O2 -mavx2`.
Compiler on ARM is GCC-13.2.1 with flags: `-O2 -mcpu=cortex-m4`.
Speed values are expressed in millions of clock cycles. "Old" code is from [7]. The gain is the old value divided by the new one.

These measurements have all been performed on the same test systems as the previous implementation, with the same parameters (newer compiler versions are used, but benchmarking the older code with the new compilers shows that this version update does not noticeably change performance).

Temporary RAM usage is the same for both architectures and for all three schemes, at $22n$ bytes (hence 5632, 11264 and 22528 bytes, for $n = 256$, 512 and 1024, respectively). Previous code needed $26n$ bytes; gain is thus 1.18. As in [7], stack space is not counted here, but the non-recursive implementation manages to keep the cost below 1 kilobyte or so.

## References


1. L. Babai, *On Lovász' lattice reduction and the nearest lattice point problem*, Proceedings on STACS 85 2nd annual symposium on theoretical aspects of computer science, pp .13-20, 1985.
2. J. Bos, O. Bronchain, L. Ducas, S. Fehr, Y.-H. Huang, T. Pornin, E. Postlethwaite, T. Prest, L. Pulles and W. van Woerden, *Hawk: a signature scheme inspired by the Lattice Isomorphism Problem*, https://hawk-sign.info/
3. L. Ducas, E. Postlethwaite, L. Pulles and W. van Woerden, *Hawk: Module LIP makes Lattice Signatures Fast, Compact and Simple*, https://eprint.iacr.org/2022/1155
4. P.-A. Fouque, P. Kirchner, T. Pornin and Y. Yu, *BAT: Small and Fast KEM over NTRU Lattices*, https://eprint.iacr.org/2022/031
5. T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte and Z. Zhang, *Falcon*, Technical report, National Institute of Standards and Technology, 2019,

https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions
6. *Post-Quantum Cryptography*, National Institute of Standard and Technology, https://www.nist.gov/pqcrypto
7. T. Pornin, *Improved Key Pair Generation for Falcon, BAT and Hawk*, https://eprint.iacr.org/2023/090
8. T. Pornin and T. Prest, *More Efficient Algorithms for the NTRU Key Generation Using the Field Norm*, Public Key Cryptography - PKC 2019, Lecture Notes in Computer Science, vol. 11443, pp. 504-533, 2019.