Improved Key Pair Generation for Falcon, BAT and Hawk

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

26 February, 2023

Abstract. In this short note, we describe a few implementation techniques that allow performing key pair generation 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 faster than previously published implementations, especially when running on small embedded systems, and uses less RAM.

1 Introduction

We consider here three lattice-based cryptographic schemes:

- The Falcon signature scheme[6] is a lattice-based signature scheme that has been selected for standardization as part of NIST's Post-Quantum Cryptography project. It is based on an NTRU lattice: for a given degree $n = 2^\ell$, the private key is a pair of polynomials f and g, with small integer coefficients and taken modulo $X^n + 1$, completed with another pair of such polynomials F and G, again with small integer coefficients, such that the *NTRU equation* is fulfilled:
$$fG - gF = q$$
with q being a small, fixed integer (in the case of Falcon, $q = 12289$). Key pair generation consists in sampling f and g with an apppropriate distribution on coefficients, then solving the NTRU equation to find matching F and G; this last part is expensive, and where most of the cost of key pair generation is spent. Nominally supported degrees are $n = 512$ and $n = 1024$ (for $\ell = 9$ and $\ell = 10$, respectively), but smaller versions (meant mostly for test purposes) are possible.
- The Hawk signature scheme[3] is based on a different (presumed) hard problem, but uses lattices similar to those used by Falcon. In Hawk, f and g use a different sampling distribution and generally have a smaller norm, and the target integer for the NTRU equation is $q = 1$. Degree is $n = 256$, 512 or 1024 (the degree 256 variant is the "challenge" version, expected to offer a security level of roughly 2^{64}).
- The BAT key encapsulation mechanism[4] also uses a complete NTRU lattice. In BAT, the f and g polynomials have an even smaller average norm than in Hawk; the target integer for the NTRU equation is $q = 128$, 257 or 769, depending on the degree, which is $n = 256$, 512 or 1024. BAT key pairs also include an extra polynomial w, which is used in decapsulation; as will be explained later on, extra constraints apply on w, which indirectly make the key pair generation more expensive.

For all three algorithms, the key pair generation can be performed with tolerable efficiency using the techniques described in [8], which we summarize here. The process can be viewed recursively:

- If f and g are polynomials modulo $X+1$, then they are integers, and finding corresponding F and G is done with a variant of the extended Euclidean algorithm.
- If f and g are polynomials modulo X^n+1, with $n = 2^\ell \geq 2$, then the problem can be reduced to degree $n/2$ by noticing that for each polynomial f, a dual polynomial $f^\diamond$ can be defined by negating the odd-degree coefficients, and then all odd-degree coefficients of $ff^\diamond$ are zero. We thus apply the following steps:
  1. Compute f' and g', polynomials modulo $X^{n/2}+1$, such that $f'(X^2) = ff^\diamond$ and $g'(X^2) = gg^\diamond$.
  2. Invoke the NTRU process solving recursively on (f', g'), yielding F' and G' (modulo $X^{n/2}+1$) such that $f'G' - g'F' = q$.
  3. Compute an *unreduced* solution $(F, G) = (F'(X^2)g^\diamond, G'(X^2)f^\diamond)$.
  4. Reduce (F, G) relatively to (f, g) with Babai's round-off algorithm[2] so that the largest coefficient of (F, G) is roughly the same size as the largest coefficient of (f, g).

The salient implementation issues with this procedure are the following:

- While each recursive iteration halves the degree of the involved polynomials, it also about doubles the sizes of the coefficients. At the deepest levels of the recursion, integer sizes range in the thousands of bits.
- All the values are secret, and thus require appropriate handling to avoid side-channel leakage. In particular, constant-time discipline mandates that all integer-holding arrays use a predictable size which does not vary with the actual size of the involved integers.
- The reduction of (F, G) against (f, g) involves rounding divisions. In the original presentation of Falcon, this was performed with approximations of the polynomials with floating-point values, and the division was computed in the FFT domain. Use of floating-point is problematic, because hardware implementations of the operations are not necessarily constant-time (in particular for divisions), and small embedded systems typically lack any floating-point hardware with the required precision. In [7], a constant-time emulation of floating-point operations with only integer computations is used, but at a rather large cost, making the key pair generation quite expensive.

In this note, we describe a new implementation of the BAT, Falcon and Hawk key pair generation, which is fully constant-time, and avoids all use of floating-point operations. It is also faster, and uses somewhat less RAM, than previously published implementations. The source code (in C) is made available (with a very permissive license) on:

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

This code is meant to be reusable into various schemes, by simply importing the files. It should support adaptation to other parameter sets, if required by newer lattice-based schemes working over NTRU lattices.

2 Improved Implementation Techniques

We describe here the improvements on the techniques used in previous implementations, in particular those for Falcon[7] and BAT[4]. They can be summarized as follows:

- Polynomial coefficients are interleaved with word granularity.
- Computations modulo small 31-bit primes p_i use Montgomery representation with $R = 2^{32}$ (instead of $R = 2^{31}$).
- Babai's reduction uses fixed-point approximations of values over 64 bits (with 32 fractional bits).
- Tighter bounds on integer ranges reduce the cost of iterations during reduction.
- A non-negligible rejection rate is accepted, with quick detection of reduction failures.
- Intermediate (f, g) values are now saved for most recursion layers instead of being recomputed.
- Operations at recursion depths 0 and 1 use a better ordering which reduces memory usage.
- Generation of (f, g) uses smaller tables and fewer PRNG bits per coefficient, to reduce the PRNG cost.
- An AVX2-optimized version has been written, with AVX2 intrinsics used in about all operations.

Polynomial Memory Layout. Each polynomial within the algorithm consists of m coefficients (with m a power of two), and these coefficients are integers within a given range; each integer uses j 32-bit words[1]. Previous implementations would store the coefficients sequentially, i.e. first the j words for the coefficient of degree 0, then the j words for the coefficient of degree 1, and so on. The new implementation interleaves the coefficient values in RAM: the representation first lists the first word of each coefficient in consecutive memory slots (m words), then the second word of each coefficient (m words), and so on. The total size is unchanged (mj words), but the new layout improves spatial locality. In particular, cross-coefficient operations (such as the NTT) now use values which are consecutive in RAM and can be efficiently loaded up conjointly when working with SIMD instruction sets (e.g. AVX2 opcodes on x86 CPUs).

Montgomery Reduction. Most computations on polynomials use RNS representation of coefficients: each coefficient c is represented by the words $c_i = c \bmod p_i$, for a fixed list of prime moduli p_i. The p_i values are all distinct, slightly lower than 2^{31}, and chosen such that $p_i - 1$ is a multiple of 2048 so that the NTT may be applied on polynomials modulo p_i. Montgomery multiplication is used: for integers x and y modulo p_i, their Montgomery product is $(xy)/R \bmod p_i$, for R a given power of 2 greater than p_i. Previous implementations used $R = 2^{31}$, but the new code uses $R = 2^{32}$: this helps in particular 32-bit implementations (and, indirectly, AVX2 code) because a shift of a 64-bit value by 32 bits has only compile-time costs on a 32-bit architecture (it is just a matter of using the "high" register), contrary to a 31-bit shift, which has non-negligible runtime costs.

[1] Each 32-bit word contains a 31-bit limb; the top bit is left at zero. Use of 31-bit limbs makes it easier to achieve a portable implementation, and has performance benefits, especially on architectures with no intrinsic carry flags such as RISC-V.

Fixed-Point Approximations. When applying Babai's reduction, the following process is applied:

- Scaled approximations of (f, g) are obtained as 64-bit values with a fixed-point convention (half of the bits are fractional, which means that the 64-bit integer v represents the rational number $x = v/2^{32}$). The scaling factor is 2^s for some integer s, i.e. we really extract rounded values of $(f/2^s, g/2^s)$. Value s is secret and measured in constant-time by looking at the top few words of the coefficients. It is also adjusted so that the largest coefficients of the extracted approximation have absolute value close to, but less than $2^{15}/m$; this value was chosen so that computations on coefficients do not overflow the representable fixed-point range, but we still keep as many precision bits as we can.
- The FFT representations of the approximate f and g are computed, then the FFT representations of $f^*/(ff^* + gg^*)$ and $g^*/(ff^* + gg^*)$ are obtained, with f^* being the Hermitian adjoint of f. In order to avoid losing too much precision, division operations must be performed exactly like this: a coefficient of f^* (in FFT representation) is divided by a coefficient of $ff^* + gg^*$ (also in FFT representation). The division is performed with a constant-time bit-by-bit process.
- We then repeatedly reduce F and G by running several iterations of the following loop:
  1. Scaled approximations of (F, G) are extracted into the fixed-point format, again with a constant-time process: we use a scaling factor $s + t$, i.e. we get rounded versions of $(F/2^{s+t}, G/2^{s+t})$. The value t is *public*.
  2. The approximate F and G are converted to FFT, then the coefficients computed above are used to obtain $(Ff^* + Gg*)/(ff^* + gg^*)$ in FFT representation; only multiplications and additions are needed at this point. The resulting polynomial is converted back to normal representation (inverse FFT) then its coefficients are rounded to integers, yielding the polynomial k.
  3. F and G are updated by subtracting $2^t kf$ and $2^t kg$ from them, respectively.

The successive iterations use decreasing values of t; the initial value is set to an experimentally determined optimum: if t is too large, then the first few rounds are "blank" (they overestimate the sizes of F and G, and result in $k = 0$), which decreases performance, but if t is not large enough then the approximations of F and G overflow and the reduction yields a wrong result.

As in the previous implementations, the subtraction of $(2^t kf, 2^t kg)$ from (F, G) uses either plain integers, or an RNS+NTT representation; the latter requires converting F and G between RNS and plain representations repeatedly (we need the plain representations to extract the approximations), but still improves performance when the current degree is large enough.

Tighter Bounds on Ranges. Since a full constant-time discipline must be maintained, all internal arrays for holding polynomial coefficients have a fixed length that depends on the algorithm, degree and recursion depth, but not on the actual polynomial values. Extensive experiments have been performed to obtain the average maximum size of coefficients of f, g, F and G (before and after reduction) at all steps of the algorithm, with standard deviations; then, array sizes have been set so as to accept values up to five standard deviations away from the average. The number of top words to inspect in the reduction, when extracting scaled approximations of f, g, F and G, has been set with a similar process, in contrast to the previous

implementation of Falcon, which systematically used the top 10 words. The number of top words to inspect varies with the algorithm, degree and depth, but is at most 7 in all cases, and usually smaller.

Rejection Rate and Quick Failure Detection. All the process is based on heuristics on coefficient sizes and precision of computations. This process may fail for some pairs (f, g) which should have theoretically been valid. This does not induce any security weakness, as long as any failure entails resampling of a new (f, g) pair; the reduction in security cannot exceed that which is entailed by the reduction in the effective key space, which is of 1 bit if half of potential keys are rejected. We can thus tolerate a non-negligible rejection rate, if that promotes average performance. For instance, increasing the assumption on the reduction efficiency (i.e. by how many bits the t scaling factor is reduced at each iteration) makes the reduction faster, but increases the rejection rate, so there is a trade-off with an optimal value for performance.

The new implementation decreases the cost of rejections by performing a fast check on the current solution: at every layer of the recursion, when the (F, G) values have been obtained for that depth, the solution is checked modulo a small prime p_1. The cost of the check is low, and it almost always detects failed reductions, allowing an early abort of the process in that case.

In our selected parameters for BAT, Falcon and Hawk, the failure rate for solving the NTRU equation is always lower than 36% for $n = 1024$, and lower than 15% for $n = 512$.

Saving Intermediate (f,g). In the recursive process, each layer at depth d receives f and g polynomials with degree less than $2^{\ell-d}$, and, in the descending phase, computes the f and g for the next layer. When the recursive call returns, the f and g polynomials for the current layer must then be used again. The previous implementations of BAT, Falcon and Hawk systematically discarded f and g in order to save space, and recomputed them when needed by starting from the top-level f and g (of degree less than $n = 2^{\ell}$). However, careful analysis of memory allocations showed that such discarding was not needed beyond a certain depth (which depends on the algorithm and the overall degree); the new implementation can save the input intermediate (f, g) during the recursive call, and use the saved values instead of recomputing them.

Improved Memory Usage. The highest RAM usage is in the top two layers of the computation; the original Falcon implementation required $28n$ bytes for these steps (hence 28672 bytes for $n = 1024$). The new implementation lowers that value to $24n$ by more carefully managing the memory buffers. Notably, for a given integer polynomial f in NTT representation (in the usual "bit-reversal" order), the NTT representation of its Hermitian adjoint f^* is easily obtained by using the coefficients in reverse order; thus, when computing $ff^* + gg^*$, there is no need to maintain separate memory buffers for f and f^*.

Additional savings were obtained at the API level: the original Falcon implementation returned f, g and F (and optionally G) in caller-provided arrays, so the actual RAM requirement was $31n$; in the new implementation, F and G are returned inside the temporary area (at a predictable emplacement), so the total RAM requirement is $26n$. For BAT, the extra w

polynomial is also returned in the temporary area; for Hawk, the public key polynomials (q_{00}, q_{01} and q_{11}) are returned as well inside that area.

Lower PRNG Usage. The candidate (f, g) values should be sampled with a given distribution, but do not need a great precision; indeed, each (f, g) pair lives "in isolation" and any known bias in the sampling of a given private key has no incidence on any other key pair. This contrasts with the sampling of the noise used in each signature (for Falcon and Hawk), where all signatures relate to the same signing key, and even small biases in that sampling can *cumulatively* reveal too much about the private key. For key pair generation, we can use a much cruder approximation. In this new implementation, we use cumulative distribution tables with 16-bit scaling, i.e. the values in the table are probabilities multiplied by $2^{16} - 1$, and only 16 random bits are used for each coefficient[2]. This lowers the cost induced by the random source, and allows use of, for instance, SHAKE256, even on small microcontrollers for which SHAKE is quite expensive.

An additional optimization is to use four SHAKE256 instances in parallel, with outputs interleaved with a 64-bit granularity. This setting maps well to x86 implementations with AVX2 intrinsics, that can truly run the four SHAKE instances in parallel. On smaller systems, the SHAKE instances can be done sequentially, reusing the same context space, and simply filling the elements of f and g in a non-sequential order so as to reproduce the same output for the same input seed; thus, this interleaving convention induces only negligible overhead in all cases.

3 Performance

x86. The new implementation performance on x86 is summarized in table 1.

These measurements have all been performed on the same test system, with the same compiler. A single CPU core is used. The machine runs a fairly mundane Linux operating system, is otherwise idle, and TurboBoost has been disabled.

The following remarks can be made:

- Performance varies between the three algorithms, for a given degree, roughly in relation with the standard deviation of the distribution for the f and g polynomials. As the process goes deeper in the recursion, polynomial coefficients grow, but source (f, g) with a lower norm imply smaller integers. For instance, for $n = 1024$, the average size of the resultant of f with $X^n + 1$ (i.e. the value of f at the deepest level of the recursion) is 6276 bits for Falcon, but only 5750 bits for Hawk and 4084 bits for BAT.
- Though the (f, g) polynomials of BAT are the smallest among the three schemes, its performance is roughly similar to that of Falcon, because of the extra private polynomial w: that polynomial can be computed only *after* the NTRU equation has been solved, but there are extra requirements on w (in the notations of [4], the resulting $(G_d, \gamma F_d)$ vector must have a norm lower than a specific bound) and in practice, between 40% and 60% of computed w polynomials fail to meet these requirements, leading to frequent restarts.

[2] On Hawk, we currently use a different mechanism with a binomial distribution; this leverages the fact that Hawk enforces a *minimum* norm for (f, g) instead of a *maximum*. This part is experimental and may change in the near future.

Algorithm	Speed (x86 w/ AVX2) (Mcyc)			RAM usage (bytes)		
	Old	New	Gain	Old	New	Gain
BAT-128-256	11.70	3.48	×3.36	8144	6656	×1.22
BAT-257-512	28.00	11.24	×2.49	17408	13312	×1.31
BAT-769-1024	105.99	44.91	×2.36	34816	26624	×1.31
Falcon-256	10.49	3.67	×2.85	7936	6656	×1.19
Falcon-512	22.19	11.58	×1.92	15872	13312	×1.19
Falcon-1024	65.02	50.26	×1.29	31744	26224	×1.19
Hawk-256	4.14	1.89	×2.19	12288	6656	×1.87
Hawk-512	11.71	7.68	×1.52	24576	13312	×1.87
Hawk-1024	49.66	44.22	×1.12	49152	26224	×1.87

Table 1: Performance of the new implementation of key pair generation on x86 (Intel i5-8259U "Coffee Lake", 2.3 GHz, 64-bit mode). Compiler is Clang-14.0.0; optimization flags: `-O2 -mavx2`. Speed values are expressed in millions of clock cycles; RAM sizes are in bytes. "Old" code is from the previously published implementations for the respective algorithms[3,4,7].

- BAT, as published in [4], was already using a fixed-point approximation, thus avoiding all use of floating-point. However, instead of using a direct division to compute $f^*/(ff^* + gg^*)$, it was computing an inversion on $ff^* + gg^*$ followed by a multiplication with f^*. This was inducing a surprisingly large loss of precision, to the effect that the reduction was failing at a much higher rate (for $n = 1024$, about 70% of potential (f, g) pairs could not be reduced successfully, compared to less than 36% in the new implementation). Most of the speed advantage of the new implementation for BAT comes from this specific improvement.
- The RAM gains for BAT come from the API-level optimizations: the new implementation returns F, G and w within the caller-provided temporary array, while the previous implementation required separate output buffers; in particular, the w polynomial consists of 32-bit integers and thus uses $4n$ bytes by itself.
- Hawk key pair generation, as published in [3], was explicitly optimized for speed, not for low RAM usage. Indeed, our speed gains are somewhat lower (compared to the gains on Falcon) but we improve a lot on RAM usage.
- Costs shown here include the computation of the public key for Hawk, but not for BAT and Falcon (though they include the cost of verifying that the public key would be computable, e.g. that f is invertible modulo $X^n + 1$ and modulo $q = 12289$ for Falcon). Key encoding costs are also not included. Moreover, in the case of Hawk, where the encoded public key size is not fixed, a given implementation *may* try to enforce a maximum encoded size by restarting the key pair generation if the result does not fit; such a strategy would mechanically involve some extra overhead (depending on how stringent the size goal is) which is not measured here.

ARM Cortex M4. Our implementation is portable C (the use of AVX2 intrinsics is optional), and can be compiled and used as is on a large range of platforms, including microcon-

trollers such as the ARM Cortex M4. We measured our new code on an STM32F4 "discovery" board (STM32F407VG-DISC1); this is the same board as the one which was used for the Falcon implementation discussed in [7]. Table 2 shows the measured performance.

Algorithm	Speed (ARM Cortex M4) (Mcyc)		
	Old	New	Gain
BAT-128-256	-	23.16	-
BAT-257-512	-	69.19	-
BAT-769-1024	-	233.66	-
Falcon-256	-	24.60	-
Falcon-512	171.29	73.84	$\times 2.32$
Falcon-1024	513.95	287.94	$\times 1.78$
Hawk-256	-	18.66	-
Hawk-512	-	47.92	-
Hawk-1024	-	226.35	-

Table 2: Performance of the new implementation of key pair generation on ARM Cortex M4. Compiler is GCC-10.3.1; optimization flags: `-O2 -mcpu=cortex-m4`. Speed values are expressed in millions of clock cycles. "Old" code is from [7].

The following notes apply:

- Plain C is used for all the code, except for SHAKE: the inner Keccak-f permutation uses an assembly implementation.
- BAT and Falcon use here a random source based on ChaCha8[3]. On that particular platform, ChaCha8 is about ten times faster than SHAKE256. This reduces the overall key pair generation cost by about 6%. For Hawk, we stick to SHAKE256 so as to support a compact private key storage format, in which only the seed used to generate (f, g) is stored, instead of f and g themselves. ChaCha8 can be considered as adequately secure[1].
- Averages were obtained over 1000 key pairs (for each algorithm and each degree).
- Like in the previously published speed benchmarks for Falcon on ARM Cortex M4[7], we use the microcontroller at a 168 MHz clock rate. Instruction and data caches are enabled, since the Flash chip cannot fully serve instructions at that rate to the CPU. Cache misses can induce a few extra wait cycles. Experiments show that for the key pair generation process described here, the cycle counts are increased by only about 0.7% compared with the more common cache-less 24 MHz configuration use in some benchmark frameworks such as pqm4[5] (for some specific pieces of code, the difference can be more dramatic, e.g. a 10% overhead is observed for SHAKE).

[3] The implementation API can use an arbitrary random source, provided as a callback function pointer with an opaque context; we provide both SHAKE and ChaCha8, but anything else could be used.

– We use the optimization flags "`-O2`", like in [7]; the resulting code footprint for the library is about 64 kBytes[4]. Since microcontrollers are often constrained in firmware size, it is common to compile with size optimization instead ("`-Os`" with GCC). In our case, that would result in a binary about 15% smaller, but also 30% slower.
– The main improvement source on the ARM Cortex M4 is the use of fixed-point, which supports vastly faster operations than the constant-time floating-point emulation that was used in [7]. For instance, additions and multiplications of 64-bit fixed-point values can be done in 2 and 4 clock cycles, respectively, on an ARM Cortex M4, while the corresponding floating-point operations used 99 and 64 cycles, respectively, in the old code (not counting the function call overhead and register-saving). A few other improvements apply, such as the use of $R = 2^{32}$ instead of 2^{31} in the Montgomery representation of modular integers.
– RAM usage is the same on the ARM Cortex-M4 as on other platforms; thus, we get the same gains relatively to the previous implementations as on x86. On small microcontrollers, RAM tends to be a very scarce resource, which increases the importance of optimizing that particular aspect of the implementation.

4 Conclusion

Our new implementation, `ntrugen`, is meant to be reused in future cryptographic schemes that leverage NTRU lattices. The main takeaways are the following:

– We removed all use of floating-point operations in the key pair generation.
– We reduced RAM usage, which is important for embedded systems.
– We also obtained substantial speed improvements over previously published implementations of the key pair generation of BAT, Falcon and Hawk, on both small (ARM Cortex M4) and large (x86 with AVX2) systems.

References

1. J.-P. Aumasson, *Too Much Crypto*, https://eprint.iacr.org/2019/1492
2. 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.
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. M. Kannwischer, J. Rijneveld, P. Schwabe and K. Stoffelen, *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*, https://eprint.iacr.org/2019/844

[4] A binary that supports only Falcon, instead of all three of BAT, Falcon and Hawk, would be somewhat smaller, around 57 kBytes.

6. 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
7. T. Pornin, *New Efficient, Constant-Time Implementations of Falcon*, https://eprint.iacr.org/2019/893
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.