# New Efficient, Constant-Time Implementations of Falcon

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

**Abstract.** A new implementation of Falcon is presented. It solves longstanding issues in the existing reference code: the new implementation is constant-time, it does not require floating-point hardware (though it can use such hardware for better performance), it uses less RAM, and achieves much better performance on both large systems (x86 with Skylake cores, POWER8,...) and small microcontrollers (ARM Cortex M4). In particular, signature generation with Falcon-512 takes less than 390k cycles on a Skylake (82k cycles only for verification), and about 19.4 million cycles on an ARM Cortex M4.

## 1 Introduction

Falcon[9] is a post-quantum signature algorithm, submitted to the NIST Post Quantum Cryptography Standardization Process; Falcon has been retained for Round 2. We present here a new implementation of Falcon that solves some issues with the reference code as submitted.

**The new implementation is constant-time.** This is obtained from several characteristics:

- A new Gaussian sampler, designed by Prest, Ricosset and Rossi[10], is used. The new sampler uses rejection sampling with carefully tuned parameters such that observation of the rejection rate yields no satistically exploitable information within the security parameters of Falcon.
- Use of floating-point has been optimized to avoid any non-constant-time operation during the signature process. In particular, divisions and square roots appear only in key generation and cannot yield more than negligible information on the private key.
- All memory accesses are done at non-secret addresses. The memory access pattern does not depend on any secret information, both for key generation and signature generation.

**Floating-point support is not required.** While the new implementation can use floating-point hardware when available, it also includes floating-point emulation code that uses only integer operations. This emulation code is fully constant-time for all values that may appear as part of Falcon, and is portable to all platforms with a C99 compiler and the usual fixed-width integer types (`uint32_t`, `uint64_t`).

**The new implementation is fast and RAM-efficient.** When AVX2 opcodes are supported, the new implementation can use them; with Falcon-512 on an Intel Core i7-6567U at 3.3 GHz, the number of signatures generated per second is over 9000. On the other hand,

a variant with reduced RAM usage has been implemented; it can compute Falcon-512 signatures within less than 40 kB of RAM for temporary values, and negligible stack space usage, while still achieving more than 4000 signatures per second.

On ARM Cortex M4 CPUs, dedicated inline assembly routines have been added to improve performance; the variant with reduced RAM usage can compute a Falcon-512 signature in 41.1 million cycles, a figure which can be brought down to 19.4 million cycles if an "expanded private key" can be used (the expanded key is computed from the normal private key in about 16.2 million cycles, and uses 57.3 kB of RAM).

The new implementation is open source (MIT license) and available on the Falcon Web site:

`https://falcon-sign.info/`

The following sections describe various facets of this implementation.

## 2 Falcon Overview

Falcon is based on NTRU lattices. Operations involve polynomials in $\mathbb{Q}[X]$, taken modulo $\phi = X^n + 1$, where $n$ is a power of two ($n = 512$ for Falcon-512, 1024 for Falcon-1024). Public keys, private keys and signatures use polynomials whose coefficients are integers (i.e. part of $\mathbb{Z}[X]$); however, some intermediate values are polynomials whose coefficients are not integers.

Coefficients for private key elements are small integers (typically within the −127 to +127 range). The private key consists in four such polynomials called $f$, $g$, $F$ and $G$, which fulfil the NTRU equation:

$$fG - gF = q \mod \phi$$

where $q = 12289 = 3 \times 2^{12} + 1$ (a fixed prime integer). The public key consists in a polynomial $h = \mathbb{Z}[X]$ whose coefficients are between 0 and $q - 1$, and such that:

$$fh = g \mod \phi \mod q$$

A signature value, finally, is a polynomial with small integer coefficients, but these coefficients are slightly larger than those of keys (they can go up to about ±1080).

A polynomial $f \in \mathbb{Q}[X]/\phi$ stands for an $n \times n$ matrix such that row $i$ consists in the coefficients of $x^i f \mod \phi$. Addition and multiplication of such matrices then naturally map to addition and multiplication of the corresponding polynomials, taken modulo $\phi$. Thus, the private key is actually a short basis for a $2n \times 2n$ lattice:

$$B = \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$$

while the public key is another basis for the same lattice, but with larger vectors:

$$P = \left[\begin{array}{c|c} -h & I_n \\ \hline qI_n & O_n \end{array}\right]$$

where $I_n$ and $O_n$ are the identity and zero $n \times n$ matrices, respectively.

For a message $m$, the signature process goes thus:

– The signer generates a new random nonce $r$.
– The concatenation of the nonce $r$ and message $m$ is hashed into a polynomial $c \in \mathbb{Z}[X]/\phi$ with coefficients in the 0 to $q-1$ range.
– The signer uses his knowledge of the private key to generate two small polynomials $s_1$ and $s_2$ such that:

$$s_1 + s_2 h = c \mod \phi \mod q$$

The signature itself is the $s_2$ polynomial, along with the nonce $r$.

Signature verification consists in recomputing $c$ from the nonce $r$ and message $m$, then recomputing $s_1 = c - s_2 h \mod \phi \mod q$, and finally verifying that the aggregate vector $(s_1, s_2)$ (of dimension $2n$) indeed has a low enough norm.

The verification can be done entirely with computations over integers modulo $q$, and since $2n$ divides $q-1$, NTT can be used to speed multiplications up. This makes Falcon signature verification very efficient, both in terms of speed and of RAM usage.

Signature generation, on the other hand, uses a complicated process called Fourier sampling, which splits polynomials in a way similar to the Fast Fourier Transform; and, indeed, it is best implemented by using polynomials in FFT representation.

In FFT representation, a polynomial $f \in \mathbb{Q}[X]$ is replaced with the vector of *n complex* values $f(\zeta_j)$ where $\zeta_j$ are the primitive $2n$-th roots of unity in $\mathbb{C}$. In practice, since $f$ has real coefficients, $f(\overline{x}) = \overline{f(x)}$ for any $x \in \mathbb{C}$; since $\zeta_j$ values are pairwise conjugates of each other, the FFT representation is redundant and half the values can be omitted. Thus, the actual FFT representation of a polynomial $f$ consists in $n/2$ complex numbers, i.e. $n$ real (floating-point) numbers.

Fourier sampling is a recursive mechanism, starting with matrix $H = BB^*$ (where $B^*$ is the Hermitian adjoint of $B$) and a target vector $t = (t_0, t_1)$ where $t_0$ and $t_1$ are polynomials modulo $\phi$. Each step involves the following:

1. Consider $H$ to be a $2 \times 2$ matrix whose elements are $n \times n$ matrices.
2. Decompose $H$ into:

$$H = LDL^*$$

where $L$ is "lower triangular" (the two diagonal elements are $I_n$ and the upper-right element is $O_n$), and $D$ is "diagonal" (lower-left and upper-right elements are $O_n$). This works because $H$ is auto-adjoint ($H^* = H$).
3. Split $D_{00}$ (the upper-left coefficient of $D$) into even and odd-numbered coefficients, resulting into half-sized polynomials (i.e. $n/2 \times n/2$ matrices). Target $t_1$ is similarly split.
4. Perform Fourier sampling recursively at half dimension, with a $n \times n$ matrix built out of the two split results of $D_{00}$, and with as target vector the two polynomials obtained from the split of $t_1$.
5. Recursive Fourier sampling invocation returns a "short" vector $z_1$, in two halves of dimension $n/2$, which we merge back into a polynomial of dimension $n$ with the inverse transform of the "split" described above.
6. Compute $t'_0 = t_0 + (t_1 - z_1)L$.
7. Split $D_{11}$ and $t'_0$ and perform a second recursive invocation of Fourier sampling, which returns $z_0$.
8. Merge the two halves of $z_0$, and return $(z_0, z_1)$.

At the lowest level of the recursion, when polynomial degree is 1 and splitting is no longer possible, two small integers values are sampled along discrete Gaussian distributions. The centers of these distributions are given by the input target at that stage, and are not (necessarily) integers; moreover, the standard deviations of these distributions come from the diagonal matrix $D$ at this stage.

Take note of the following salient points:

- Since $n$ is divided by 2 at each recursion depth, maximum depth is $\log_2 n$ (base-2 logarithm). There are $n$ leaves, i.e. there will be $n$ invocations of the Gaussian sampler.
- The input matrices $H$ and LDL decompositions at each stage depend only on the private key. They may be precomputed. The various $L$ matrices, and the standard deviations, constitute together what the Falcon specification calls the "Falcon tree" or "LDL tree". In our new implementation, we denote "expanded key" the combination of the Falcon tree, and the $B$ matrix in FFT representation.
- If the Falcon tree is precomputed, then it contains one matrix $L$ of size $n \times n$, two matrices of size $n/2 \times n/2$, and so on. Total storage size is proportional to $n \log_2 n$, since each matrix $L$ of size $n \times n$ is really a polynomial in $\mathbb{Q}[X]/(X^n + 1)$, which takes (in FFT representation) $n$ floating-point elements.
- If the Falcon tree is *not* precomputed, then the $L$ and $D$ matrices along the current path in the recursion tree must still be retained, even temporarily, for a size proportional to $n$.
- While the standard deviations for the sampler can be precomputed, the centers depend on the target; thus, the actual distributions cannot be precomputed as tables. Instead, the Gaussian sampler uses rejection sampling with a bimodal Gaussian obtained from two half-Gaussians centered on 0 and 1. Rejection sampling involves computing exponentials.
- Computation of the Falcon tree (matrices $L$) involves divisions; and the tree leaves are normalized with a square root operation. However, the rest of the Fourier sampling, including the FFT and inverse FFT operations, requires only additions, subtractions, and multiplications.

# 3 Floating-Point Operations

## 3.1 IEEE-754

IEEE-754[6] defines formats and rounding rules for floating-point numbers. Falcon only uses the "binary64" type, colloquially known as "double precision". Each such value is a 64-bit word, which splits into three parts; we number bits from 0 (rightmost, least significant) to 63 (leftmost, most significant):

- Bit 63 is the *sign bit*: its value $s$ is 0 for a positive number, 1 for a negative number.
- Bits 52 to 62 constitute the *exponent*. The value $e$ ranges from 0 to 2047.
- Bits 0 to 51 are the *mantissa* $m$ which ranges from 0 to $2^{52} - 1$.

When $e \neq 0$ and $e \neq 2047$, this is a *normal value* and represents real number:

$$x = (-1)^s 2^{e-1075}(2^{52} + m)$$

Equivalently, the mantissa $m$ can be thought of as the fractional bits of a value between 1 and 2, its integral part being always equal to 1 and omitted from the encoding; that value is then scaled up or down by multiplying with $2^{e-1023}$.

Special values occur with $e = 0$ or $e = 2047$:

- When $e = 0$ and $m = 0$, the value is a zero. There are two zeros: positive zero (with $s = 0$) and negative zero (with $s = 1$). The distinction is meant to retain the sign of a product that went "too low": a vanishingly small negative result will be rounded to $-0.0$, the sign bit in a sense remembering that the value was negative before rounding. The positive zero ($+0.0$) is the default value (e.g. all additions and subtractions that yield a zero produce $+0.0$, except when adding $-0.0$ to itself, in which case the result is $-0.0$).
- When $e = 0$ and $m \neq 0$, the value is *denormalized* (this is often described as a "subnormal" value). In that case, the value is:

$$x' = (-1)^s 2^{-1074} m$$

  Compared with the formula for normal values, the scaling exponent is one more than expected, but the $2^{52}$ element has disappeared (in the equivalent description, $m$ is now the fractional bits of a number between 0 and 1, its integral part being equal to 0).
- When $e = 2047$ and $m = 0$, the value is an *infinite*. Infinite values have their own rules for additions and subtractions.
- When $e = 2047$ and $m \neq 0$, the value is a *NaN* ("not a number"). This is a placeholder for impossible values. For instance, dividing one by zero yields an infinite, but dividing zero by zero produces a NaN. Any operation where at least one operand is a NaN returns a NaN. IEEE-754 distinguishes between "quiet NaNs" and "signalling NaNs", depending on how the machine should react when encountering the values.

**Falcon does not use infinites, NaNs or subnormals.** The only special values that may appear in Falcon are zeros. This is important: it means that we do not have to care about the behaviour of the FPU (or emulation thereof) on such special values, in particular when assessing constant-timeness.

In IEEE-754, any value is exact: it stands for the exact represented real number. *Rounding* is applied when the mathematical result of an operation cannot be represented exactly. There are several rounding rules, but the default one is "round to nearest - even", which means that:

- Rounding goes to the nearest representable value.
- In case of a tie (the mathematical result is exactly mid-way between two successive representable values), rounding goes to the value whose lowest mantissa bit (bit 0) is 0 (that is, an even mantissa is preferred in case of a tie).

For instance, suppose that $x = 4856738204785675$ and $y = 4150461049955318$; these two numbers are exactly representable in IEEE-754 binary64 values. However, their sum $x+y$, which happens to be equal to $2^{53} + 1$, is not exactly representable, and stands exactly in the middle between the two closest representable values, which are $2^{53}$ and $2^{53} + 2$. In that situation, the value $2^{53}$ is preferred, because its representation uses an even mantissa (specifically, $(s, e, m) = (0, 1076, 0)$ for $2^{53}$, while $(s, e, m) = (0, 1076, 1)$ for $2^{53} + 2$).

In Falcon, "round to nearest - even" is always used.

## 3.2 Hardware FPU

A number of architectures offer more or less complete implementations of IEEE-754 operations. The C standard[3] does not *mandate* strict conformance to IEEE-754, but supports it, and moreover defines the exact rules to do so; a C compiler that supports IEEE-754 will define the `__STDC_IED_559__` macro to signal this property, and will follow precise rules, most notably that the C type `double` corresponds to the IEEE-754 binary64 type (and thus has size 64 bits).

Using exactly the precision of the binary64 type is important for the implementation of the round-to-nearest-integer function. The previous reference implementation of Falcon used the `llrint()` function; since this function implementation is not necessarily constant-time, our new implementation instead uses the following method for a floating-point value $x$:

- If $x > 2^{52}$ then it already represents exactly an integer, and we can use a simple type cast, which will use the conversion to integer opcode.
- If $0 \leq x \leq 2^{52}$, then $(2^{52} + x) - 2^{52}$ computes exactly the required rounding, provided that the initial addition uses the binary64 precision and round-to-nearest-even rounding.
- If $x < 0$ then there are two similar sub-cases, depending on whether $x < -2^{52}$ or not.

For constant-time processing, all sub-cases must be performed, and the correct result filtered with bitwise operations. This yields the following code:

```
int64_t sx, tx, rp, rn, m;
uint32_t ub;

sx = (int64_t)(x - 1.0);
tx = (int64_t)x;
rp = (int64_t)(x + 4503599627370496.0) - 4503599627370496;
rn = (int64_t)(x - 4503599627370496.0) + 4503599627370496;

/*
 * If tx >= 2^52 or tx < -2^52, then result is tx.
 * Otherwise, if sx >= 0, then result is rp.
 * Otherwise, result is rn. We use the fact that when x is
 * close to 0 (|x| <= 0.25) then both rp and rn are correct;
 * and if x is not close to 0, then trunc(x-1.0) yields the
 * appropriate sign.
 */
m = sx >> 63;
rn &= m;
rp &= ~m;
ub = (uint32_t)((uint64_t)tx >> 52);
m = -(int64_t)((((ub + 1) & 0xFFF) - 2) >> 31);
rp &= m;
rn &= m;
tx &= ~m;
return tx | rn | rp;
```

This process works and is constant-time as long as the additions and subtractions on floating-point values, and conversions from floating-point to integers (which use "truncate towards zero" semantics), are constant-time and use the binary64 precision exactly.

In practice, strict IEEE-754 compliance is not guaranteed. Modern x86 CPU in 64-bit mode use the SSE2 unit for all floating-point operations, which guarantees correct behaviour and rounding as per the specification. On other systems, this is more a hit-and-miss game:

- On x86 in 32-bit mode, floating-point operations may be performed with the "387 FPU" (the floating-point instruction set introduced with the 80387 co-processor). This FPU supports several rounding precisions, and typically defaults, for intermediate computations, to a larger precision (FPU registers have size 80 bits, with 64-bit mantissas)[1].
  In order to ensure rounding at the exact binary64 precision, our implementation sets the appropriate bits in the FPU control word (using inline assembly with the `fstcw` opcode) before performing key pair or signature generation, and sets back the previous precision afterwards.
- On some systems (in particular PowerPC), the hardware FPU provides "fused multiply-add" (FMA) opcodes, that combine a multiplication and an addition in a single operation, the IEEE-754 rounding being applied only once. While FMA opcodes are "correct" from a security point of view, they imply some slightly different results when compared with platforms that do not have or use FMA opcodes, preventing reproducibility of test vectors. To prevent such contractions of expressions, C99 defines a standard `#pragma` directive:

  ```
  #pragma STDC FP_CONTRACT OFF
  ```

  but it appears to be ignored by GCC 6.2.2, for which a GCC-specific directive must be used:

  ```
  #pragma GCC optimize ("fp-contract=off")
  ```

- Some hardware provides IEEE-754 rounding, except for subnormals. This is typically the case of embedded PowerPC hardware, which return zero instead of a subnormal. Exact IEEE-754 rounding can then be achieved only with extra test code or by having the hardware trigger a CPU exception when a subnormal result is obtained, for the operating system to perform corrective actions to yield the proper result. Since Falcon does not use subnormals, this situation does not arise in practice, and the rounding-to-zero of subnormals is perfectly acceptable for our implementation.
- On some platforms, there is no hardware FPU, and the software routines that emulate FP operations, as provided by the compiler, may have improper rounding in some cases[2].

[1] This depends on the operating system conventions; on FreeBSD, binary64 precision is used; on Darwin, the SSE2 unit is used instead of the 387 FPU. The larger precision is the default on Linux and Windows, though.

[2] This is for instance the case with the routines provided by GCC 7.3.0 for the FPU-less ARMv7-M CPU such as the Cortex M3 and M4; while they are correct most of the time, they get the rounding wrong in the last bit in some cases, e.g. when adding 1048576.0000000002 to 9007199254740989: the correctly rounded result is 9007199255789566, but on the ARM with GCC-provided emulation, one gets 9007199255789563.6 instead.

However, the compiler-provided routines are usually not constant-time, and the emulation routines provided in our implementation should be used instead.

There is no C operator for square roots, but most FPUs implement that operation as a dedicated opcode. On many platforms, calling the `sqrt()` library function results in the inlining of that opcode, along with a test on the operand to call the library function in case the operand is not a nonnegative, finite value[3]. Since Falcon never tries to extract the square roots on negative numbers, the library function is never actually called, but the dependency on `libm` is still present. To avoid it, we use dedicated inline assembly and/or compiler intrinsics to invoke the square root opcode and not the library function; this is supported on x86 (32-bit and 64-bit, with GCC, Clang and MSVC), PowerPC (32-bit and 64-bit, both big-endian and little-endian, with GCC, Clang and IBM XLC) and ARM (32-bit "armhf", and 64-bit "aarch64", with GCC and Clang).

**Constant-time status:** using the hardware FPU leads to a constant-time implementation of Falcon only if the hardware offers constant-time opcodes. The core operations that will be typically done by the FPU are the following:

- Additions and subtractions.
- Multiplications.
- Divisions.
- Square roots.
- Conversions between integers and floating-point values (as per C rules, these apply truncation toward zero).

In Falcon, all operands and results are normal values or zeros; no subnormals, infinites or NaNs are used. Notably, divisors are never zero, square root operands are always positive, and when converting a floating-point to an integer, the result is always representable in the range of the target integer type. Moreover, divisions and square roots occur only during computations of the Falcon tree.

Following analysis in [1], on recent x86 CPU, additions, subtractions, multiplications and conversions, as used in Falcon, are fully constant-time. Divisions and square roots are *mostly* constant-time, except that some "shortcuts" are applied when a divisor is a power of two, or a square root operand is a power of four: in these cases, the operation is slightly faster.

Crucially, such a situation may occur only within the Falcon tree building, and when it occurs in a key, it will occur always at the same place within the tree. If observed by outsiders, they may learn only a static piece of information on the private key, unimpacted by the random nonces and messages; thus, the leak, if present, cannot be exploited into a larger key recovery or forgery attack. Experimentally, over 470,000 Falcon-512 key pairs, and 160,000 Falcon-1024 key pairs:

- A division by a power of two happens conjointly with a square root of a power of four, for a leaf of the Falcon tree. The square root operand is then $2^{14}$ (no other value is possible). No case was observed otherwise, neither higher in the tree, or during key pair generation.
- About 0.18% of Falcon-512 keys presented such a case, and 0.17% of Falcon-1024 keys.

[3] The FP facilities can be configured to trigger exceptions or do some other error reporting, which is implemented by the library functions but not by the hardware opcode; the call is then meant to manage errors.

We can thus estimate that the side-channel leak due to some cases of divisions and square roots completing slightly faster than usual may happen in about one every $2^9$ key pairs, and will yield only a static information about which of the $n = 2^9$ or $2^{10}$ tree leaves yielded an input of $2^{14}$ to the square root, i.e. an information worth at most 18 or 19 bits of secret key material.

The leak is hard to detect (the timing difference is less than 10 clock cycles). Even if it can be detected in practice, there is no known method by which such an information may be exploited in an attack. If such a method is ever found, the reduction in security level will not exceed these 18 or 19 bits, and will impact only one in every 500 keys. We thus consider that this leak is negligible; this is the sense in which we can claim Falcon to be "fully constant-time".

There is a relatively simple method by which even this negligible leak can be removed. It suffices to compute the Falcon tree after key pair generation, not necessarily retaining the $L$ matrices, and then to verify that none of the tree leaves involves one of the blacklisted operands (square root of exactly $2^{14}$). If such a case is reached, the newly generated key pair can be discarded, and a new one generated in its place. The computational overhead would be moderate (tree generation cost is between 1% and 10% of key pair generation cost, depending on architecture, and only 0.18% of keys would have to be regenerated). This method would simply remove all doubts about this slight theoretical leak. We have not yet implemented it, for three reasons:

- Computing the Falcon tree, even without retaining the $L$ matrices, uses more temporary RAM than the rest of key pair generation; for Falcon-1024, this would require about 49.1 kB, while key pair generation itself fits in 28.7 kB. It is conceivable that this RAM usage may be reduced; more optimization work is needed in that area.
- One of the NIST test vectors includes an affected key. Rejecting it would modify that test vector.
- On a more general basis, which values are "forbidden" depends on the underlying implementation of floating-point operations. Our own software emulation (described in the next section), for instance, does not have any forbidden value. But for reproducibility of test vectors, all implementations must use the exact same rules.

In a future version of our Falcon implementation, when exact rules for forbidden values are properly specified, the key rejection process described above will be implemented. We still stress that, from our analysis, the leak appears to be negligible anyway.

## 3.3 Software Emulation

Within the new Falcon implementation, all floating-point values are represented by a type called `fpr`, and all operations use functions such as `fpr_add()`. When using the hardware FPU, the `fpr` type is defined to be a `struct` with a single field of type `double`, and the operation functions are all declared "`static inline`" and simply apply the relevant C operators. The use of a wrapping `struct` is meant to detect cases where raw C operators would be used on the values, instead of the wrapping functions; all calls being inlined, the compiler can bypass the wrapping, which thus induces no runtime cost.

When software emulation is enabled with the compile-time macro `FALCON_FPEMU`, the `fpr` type is an alias for `uint64_t`, and the elementary operations use functions implemented with only integer operations, and fully constant-time. These functions do not handle special

cases (infinites, NaNs and subnormals), but provide correctly rounded results as per IEEE-754 rules for all normal values and zeros.

The assertion that a given piece of code is "constant-time" depends on some assumptions about the implementation of the elementary operations by the hardware, and about the way the compiler translates source code into CPU opcodes. In particular, our new implementation of Falcon, when using `FALCON_FPEMU`, is constant-time, under the following assumptions:

- Integer multiplications of 32-bit values with a 64-bit result execute in a time independent of the values of the operands.
- Left and right shifts of 32-bit values execute in a time independent of both the value which is to be shifted, and the shift count (which is between 0 and 31).

While most modern CPU offer constant-time multiplications opcodes, some relatively widespread platforms, especially in embedded microcontrollers, have variable-time multiplications[8]. In the ARM Cortex line, the Cortex M4 offers constant-time $32 \times 32 \rightarrow 64$ multiplications, but the Cortex M3 does not. The primary embedded target for our new implementation is the Cortex M4.

Shifts with a potentially secret count are used especially in the implementation of floating-point additions and subtractions: each operand has its own exponent, and at least one must be shifted by an amount that depends on the difference between these exponents, for the two mantissas to be properly aligned for the addition or subtraction operation.

Most CPU, even small ones, have constant-time shifts, because they internally use a barrel shifter. This would not have been the case on many CPUs from the previous century; the most recent notable exception is the Pentium IV (NetBurst core), which lacked a barrel shifter and whose shifts would take a variable amount of time, depending on the shift count. In Intel lines of CPU, the NetBurst core was replaced in 2006 with the line of cores called (confusingly) "Core" and derived from the P6 microarchitecture used by Intel *before* NetBurst (the NetBurst was thus a dead-end in the Intel CPU evolutionary tree).

It shall be noted that we are here talking about shifts of 32-bit values, not 64-bit. On a generic 32-bit architecture, 64-bit integers are represented over two registers, and if the shift count may range up to 63, then conditional execution may be used to handle the two subcases, depending on whether the shift count is lower than 32 or not. Indeed, on 32-bit PowerPC, the following C function:

```
uint64_t lsh(uint64_t x, unsigned n)
{
    return x << n;
}
```

will be compiled (by GCC 6.2.0, optimization flags `-O2`) into:

```
lsh:
        stwu 1,-16(1)
        addic. 9,5,-32
        blt 0,.L2
        slw 3,4,9
        li 4,0
```

```
        addi 1,1,16
        blr
        .p2align 4,,15
.L2:
        srwi 9,4,1
        subfic 10,5,31
        srw 9,9,10
        slw 3,3,5
        or 3,9,3
        slw 4,4,5
        addi 1,1,16
        blr
```

We notice that the second opcode (`addic.`) subtracts 32 from the shift count, updating the flags; then, the third opcode (`blt`) jumps to label `.L2` if the result of the subtraction is negative, i.e. if the shift count is in the 0 to 31 range. This non-constant-time behaviour, observable by outsiders through timing or cache attacks, thus leaks the bit 5 of the shift count.

To avoid such issues, we use the following function instead:

```
uint64_t fpr_ulsh(uint64_t x, unsigned n)
{
    x ^= (x ^ (x << 32)) & -(uint64_t)(n >> 5);
    return x << (n & 31);
}
```

which avoids the issue: the bit 5 of the shift count is handled specially, and the actual shift is only with a count in the 0 to 31 range. GCC still generates a sequence of opcodes that includes a conditional jump, but this is because GCC apparently lacks the ability to notice that `n&31` can never be outside of the 0 to 31 range; the conditional jump will be *always* taken, and thus will no longer leak information on the shift count.

On some architectures (notably x86 and ARM), opcodes are provided, that can do more than a shift of a 32-bit value by a count in the strict 0 to 31 range; for these systems, compilers can and do generate branchless, constant-time code for shifts of 64-bit values. But our integer-only implementation is meant to provide constant-time processing in a portable way, and thus needs to use the "safe" function `fpr_ulsh()`.

Our software emulation code implements all needed operations in full constant-time processing. This includes additions, subtractions and multiplications, but also conversions to integers (both with truncating toward 0 and rounding to the nearest integer) and from integers (which requires finding the top non-zero bit of the operand in a constant-time way), divisions, and extraction of square roots. The compiled code is typically slower than the hardware FPU opcodes by a factor of 20 or so. However, it is highly portable and reproducible, and safe based on fewer assumptions on the hardware than the implementation using the FPU.

## 3.4 ARM Cortex M4

The ARM Cortex M4 implements the ARMv7-M architecture. That core provides two (optional) components: a small DSP, that can make SIMD operations (integer-only computations, on limited-size values), and a hardware FPU. That hardware FPU supports only the

binary32 type, not binary64 (its abilities with regards to binary64 values are limited to converting to binary32, with an unavoidable loss of precision, and back). Therefore, that hardware FPU is unsuitable to Falcon, and we do not use it.

If compiling with the native `double` type, then the compiler will emit calls to software emulation routines. These routines are somewhat more efficient than our code, but they are not constant-time. Notably, they implement shortcuts and return much quicker when the operands allow for it (e.g. multiplying by zero, or adding together two values such that one has a much smaller exponent than the other). For safe operations, the `FPEMU` code must be used.

Our implementation provides assembly implementations for the core floating-point operations: conversions from integer to floating-point (with the `fpr_scaled()` function), additions (`fpr_add()`), multiplications (`fpr_mul()`), divisions (`fpr_div()`) and square roots (`fpr_sqrt()`)[4]. The result is more than twice faster than the generic `FPEMU` code, and close to the speed of the non-constant-time routines provided by the compiler for normal, non-shortcut operands. Among the CPU features that are not easily leveraged by generic C code, one may note the following:

- Flags for carries and borrows are available, helping with operations over integers larger than 64 bits.
- Left and right shifts with the `lsl` and `lsr` opcodes, using a register for the shift count, actually work on the *eight* least significant bits of that register; i.e. the shift count can meaningfully range up to 255, instead of 31.
- The `clz` opcode counts the number of leading zeros in a 32-bit value: it locates the top non-zero bit in a constant-time and very fast way (1 cycle). This is very convenient for normalizing results of operations, especially additions.
- Opcodes for inserting (`bfi`), extracting (`ubfx`) or clearing (`bfc`) arbitrary groups of consecutive bits in a 32-bit word allow for efficient extraction and reassembly of the components (sign bit, exponent, mantissa) of a floating-point value.

The assembly functions are provided as "naked functions", i.e. C functions whose complete body is inline assembly, as supported by GCC and Clang. There is thus no separate assembly source file, and the compilation process is unchanged. The functions honour the calling conventions of the ABI: all modified registers (except the scratch registers `r0` to `r3`) are saved, and the 64-bit alignment of the stack pointer is maintained.

Since the DSP and single-precision FPU are not used, our code will actually run properly on an ARM Cortex M3 as well. However, the $32 \times 32 \rightarrow 64$ multiplication opcodes of the M3 are *not* constant-time[5] (and are somewhat slower than on the M4), meaning that our Falcon implementation, running on the M3, would not be safe against side-channel attacks (although how the information leak could be turned into an actual attack is not known at the time). This remark applies to both our inline assembly code and the generic `FPEMU`.

[4] Subtraction is done by inversing the sign bit of the second operand and then using addition, and thus does not warrant its own assembly implementation.

# 4 Constant-Time Processing

Apart from the constant-time implementation of floating-point operations, as described in the previous section, a number of modifications have been done in order to ensure that Falcon operations are fully constant-time.

**Key pair generation.** Key pair generation involves generating the $f$ and $g$ polynomials with a simple Gaussian distribution, then solving the NTRU equation to compute $F$ and $G$.

The generation of $f$ and $g$ uses a simple table-based process: for each coefficient to generate:

- A random bit $s$ is generated, as well as a random value $v_0$. If $v_0$ is lower than a given constant $c_0$, then the coefficient will be zero.
- A random 63-bit value $v_1$ is generated, and a table of constants is used: the index $k$ of the first value $c_k$ which is not greater than $v_1$ is obtained (the table elements are indexed starting at 1, and are in decreasing order, the last one being 0, thus guaranteeing a unique solution).
- If $v_0 < c_0$, then the new polynomial coefficient is 0. Otherwise, its value is $(-1)^s k$.

All operations are performed systematically, i.e. the test on $v_0$ is done in a constant-time way, and the lookup of $v_1$ in the table is always done, regardless of the result of the test on $v_0$. Moreover, the table lookup is done by reading *all* elements of the table, with no early exit.

When solving the NTRU equation, polynomials of various degrees, whose coefficients are big integers, are computed. Most of the reference code was already naturally constant-time, since it was using RNS (residue number system) for big integers, working on an expected length of the integers rather than their true length. Two major changes were made to achieve constant-timeness:

- At the deepest level of the formally recursive process, an extended GCD between two big integers is performed. We imported the constant-time binary GCD implementation from BearSSL[2].
- When reducing a pair of polynomials $(F', G')$ with regards to another pair $(f', g')$ that have smaller coefficients, using Babai's nearest plane algorithm, the implementation repeatedly uses floating-point approximations of the polynomial coefficents, in order to obtain small reduction coefficients that will shave off 25 to 30 bits from the coefficients of $(F', G')$. The reference implementation was measuring the actual size of the coefficients, in order to have the best possible approximation, and stopped the loop when no further reduction was achieved. Our new implementation, instead, uses a statistical estimate of the current sizes of the coefficients to obtain a floating-point approximation, reading a full 310-bit range (ten 31-bit words) for each integer; moreover, the loop conservatively assumes that the reduction was of only 25 bits each time, and iterations continue until the estimated size of $(f', g')$ is reached.

These changes induce a few extra costs, and our key generation process is bout 10% to 15% slower than the previous reference code. Memory usage is unchanged, and everything is fully constant-time[5].

[5] Babai's reduction involves some floating-point divisions; theoretically, one such division may use a divisor which is a power of two, that a hardware FPU could leverage as a shortcut. Such an event

**Gaussian sampling.** The new Gaussian sampler[10] is similar to the one from the reference code, with the following differences:

- The half-Gaussian distribution, which is the basis for the rejection sampling, uses a table which is accessed in a constant-time way, similarly to the generation of coefficients in key pair generation (described above).
- Rejection sampling implies computing $e^{-x}$ for values $x$ such that $0 \leq x \leq \log 2$. While the reference code was using an implementation of the exponential lifted from the well-known `fdlibm`[4], our new code uses the polynomial approximation from FACCT[11]. The polynomial avoids any use of division. Moreover, when using `FPEMU`, the polynomial is evaluated to the required precision using plain integers only, without exponents, leading to constant-time and much faster code.

While the exponential evaluation is faster, the table lookup is slower than in the previous reference code, since it now needs to read all table values systematically. Moreover, the new parameterization from [10] implies a lower acceptance rate (about 73%, down from the previous 93%) and thus more required iterations per signature.

When using AVX2 opcodes explicitly (with the `FALCON_AVX2` compile-time option), the table lookup is more efficient, since AVX2 can read 32 bytes in one go and perform four 64-bit comparisons in parallel. Moreover, AVX2 opcodes are used for a much faster implementation of the ChaCha20 stream cipher that generates the pseudorandom values on which the sampling operates. Finally, when AVX2 is used, the polynomial approximation for $e^{-x}$ is performed with an alternate circuit that computes *more* multiplications and additions, but with a lower circuit depth: this improves performance because CPUs with AVX2 support tend to have large abilities at parallel evaluation. Indeed, on a Skylake core, a floating-point addition has a latency of 4 cycles, but a reciprocal throughout of only 0.5 cycles, meaning that eight additions can be issued before the result of the first one is available. Evaluation latency turns out to be a more important factor for performance than the raw operation count.

**Hash-to-point and signature encoding.** Nominally, "constant-time" is about not leaking information on the *private key* through timing-based side channels. However, the signature value, and the signed message, are not considered secret. Indeed, if the message $m$ is secret but is of low enough entropy to allow an exhaustive search attack, then the signature value and the public key can necessarily be used as a stop condition for such an attack: the attacker "tries" various potential values for $m$, and knows he reached the right one when the signature verification succeeds. This property is common to all signature algorithms, not just Falcon.

Our implementation primarily focuses on the normal situation of non-secret signatures and public keys. However, we also provide some support for the unusal scenario where public keys and signatures are secret, and messages may be subject to an exhaustive search. In the reference implementation of Falcon, two characteristics failed to provide side-channel protection in these cases:

---

appears to be exceedingly rare; we did not observe it a single time over more than 500,000 key pair generations. Even if it ever happened and could be observed – a difficult feat for the attacker, since it is by nature a one-time event, never repeated, and the timing difference is less than 10 cycles – there is no currently known way of exploiting that small bit of information.

– The hash-to-point mechanism repeatedly obtains 16-bit values from SHAKE256 (with input $r \parallel m$); if the value is between 0 and 61444, it is reduced modulo 12289; otherwise, the 16-bit value is rejected and a new one obtained. This process ensures that hashed messages will be uniformly distributed, but it is inherently non-constant-time and can serve as a stop condition for an exhaustive search attack on $m$.
– Signatures are encoded with a variable-size compression scheme. Each coefficient of $s_2$ yields a number of bits which depends on the coefficient value.

To solve the first item, we provide a constant-time variant of the hash-to-point mechanism. That variant first obtains sufficiently many 16-bit values to almost always have enough (the number of extra values is adjusted so that the probability of not getting enough is lower than $2^{-256}$, i.e. negligible). Then, out-of-range values are marked as "holes" in the list, and squeezed out in a number of passes. For a Falcon degree $n$, $\log_2 n$ passes are needed, and each pass uses only constant-time conditional swaps. Some overhead is unduced by the over-extraction of values from SHAKE256 and the conditional swaps; this overhead is relatively small with regards to the signature generation process, but non-negligible for the faster signature verification process. We thus made this process optional.

For the second item, we defined a second signature encoding process where each coefficient of $s_2$ is encoded over a fixed number of bits. Extensive experiments have measured a maximum range for these coefficients of ±1077; thus, 12 bits ought to be enough with overwhelming probability. The signature generation process furthermore includes an explicit check: if any coefficient is not in the -2047 to +2047 range, the signature is discarded, and a new one is obtained. This never happens in practice.

# 5 Portability and Performance

Our new implementation has been successfully tested on several architectures. A "successful" test implies exact reproduction of 100 test vectors on pseudorandom messages and seeds for each of Falcon-512 and Falcon-1024. Each test vector involves generating a public/private key pair, and then signing a message. A specific seed is used for each test. Test systems were the following:

– x86, 64-bit mode (amd64 on Intel i7-6567U):
  • Linux (Ubuntu 18.04), GCC 7.4.0
  • Linux (Ubuntu 18.04), Clang 6.0.0
  • Windows (10), MSVC 2015
– x86, 32-bit mode (i386 on Intel i7-6567U):
  • Linux (Ubuntu 18.04), GCC 7.4.0
  • Linux (Ubuntu 18.04), Clang 6.0.0
  • Windows (10), MSVC 2015
– PowerPC, 64-bit, little-endian (ppc64le on POWER8):
  • Linux (Ubuntu 16.10), GCC 6.2.0
  • Linux (Ubuntu 16.10), Clang 3.8.1
  • Linux (Ubuntu 16.10), XLC 13.1.5
– PowerPC, 64-bit, big-endian (ppc64be on POWER8):
  • Linux (Ubuntu 16.10), GCC 6.2.0

    - Linux (Ubuntu 16.10), Clang 3.8.1
  - PowerPC, 32-bit, big-endian (ppc on POWER8; kernel is ppc64be):
    - Linux (Ubuntu 16.10), GCC 6.2.0
    - Linux (Ubuntu 16.10), Clang 3.8.1
  - ARM, 64-bit, little-endian (aarch64 on Cortex-A53):
    - Linux (Ubuntu 17.04), GCC 6.3.0
    - Linux (Ubuntu 17.04), Clang 3.9.1
  - ARM, 32-bit, little-endian (armhf on Cortex-A53):
    - Linux (Raspbian 8), GCC 4.9.2
    - Linux (Raspbian 8), Clang 3.5.0

All these systems have a usable hardware FPU. Both `FPNATIVE` and `FPEMU` engines work on each machine. On the x86 systems, all tests were performed both with and without usage of AVX2 intrinsics.

## 5.1 RAM Usage

In the external API of our new implementation, temporary buffers are provided by the caller, so that they may be allocated where appropriate for the usage context. Temporary buffer sizes are the following (all sizes are expressed in bytes):

| degree | make pub | verify | keygen | sign (tree) | expand key | sign (dyn) |
|---|---|---|---|---|---|---|
| 512 | 3073 | 4097 | 15879 | 25607 | 26631 | 39943 |
| 1024 | 6145 | 8193 | 31751 | 51207 | 53255 | 79879 |

Sizes of public, private and expanded private keys are:

| degree | public key | private key | expanded key |
|---|---|---|---|
| 512 | 897 | 1281 | 57344 |
| 1024 | 1793 | 2305 | 122880 |

Note that decoding a private key from its normal format (of length 1281 bytes for Falcon-512, containing $f$, $g$ and $F$) into an expanded key requires 57344 bytes for storing the expanded key (Falcon tree) *and* 26631 bytes of temporary storage for computing the expanded key. That temporary storage may be reused for signature generation operations (which use 25607 bytes, when an expanded Falcon-512 key is available).

Signature sizes depend on whether compression is used or not. Not using compression yields larger signatures, but with a fixed size; this is the format that should be used when signed messages are low-entropy secret values. When using compression, signatures have a maximum theoretical size (largest size needed for any vector whose norm is short enough to possibly be a valid signature), but also a measured average and standard deviation, shown below:

| degree | uncompressed | compressed max | compressed avg (std. dev) |
|---|---|---|---|
| 512 | 809 | 752 | 651.59 (2.55) |
| 1024 | 1577 | 1462 | 1261.06 (3.57) |

The averages have been measured over 10,000 signatures (100 random key pairs, 100 signatures per key pair). Take care that the signature format now includes the 40-byte nonce; the Falcon specification lists some measured sizes *without* the nonce.

The nonce size has been chosen so that within the usage limits specified by the NIST Post-Quantum Cryptography project ($2^{64}$ maximum number of signatures per private key), risks of nonce reuse are at most $2^{-192}$. Falcon can use much shorter nonces, as long as some mechanism is in place that prevents nonce reuse; "40 bytes" is the safe length for random generation without maintaining any mutable state.

## 5.2 Intel i7-6567U

The test system is a MacBook Pro system, running Linux (in a virtual machine) in 64-bit mode (amd64). CPU is an Intel i7-6567U (Skylake core), nominally running at 3.3 GHz; however, this CPU uses TurboBoost that raises the frequency up to 3.6 GHz under load. The values below have been measured by running the tested operation sufficiently many times that the total time (measured with `clock()`) is at least 2 seconds. Time per operation is given in microseconds ($\mu$s) and clock cycles, the latter assuming a 3.6 GHz clock rate (when performing the test, the machine is mostly idle with no potentially expensive background task such as music streaming, or Slack). Precision can be estimated to about 1% or 2%.

Three configurations are measured:

- `fpemu`: generic integer emulation of floating-point operations, no use of the native FPU or SSE2 hardware. Compilation with Clang 6.0.0, optimization flags: `-O3`
- `fpnative`: use of the native FP hardware (SSE2 unit) through the `double` type; no compiler intrinsics. Compilation with Clang 6.0.0, optimization flags: `-O3`
- `avx2`: use of the native FP hardware (SSE2 unit) through the `double` type, enhanced with explicit use of the AVX2 and FMA intrinsics. The compiler also automatically uses AVX2 opcodes for some operations (in particular automatic vectorization of the signature verification process, which is expressed in integer C code only). Compilation with Clang 6.0.0, optimization flags: `-O3 -march=native -mcpu=native`

The measured speeds for Falcon-512 are the following (the operations marked with "ct" mean that constant-time hash-to-point and the fixed-size signature size are used, to ensure privacy of the signed message itself; but all other operations are still constant-time with regards to the private key):

| **Falcon-512** | **fpemu** | | **fpnative** | | **avx2** | |
|---|---|---|---|---|---|---|
| **operation** | $\mu$s | cycles | $\mu$s | cycles | $\mu$s | cycles |
| key pair generation | 17,060.00 | 61,416,000 | 7,510.00 | 27,036,000 | 7,350.00 | 26,460,000 |
| sign (dynamic) | 5,098.28 | 18,353,808 | 323.32 | 1,163,952 | 244.76 | 881,136 |
| sign (dynamic, ct) | 5,103.22 | 18,371,592 | 342.73 | 1,233,828 | 261.82 | 942,552 |
| expand private key | 2,105.37 | 7,579,332 | 105.72 | 380,592 | 98.31 | 353,916 |
| sign (tree) | 2,172.58 | 7,821,288 | 172.97 | 622,692 | 108.27 | 389,772 |
| sign (tree, ct) | 2,198.05 | 7,912,980 | 188.41 | 678,276 | 126.41 | 455,076 |
| verify | 27.25 | 98,100 | 26.61 | 95,796 | 22.67 | 81,612 |
| verify (ct) | 44.03 | 158,508 | 43.89 | 158,004 | 40.16 | 144,576 |

For Falcon-1024, the following timings have been measured, on the same CPU and with the same three configurations:

| **Falcon-1024** | **fpemu** | | **fpnative** | | **avx2** | |
|---|---|---|---|---|---|---|
| **operation** | $\mu$s | cycles | $\mu$s | cycles | $\mu$s | cycles |
| key pair generation | 48,870.00 | 175,932,000 | 22,400.00 | 80,640,000 | 21,940.00 | 78,984,000 |
| sign (dynamic) | 11,093.83 | 39,937,788 | 664.29 | 2,391,444 | 495.63 | 1,784,268 |
| sign (dynamic, ct) | 11,068.68 | 39,847,248 | 699.78 | 2,519,208 | 528.05 | 1,900,980 |
| expand private key | 4,651.31 | 16,744,716 | 215.40 | 775,440 | 199.18 | 717,048 |
| sign (tree) | 4,730.34 | 17,029,224 | 353.74 | 1,273,464 | 219.46 | 790,056 |
| sign (tree, ct) | 4,765.49 | 17,155,764 | 386.78 | 1,392,408 | 253.80 | 913,680 |
| verify | 55.76 | 200,736 | 56.24 | 202,464 | 43.94 | 158,184 |
| verify (ct) | 88.80 | 319,680 | 88.71 | 319,356 | 77.77 | 279,972 |

## 5.3 ARM Cortex M4

For benchmarking our code on the ARM Cortex M4, we used a STM32F4 "discovery" board (STM32F407VG-DISC1). The board provides an ARM Cortex M4 core that can run at up to 168 MHz, along with 192 kB of RAM (in two separate chunks of 128 and 64 kB, respectively), and 1 MB of ROM (Flash).

This is the same board as the one used by the pqm4 project[7]. We actually provided a draft version of our implementation to pqm4, hence their measures are very similar to ours. There are slight differences which are mostly attributable to cache effects.

Indeed, while the M4 core does not have caches by itself, it obtains instructions and data from the Flash and RAM. While RAM accesses reliably complete within one cycle, even at 168 MHz, Flash accesses are slower. The board thus provides an instruction cache of 1 kB and a data cache of 128 bytes for all accesses to Flash, along with a prefetcher. At 168 MHz, each access to Flash has an additional latency of 5 cycles (i.e. a read access initiated at cycle 1 provides the data at cycle 7 instead of cycle 2 for an access to RAM). The caches work by lines of 16 bytes (128 bits), and the prefetcher tries to issue a read for the next line sufficiently in advance, based on some limited branch prediction. These schemes are mostly effective, but do not fully compensate for the 5 extra wait states. In particular, in our SHAKE256 implementation, the main loop has about 2 kB in size and thus exceeds the I-cache size; we thus expect that performance is limited by the read bandwidth for instructions. Each read for 16 bytes takes 5 cycles, but delivers only 4 instructions (for the general case of 32-bit instructions), thereby incurring a 1-cycle stall.

To avoid these issues, pqm4 makes all measures at the reduced frequency of 24 MHz, where Flash access has no wait state (i.e. it completes within one cycle). We chose the other option of running at 168 MHz, with caches and prefetcher enabled.

For our key pair generation measures, we enabled the use of ChaCha20 as pseudorandom generator: this prevents reproducibility of test vectors, but is safe and saves about 40 million clock cycles per key pair.

We here list times in milliseconds. Clock cycles have been measured with the integrated cycle counter (`DWT_CYCCNT`), with excellent precision. All measures are averages over 200 key pairs or signatures. Compiler is GCC 6.3.1 with optimization `-O2 -mcpu=cortex-m4` and our software emulation with inline ARM assembly is used.

| Operation | Falcon-512 | | Falcon-1024 | |
|---|---|---|---|---|
| | ms | cycles | ms | cycles |
| key pair generation | 1,020.20 | 171,394,151 | 3,061.38 | 514,312,539 |
| sign (dynamic) | 246.32 | 41,381,272 | 537.89 | 90,365,144 |
| expand private key | 96.39 | 16,194,107 | 213.11 | 35,802,561 |
| sign (tree) | 116.69 | 19,604,182 | 250.58 | 42,096,639 |
| verify | 3.04 | 510,751 | 6.19 | 1,039,169 |

When using the constant-time hash-to-point and signature encoding (in the unusual case where signed data is secret and low-entropy, and the signature and public keys are kept secret), add 328,625 cycles (1.95 milliseconds) to all Falcon-512 signature generations and verifications (615,145 cycles or 3.66 milliseconds for Falcon-1024).

# 6 Conclusion

We presented new, optimized implementations of Falcon, that fix some longstanding issues with the reference code: Falcon can now be said to be constant-time, and to be able to run on small embedded hardware that does not offer a hardware FPU. We moreover managed to keep all test vectors reliably reproducible, thereby showing that all our configurations compute the same things. This is especially important for lattice-based cryptographic systems, because security relies on a precise distribution of random samples, which cannot be easily tested for correctness.

When a hardware FPU is present and usable, Falcon performance is quite satisfying; it is not completely on par with the best elliptic curve signature schemes, but still in the same order of magnitude. Falcon-512 is also vastly faster than RSA-2048 on the same hardware, while providing post-quantum security, and relatively compact signatures and public keys.

On small embedded microcontrollers, Falcon performance suffers from the complexity of floating-point operations, but still achieves tolerable performance, e.g. sub-second signature generation time even at moderate operating frequencies (e.g. 64 MHz). Moreover, Falcon signature *verification* is very fast, making it suitable for the common situation of a small embedded system that verifies a signature on its firmware at boot time.

# References


1. *Towards Verified, Constant-time Floating Point Operations*, M. Andrysco, A. Nötzli, F. Brown, R. Jhala and D. Stefan, Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, `https://doi.org/10.1145/3243734.3243766`
2. BearSSL, `https://www.bearssl.org/`
3. *ISO/IEC 9899:1999 – Programming languages – C*, 1999.
4. `fdlibm`, `https://www.netlib.org/fdlibm/`
5. *A performance study of X25519 on Cortex-M3 and M4*, W. de Groot, `https://research.tue.nl/en/studentTheses/a-performance-study-of-x25519-on-cortex-m3-and-m4`
6. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985, `https://doi.org/10.1109%2FIEEESTD.1985.82928`
7. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4*, M. J. Kannwischer, J. Rijneveld, P. Schwabe and K. Stoffelen, `https://eprint.iacr.org/2019/844`

8. *Constant-Time Mul*, T. Pornin, `https://www.bearssl.org/ctmul.html`
9. *Falcon*, T. Prest, P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Ricosset, G. Seiler, W. Whyte and Z. Zhang, Technical report, National Institute of Standards and Technology, 2017, `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/round-2-submissions`
10. *Simple, Fast and Constant-Time Gaussian Sampling over the Integers for Falcon*, T. Prest, T. Ricosset and M. Rossi, to be presented at the Second NIST PQC workshop (August 2019).
11. *FACCT: FAst, Compact, and Constant-Time Discrete Gaussian Sampler over Integers*, R. K. Zhao, R. Steinfeld and A. Sakzad, `https://eprint.iacr.org/2018/1234`