Falcon on ARM Cortex-M4: an Update

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

29 January, 2025

Abstract. This note reports new implementation results for the FALCON signature algorithm on an ARM Cortex-M4 microcontroller. Compared with our previous implementation (in 2019), runtime cost has been about halved.

1 Results

The FALCON signature scheme[5] is one of the signature algorithms selected by the NIST Post-Quantum Cryptography standardization project[12]. The scheme is known for its implementation complexity, especially its use of floating-point operations, making it challenging to implement safely and efficiently on small microcontrollers. In 2019, we reported some performance figures for a constant-time implementation of FALCON on various platforms, including a microcontroller using an ARM Cortex-M4 CPU[13]. We reworked this M4 implementation with more optimizations; the result is available (and dedicated to the public domain) at:

https://github.com/pornin/c-fn-dsa

In a nutshell, this new implementation is about twice faster than the 2019 code. Table 1 lists the achieved performance.

Operation	FALCON-512		FALCON-1024	
	speed (cycles)	RAM (bytes)	speed (cycles)	RAM (bytes)
keygen	71943764	13343 + 928	284031714	26655 + 928
sign	22008433	39967 + 2048	47778012	79903 + 2160
verify (orig)	255306	2079 + 872	518594	4127 + 872
verify (BUFF)	358517	2079 + 872	724171	4127 + 872

Table 1: Average speed (clock cycles) and RAM usage (bytes) of FALCON on ARM Cortex-M4. RAM consists of a configurable transient work area, and some additional stack use.

Hardware. The used hardware is an STM32F407G-DISC1 microcontroller board, using a STM32F407VGT6 microcontroller. All speed measurements have been taken by using the board at 24 MHz speed, as is customary in the literature[11]; at that relatively low frequency

(the board can be used at up to 168 MHz), caches can be disabled because RAM and ROM (Flash) accesses normally complete with minimal latency.

Most of the code is written in C, with some assembly routines. The compiler is GCC, version 13.2.1. Relevant code generation and optimization flags are:

```
-O2 -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16
```

Note that the used M4 CPU includes the optional DSP and floating-point units; the latter is not directly usable for FALCON signature generation, since it supports only 32-bit precision (IEEE 754 type "binary32"); however, the floating-point registers are a convenient fast-access storage area, which the C compiler uses, and which we also leverage in our assembly routines.

Key pair generation, signature generation and signature verification in FALCON all have varying execution times (in signature generation and verification, this is due to the use of rejection sampling, both for converting the message to sign into a pseudorandom polynomial, and for the Gaussian sampling). The reported values are averages over 15000 key pairs and signatures for each degree.

Scheme. The implemented scheme is the "padded" variant of FALCON, in which signatures are padded to a fixed size (of 666 or 1280 bytes, respectively) and the signing process is restarted (with a random nonce) if the signature cannot be encoded within that target size.

In the original scheme (tagged "orig" in table 1), a random 40-byte nonce and the message are hashed together (with SHAKE256) in order to produce a random polynomial; we also implemented a modified scheme (tagged "BUFF") in which an arbitrary context string and the hash of the public key are also injected in the SHAKE256 input, in the same way as is done in ML-DSA[9]. Including this hash provides some extra security properties beyond strict signature unforgeability[4] and helps with proving security in multi-user settings; this *may* be part of what NIST will standardize into the upcoming FN-DSA standard. The BUFF variant implies hashing the public key (with SHAKE256); the performance difference observed in table 1 is exactly the cost of this extra hashing operation. The cost technically applies to signature generation as well, but it does not make much of a difference in practice, since signature generation is a lot more expensive (the costs listed for signature generation in table 1 are for the BUFF variant). The "BUFF" variant also regenerates the 40-byte nonce when the signature process fails and restarts, as suggested in [6]; this helps with provability, and has no detectable impact on performance, especially since signature restarts are rare (about 1/230000 probability with FALCON-512).

Signature generation uses an internal PRNG. In the submitted FALCON implementation, this PRNG was using a ChaCha20 variant; in this new implementation, SHAKE256 is used. As will be explained in section 2, for this platform, the cost of the PRNG is only a small part of the overall signature generation code.

Security Stance. Our implementation should be always *correct* and *safe*. In particular, all operations should be *constant-time*, a traditional name that really means that it should not be feasible to infer information on secret data from timing-based measurements. This should hold not only on the ARM Cortex-M4 CPU, but also on most other CPUs compatible with the ARMv7M architecture (i.e. roughly speaking, larger ARM CPUs in 32-bit mode). Such a property can only be achieved on a best-effort basis, because it relies on individual instructions to be constant-time by themselves, and on any compilation layer not to turn apparently

constant-time code into non-constant-time constructions, in particular by recognizing that a variable value is a Boolean in disguise and using it in conditional jumps to shortcut some computations. Since some platforms use hidden JIT compilation techniques (e.g. NVIDIA's "project Denver" and subsequent designs), whether constant-time properties are maintained can be impossible to verify in practice.

In our implementation, this means the following:

- No memory access is performed at a secret-dependent address.
- No conditional jump uses a secret condition.
- At the assembly level, we do not use the conditional execution opcode (`it`). This opcode appears to be constant-time in the M4 itself (i.e. the wrapped instructions take the same amount of time regardless of whether they are executed or not), but it could plausibly be subject to non-constant-time optimizations in larger CPUs (e.g. handling as a conditional jump with speculative execution based on a prediction on the condition value).
- Pure computational instructions, including multiplications with a 32-bit or 64-bit output size, are expected to be constant-time, to the exclusion of division opcodes (`udiv`, `sdiv`), which we do not use on secret data.

An *important caveat* is that we do not try to achieve any kind of protection against power analysis or fault attacks. Such attacks can be (potentially) mitigated only through a careful analysis taking into account the physical characteristics of the target device, something which cannot be done in all generality in portable software. The only side-channel attacks that we attempt to thwart in our implementation are timing attacks.

Usability Stance. Our implementation is meant to be production-level; in other words, it should be usable in various applications right away. It is *optimized* for an ARM Cortex-M4, but it should run "as is" on larger ARMv7 platforms, possibly using more complex operating systems than what are typically encountered on microcontrollers. This has a few consequences:

- The code is reentrant and thread-safe. No modifiable global data is used.
- The register `r9` is not used by any of our assembly routines.
- Some effort has been made toward code compacity, e.g. by not fully unrolling time-critical loops.

The register `r9` is also called the "static base" in the standard ARM application binary interface (dubbed AAPCS[2]). It is nominally reserved for any purposes (that the target platform must document). In practice, that register may be used either to access thread-local storage, or to support position-independent code. In the latter case, a given platform *might* implement asynchronous interruptions as a signal delivery, and the signal handler may then rely on the value of `r9` to be correct. The operating system would not necessarily take care, or even be able, to set the register to the correct value before entering the signal handler; it has been reported that very early versions of iOS were operating in that way. If some routine temporarily modifies `r9` and a signal is delivered just at that moment, then execution may derail, possibly in ways that can be leveraged by attackers to gain control of the system. This failure mode would be hard to detect in unit tests. Fortunately, most operating systems, and in particular bare-metal microcontrollers (with no operating system at all), do not have such strict

requirements on `r9` and routines may use the register as a simple callee-saved register (e.g. like `r8`). Neverthelss, to avoid the improbable but scary failure scenario described above, we shun any use of `r9`.

Loop Unrolling and Code Size. Loop unrolling is a classic tool to decrease runtime cost, since it lowers the cost of loop management, both directly (no need to update a loop counter, no jump back to the loop start) and indirectly (without a loop counter, an extra register is free, and some memory access offsets might be hardcodable). However, it also increases the code footprint, which can be an issue on small embedded systems where code space (ROM/Flash) may be of limited size. There thus are trade-offs between speed and code size. In our implementation, we voluntarily avoided some unrolling, especially in the NTT and SHAKE implementations. For instance, if using only signature verification, then the code footprint totals to 11840 bytes. This includes 4096 bytes for the NTT tables (that could be truncated to 2048 bytes if restricting the implementation to FALCON-512) and 4184 bytes for the SHAKE core. The well-known XKCP implementation of SHA3/SHAKE[16] is faster than our code, but also partially unrolled (four rounds), and its code footprint is 10632 bytes.

The speed we report here appears to be "the record" for signature verification, among other reported numbers. It is faster than a recently published result from Choi, Yoon and Seo[3]. However, our gains are mostly in the "auxiliary" parts (e.g. public key and signature decoding), while our NTT and inverse NTT are slower than theirs, because our code is not unrolled (but a lot more compact). It would be fairly easy to reduce the cost of our FALCON-512 signature verification by about 50k cycles, by simply replacing the SHAKE implementation with the XKCP one[1], and the NTT and inverse NTT with the code from [3]. The result would be formally faster, though possibly less usable in practice due to the inflated code size.

Baseline. To estimate how fast (or slow) our implementation is, it may be relevant to compare its performance with that of the other lattice-based signature scheme selected by NIST, the Dilithium scheme (standardized as ML-DSA[9]). An assembly-optimized implementation is reported in [1]; for the lowest security level ("Dilithium2"), key pair generation, signature generation and signature verification cost 1.596, 4.093 and 1.572 million cycles, respectively. On the same hardware, with our implementation, FALCON-512 has a much more expensive key pair generation (about 45 times), more expensive signature generation (5.4 times), but faster verification (4 to 6 times faster, depending on whether the BUFF or original variant is used). This highlights that FALCON is challenging but not impossible to use for generating keys and signatures on microcontrollers, but is also substantially more efficient for signature verification (and with a lower RAM usage). Since FALCON keys and signatures are also much shorter than ML-DSA's, we envision that FALCON, once standardized (under the name FN-DSA), will be a preferred choice for the common scenario of an embedded device verifying a signature on its upgradable firmware at boot time.

[1] This would require some slight changes in the caller sites, because we access output SHAKE bytes directly from the SHAKE context, which relies on that context using little-endian without the even/odd-indexed split which is commonly used in 32-bit SHAKE implementations such as XKCP.

2 Optimization Techniques

2.1 Cost Statistics

In order to optimize code, it is useful to know where the CPU time is spent. Figure 1 shows the relative cost of hashing (Keccak-f function, which is the core of SHAKE) in key pair generation and in verification (considering here the BUFF variant). Figure 2 considers signature generation, with a breakdown for individual floating-point operations. Both figures are for FALCON-512, but the relative costs for FALCON-1024 are very similar. The values are also for a small signed message; for a very long message, the cost of hashing necessarily dominates both signature generation and verification.

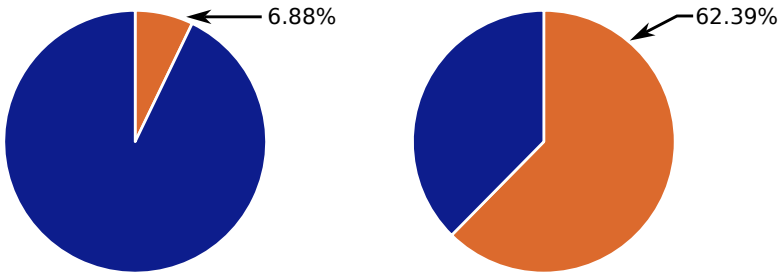

Fig. 1: Relative cost of SHAKE in key pair generation (left) and signature verification (right). This uses the BUFF variant of the verification.

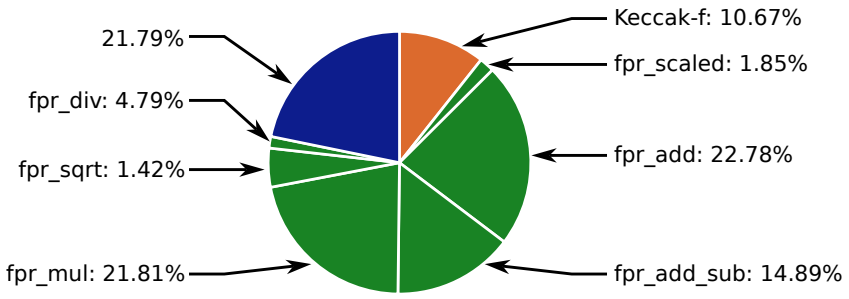

Fig. 2: Relative costs of SHAKE and floating-point operations in signature generation.

SHAKE is used in key pair generation for producing candidate pairs of small polynomials (f, g). In signature generation, SHAKE is mostly used as a source of pseudorandom bytes for the Gaussian sampling, which is performed $2n$ times (with n being the degree, 512 or 1024). We see in figures 1 and 2 that for signature generation, the time spent in performing SHAKE computations is slightly over 10% of the total, and less than 7% for key pair generation. Therefore, optimizing SHAKE will not matter much for the overall costs of these two operations. However, it is the dominant cost in signature verification.

2.2 Memory Accesses

Efficiency of RAM accesses on the M4 depends on a number of parameters; we list here some useful remarks.

Alignment. The M4 can in general be considered to tolerate unaligned memory accesses. If an unaligned 32-bit value must be read from RAM or written to RAM, the CPU will break it down into two or three accesses, but the overall overhead will be lower than if the same access split was performed in software. It should be noted that the opcodes that can read or write several 32-bit words in a single instruction (`ldrd`, `strd`, `ldm`, `stm`) do not tolerate unaligned accesses, and trigger a CPU exception if the address is not a multiple of 4. In our implementation, we ensure that all temporary values are suitably aligned (in fact, 256-bit alignment of the temporary buffer is enforced); unaligned accesses may happen mostly when serializing or deserializing data from bytes (e.g. hashing the message, or decoding the public key).

Store Buffer and Pipelining. Memory store operations in the M4 (`str`) use an intermediate store buffer which is used to perform the actual store operation asynchronously. This means that a lone store can usually be considered to execute in one cycle, but that stalling can happen if the instructions immediately before or after the store also access memory. Successive `str` operations pipeline, e.g. a sequence of three stores will execute in three cycles. Note that, in the absence of stalls, two successive `str` opcodes use two cycles while a combined `strd` opcode uses three cycles; a consequence is that in tight loops, `strd` should be avoided (but its use should otherwise be recommended in that it promotes code compacity).

The M4 can pipeline memory accesses more generally, between loads and stores, provided that they don't have extra dependencies (specifically, pipelining is prevented if a load operation modifies a register which is used to compute the address of the next load or store instruction). A sequence of n `ldr` opcodes (with no dependencies) will execute in $n + 1$ cycles. The double-load instruction (`ldrd`) executes in three cycles but does not pipeline; thus, a sequence of two `ldrd` uses 6 cycles while an equivalent sequence of four `ldr` will use only 5 cycles.

This leads to the following guidelines for optimizing memory accesses:

– In general, memory loads should be grouped together in sequences of `ldr` instructions (to leverage pipelining). A "load multiple" opcode (`ldm`) can offer the same performance as a pipelined sequence of loads with a more compact code, but only if the target addresses are successive. `ldrd` should be used only when exactly two values must be read from two successive memory slots, and there is no immediately preceding or following memory access with which these loads could be pipelined.

– Memory stores can be grouped together to the extent that it is compatible with pipelining, but may also be spread out and interleaved with computational instructions so as to benefit from the store buffer. *n* `str` opcodes, properly pipelined or spread out, will execute in *n* cycles, which is faster than what a "store multiple" opcode (`stm` or `strd`) can achieve.

Contention with Instruction Fetching. The M4 must perform memory accesses to fetch instructions to execute. In general, this happens in parallel with explicit memory accesses, but it may happen that the two accesses somehow compete for the same resources, and stalls happen. The exact conditions for this event are not documented. *Heuristically*, this problem can be avoided by ensuring that instructions are 32-bit aligned: individual instructions are encoded over either 16 bits or 32 bits; stalls due to contention with the instruction fetching unit may happen when instructions with a 32-bit encoding are located at an address which is not a multiple of 4. Purely computational routines which work on registers only are not impacted by such stalls, and can mix 16-bit and 32-bit instructions freely (and, in general, 16-bit instructions *should* be used when possible since that promotes code compacity). However, routines that need to perform memory accesses (for instance a Keccak-f implementation for SHAKE) should strive to ensure alignment of all 32-bit instructions, by either not using any 16-bit instruction, or pairing such instructions together. At the assembly level, most 16-bit instructions can be converted to 32-bit instructions by adding a `.w` suffix to the instruction name.

Floating-Point Registers. The M4 includes a floating-point unit, which is not really usable for Falcon since it supports only the "binary32" IEEE 754 type. However, the floating-point registers can be used as a temporary storage area. There are 32 such registers (`s0` to `s31`) and, in the call convention (AAPCS), `s0` to `s15` are caller-saved, i.e. the callee can use them without saving them beforehand. The `vmov` opcode can move a value between a general purpose register and a floating-point register in one cycle, which is faster than stack accesses. There is a `vmov` variant that can copy two values (in two cycles), while using only one 32-bit instruction, which is good for code compacity.

Interconnection Matrix and Caches. It is often said that the M4 does not have caches, and thus look-up tables with secret addresses should not induce information leaks through timing measurements. This is not a correct assertion. The ARM Cortex-M4 is, really, a virtual design for a CPU (usually called an IP, as "intellectual property"), which a hardware vendor will integrate with other elements into an actual physical chip. Typically, that chip will be dubbed a SoC (system-on-chip) or a microntroller, and will include some RAM and ROM/Flash, as well as various I/O pins and controllers for extra hardware, possibly with DMA accesses. The M4 itself does not include caches, but caches may be added by the integrator.

Taking as an example the STM32F407VGT6 microcontroller (which is the one in the test board that we used for our benchmarks), its datasheet includes the diagram which we reproduce on figure 3.

We see that the CPU itself communicates with its RAM, Flash and peripherals through a large interconnection matrix ("Bus matrix-S") which arbitrates between concurrent accesses.

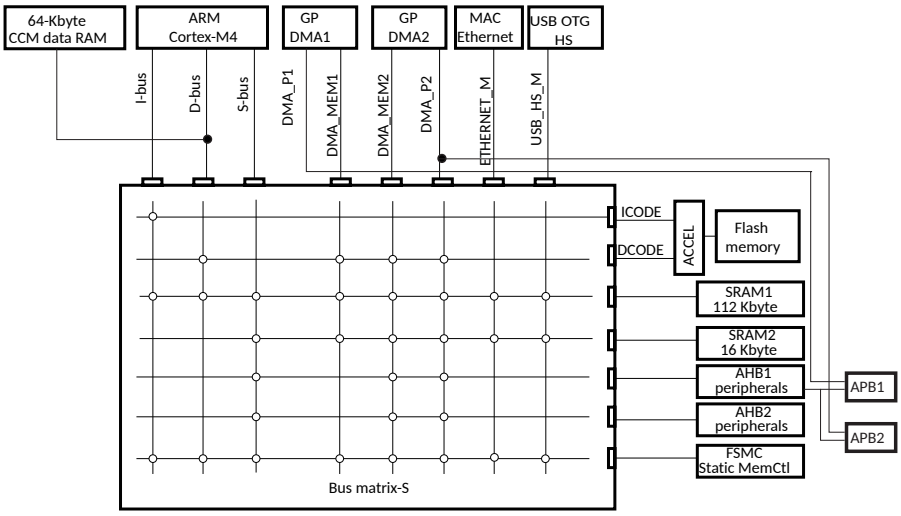

Fig. 3: Interconnection matrix in the STM32F407VGT6 microcontroller.

Arbitration is needed because each connected element (e.g. an SRAM block) cannot perform two accesses simultaneously, and thus there can be accesses occuring at the same time (e.g. from the CPU and from a peripheral performing DMA) that compete for the same resource; in such a case, one of the accesses must be stalled for a cycle. The exact method by which competing accesses are detected, and which access gets stalled in that case, is not fully documented. Many similar interconnection matrices detect *potentially* overlapping accesses by matching a subset of the address bits (which is more efficient than checking *all* bits, and heuristically sufficient on average). This implies that I/O activity, potentially triggered by an adversary at precisely chosen times, may imply DMA activity which may or may not compete with RAM and Flash accesses from the code, thus revealing, through timing differences, parts of the addresses involved in the memory accesses from the CPU. It is not clear how feasible such an attack is on the STM32F407VGT6 microcontroller, but it is plausible enough that a variant of such an attack *may* apply to some similar microcontrollers and SoCs. This is enough to warrant our stance of making no memory access at a secret address whatsoever.

Figure 3 also shows an element called "ACCEL" which is really a pair of caches, which are used to optimize read accesses from the Flash memory (the two separate caches are for instructions and data, respectively). In our benchmarks, we disable these caches, since they are not needed at the relatively low clock frequency of 24 MHz. These caches are however quite useful at higher clock rates; the board can be clocked at up to 168 MHz, and out-of-cache read accesses incur large penalties (5 cycles extra latency) at these rates. Of course, active caches imply the possibility of cache attacks, further highlighting the temerity of using look-up tables with secret addresses.

It should be noted that at high frequencies, the Flash element cannot provide instructions fast enough to feed the CPU; the maximum bandwidth is 128 bits per 5 cycles (at 168 MHz), in that particular microcontroller. If a long sequence of 32-bit instructions is executed and falls outside of the instruction cache (or that cache is disabled), then the CPU will receive only 4 instructions per 5 cycles, implying an up to 25% slowdown. This effect is mitigated by using the instruction cache, but only for routines which fit in that cache; long unrolled sequence can thus have a detrimental effect on performance. Judicious use of 16-bit instructions can help lower the CPU bandwidth requirements, but can be applied only in some situations (in particular, most 16-bit instructions can use only the "low" registers `r0` to `r7`) and can break alignment of other instructions, which can lead to in-CPU contention with the fetch unit, as described previously.

2.3 Key Pair Generation

For key pair generation, we mostly reuse the implementation from ntrugen[15]. That implementation improved on the original FALCON code by using a stricter memory management (yielding a lower RAM usage and better caching of internal values) and replacing all floating-point operations with fixed-point (over 64 bits). In our implementation, we merely added some inline assembly for four operations:

- halving an integer modulo a small prime p;
- Montgomery multiplication modulo a small prime p;
- fixed-point multiplication;
- fixed-point squaring.

The halving operation deserves some explanations because it leverages integer multiplication in a non-obvious way. The inputs are an integer x and a modulus p, with p being a 31-bit prime, and x an unsigned integer in the $[0, p-1]$ range. If x is in register `r0` and p is in register `r1`, then the halving process would naturally use a 4-instruction sequence, which is about what is obtained by letting the C compiler translate the C code:

```
    lsls    r2, r0, #31           @ r2 <- (x << 31)
    and     r2, r1, r2, asr #31   @ r2 <- p & -(x & 1)
    adds    r0, r0, r2            @ add p to x only if x is odd
    lsrs    r0, r0, #1            @ x is even, halve it
```

But we can do better, in three instructions only:

```
    ubfx    r2, r0, #0, #1        @ r2 <- 1 if x is odd, or 0
    umlal   r0, r2, r2, r1        @ add p to x only if x is odd
    lsrs    r0, r0, #1            @ x is even, halve it
```

The key to this sequence is the use of `umlal`, which computes a $32 \times 32 \rightarrow 64$ multiplication, and adds the result to a 64-bit accumulator; we use it here to add either $0 \times p$ or $1 \times p$ to x. Conceptually, a large multiplication followed by a 64-bit addition ought to be vastly more expensive than a logical AND and a 32-bit addition, but in the ARM Cortex-M4 `umlal` executes in one cycle, making it worthwhile to use.

This is in fact a general pattern: the DSP extension provides several opcodes that perform a large multiplication, then followed by one or two additions (in particular `umlal` and `umaal`), which execute in one cycle, i.e. as fast as a mere 32-bit addition. These opcodes are quite versatile and can be used to implement some operations such as shifts faster than the "normal" opcodes, mostly because they can write to two distinct registers in the same cycle.

2.4 Signature Generation

As illustrated in figure 2, more than 2/3 of the cost of signature generation is in the floating-point operations, hence optimization efforts should focus on these operations.

Floating-Point Operations. Floating-point values and operations in Falcon follow IEEE 754[8], with the type "binary64" and the rounding policy "roundTiesToEven". The C language standard allows the C compiler to temporarily extend the precision, and (under some conditions) to contract expressions to leverage extra opcodes such as fused-multiply-add (i.e. computing $xy+z$ in one opcode without intermediate rounding). In order to strictly follow the same computations inside Falcon as on any other architecture, we cannot let the compiler perform such optimizations. Moreover, we also want all operations to be performed in a constant-time way, not leaking any information on the individual values through variations in the execution time; this is typically not ensured by compiler-provided emulation routines on FPU-less hardware. For these reasons, on platforms where there is no suitable hardware for IEEE 754 operations, the Falcon code must store floating-point values in an integer type (`uint64_t`) and emulate the operations with custom routines that take care to operate in a constant-time way.

We optimize six such operations, each with a specific assembly routine:

- `fpr_scaled`: given a 63-bit integer m (signed) and an exponent e (small integer, usually zero in practice), convert $2^e m$ into a floating-point number.
- `fpr_add`: given floating-point numbers x and y, compute $x + y$ or $x - y$. Negation in floating-point values is only flipping the sign bit, hence in our situation we can consider it to be basically free, and thus addition and subtraction have the same cost (and use the same routine).
- `fpr_add_sub`: given floating-point numbers x and y, compute $x+y$ *and* $x-y$. This happens notably in FFT and inverse FFT transforms, and some work can be shared between the two operations[2].
- `fpr_mul`: given floating-point numbers x and y, compute xy.
- `fpr_div`: given floating-point numbers x and y, compute x/y.
- `fpr_sqrt`: given floating-point number x, compute $\sqrt{x}$.

In table 2, we list how many of each operation occur on average in a Falcon-512 signature generation, and the cost (in clock cycles) of each corresponding routine. The number of invocations for `fpr_scaled`, `fpr_add` and `fpr_mul` may vary (due to the use of the rejection method in the Gaussian sampling), but experimentally the standard deviation is low, and the actual number is almost always within 1% of the average.

[2] I *think* that I did not come up with that optimization concept myself; I saw this done somewhere. However, I do not remember where. If you have more information please tell me, so that I can properly credit the original inventor.

Operation	Invocations	Cost
`fpr_scaled`	11992	34
`fpr_add`	62680	80
`fpr_add_sub`	32768	100
`fpr_mul`	106685	45
`fpr_div`	2304	458
`fpr_sqrt`	512	610

Table 2: Floating-point operations used in a FALCON-512 signature (average number of invocations, and individual cost in clock cycles).

The costs listed in table 2 include the execution of all opcodes in the routine, including the final `bx lr` which returns to the caller, but excluding the `bl` opcode that the caller used to invoke the routine. The caller must also set the arguments into the proper registers; that cost is not accounted for in table 2 (and it depends on the call site). Most of our routines follow the proper call convention (AAPCS), in which registers `r4` to `r11` and `s16` to `s31` must be preserved by the callee; the saving costs are included in the costs above. `fpr_add_sub` is special, in that it needs to return *two* 64-bit values, which is not a case covered by the AAPCS, and the standard method (returning a `struct` instance at the C level) would entail using a pointer and extra memory accesses; to avoid that overhead, we invoke `fpr_add_sub` with a dedicated inline assembly routine:

```
/* Inputs are x and y, both floating-point values declared
   with type uint64_t */
register uint32_t x0 __asm__("r0") = (uint32_t)x;
register uint32_t x1 __asm__("r1") = (uint32_t)(x >> 32);
register uint32_t y0 __asm__("r2") = (uint32_t)y;
register uint32_t y1 __asm__("r3") = (uint32_t)(y >> 32);
__asm__(
    "bl     fndsa_fpr_add_sub"
    : "+r" (x0), "+r" (x1), "+r" (y0), "+r" (y1)
    : "r4", "r5", "r6", "r7", "r8",
      "r10", "r11", "r12", "r14", "s15", "cc");
uint64_t a = (uint64_t)x0 | ((uint64_t)x1 << 32);
uint64_t b = (uint64_t)y0 | ((uint64_t)y1 << 32);
```

The main consequence is that `fpr_add_sub` does not save the values of registers `r4` to `r11` and the caller must see to saving them, for some extra overhead which is not accounted for in table 2. That overhead depends on the calling site and what register values the caller wants to preserve across the call. In that sense, the runtime cost of that routine is somewhat underestimated, but by an amount which is ill-defined; nevertheless, even taking into account the cost of saving *all* registers `r4` to `r11`, `fpr_add_sub` is still substantially faster than two separate `fpr_add` calls.

Floating-Point Format and Assumptions. A "binary64" value is a 64-bit word which splits into three elements:

- a *sign bit s* of value 0 (positive) or 1 (negative);
- an *exponent e* in the $[0, 2047]$ range;
- a *mantissa m* in the $[0, 2^{52} - 1]$ range.

In general, this represents the value $(-1)^s 2^{e-1075}(2^{52}+m)$. There are some special cases, which use the extreme values of the encoded exponent range:

- If $e = 0$ then the value is either zero (if $m = 0$) or a denormalized value (if $m \neq 0$). Denormalized values are also often called "subnormals". In such cases, the represented value is $(-1)^s 2^{-1074} m$ (i.e. +1 for the exponent, but no $+2^{52}$ for the mantissa).
- If $e = 2047$ then the value is either an infinite (if $m = 0$) or a special Not-a-Number (NaN) (if $m \neq 0$). NaNs are generated from "impossible" operations such as taking the square root of a negative value.

When an operation yields a result with too large an exponent, such that instead an infinite is obtained, then this is an *overflow*. If the exponent is too low and a denormalized value is produced (or an even lower value that is rounded to zero), then this is an *underflow*. For more information on floating-point format, refer to the IEEE standard[8] and the very useful tutorial from Goldberg[10].

In Falcon, we can assume that NaNs, overflows and underflows do not happen. No operation in signature generation leads to any of these conditions. Divisors in `fpr_div` are never zero; `fpr_sqrt` is always invoked on positive values. These assumptions were heuristically assumed (and corroborated with measurements) in the original Falcon submission, and more recently formally proven true[7], with encoded exponents always in the $[547, 1102]$ range[3]. The only special case that must be handled by the code is zeros. Note that there are two zeros, a positive and a negative one (depending on the sign bit s); in our context they are interchangeable, since floating-point values are ultimately rounded to integers when producing the final signature, and the sign of any zero will be ignored. We still prefer to follow the strict IEEE 754 rules so that unit tests can validate that internal values are correct. In particular:

- For any x, $x - x$ (and $x + (-x)$) yields a positive zero, not a negative one.
- For any x and y, the sign of xy is the XOR of the signs of x and y, including if either or both is a zero. The same holds for x/y.
- $\sqrt{x}$ is well-defined when x is a negative zero, and yields a positive zero.

Floating-Point Additions. It can be seen in table 2 that floating-point additions are more expensive than floating-point multiplications. This is due to the fact that additions involve two shifts of 64-bit values by secret amounts, and these operations are expensive:

- The two mantissas must be aligned together; that is, if the exponent of x is e and its mantissa is m, and y has exponent e' and mantissa m' with $e' \leq e$, then m', as an integer,

[3] That paper also detected that the previous implementation was trying to handle underflows by rounding them to zero, but mishandling an edge sub-case; since underflows cannot happen in Falcon, we can simply ignore that case and not even try to handle underflows in any way.

must be right-shifted by $e - e'$ bits before being added to x (or subtracted from x, if x and y have opposite signs). The shift count can be large, even larger than 256, which is the limit for shift counts in the ARMv7 shift opcodes. For constant-time discipline, which case the shift count falls on must not be disclosed through timing differences or a distinct memory access pattern, hence such a shift must process all cases in parallel and select the right one.

– Once the operation has been computed over the mantissas (with enough extra bits for proper rounding), the result must be normalized with a left or right shift, so that the value falls within the $[2^{52}, 2^{53} - 1]$ range that is expected by the format. This again entails secret shift counts.

A contrario, in a multiplication, the product of the source mantissas (with their $+2^{52}$ offset) is necessarily in the $[2^{104}, 2^{106} - 1]$ range and its normalization is easier to perform efficiently. The underlying $64 \times 64 \rightarrow 128$ integer multiplication can be performed in only 4 cycles with two `umull` and two `umaal` opcodes.

The expensive shifts involved in addition can be somewhat optimized with, again, the DSP opcodes in the M4. Let's consider the second shift in the addition process: an integer value has been assembled in register pair `r6:r7` (low word in `r6`, high word in `r7`); that value is less than 2^{57} but can be much lower, even equal to zero. We want to left-shift it such that we get a value in $[2^{63}, 2^{64} - 1]$ (or exactly zero). The normal 64-bit shift sequence (that the compiler generates when shifting a `uint64_t`) is 8 instructions, and getting the number of leading zeros would require 4 or 5 more; but we can do the whole operation in 10 instructions (hence 10 cycles for execution):

```
    clz     r2, r7          @ r2 <- leading zeros of high word
    clz     r3, r6          @ r3 <- leading zeros of low word
    sbfx    r0, r2, #5, #1  @ r5 <- -1 if high word is zero, or 0
    @ At this point, if r7 is non-zero, then we want to keep r2
    @ as is, and r0 = 0, so the umlal below does not change r2.
    @ Otherwise, if r7 = 0, then r2 = 32 and we want to add r3
    @ to r2; in that case, r0 = 0xFFFFFFFF and the umlal computes
    @ r2*2^32 + r3 + r3*(2^32 - 1) = (r2 + r3)*2^32, which sets
    @ the high word (r2) to exactly the value we want.
    umlal   r3, r2, r3, r0

    @ Shift r6:r7 to the left by r2 bits.
    @ If r2 >= 32, then r7 = 0 and r0 = -1, so the umlal below
    @ moves r6 into r7 (and clears r6). Otherwise, r7 != 0, r0 = 0,
    @ and r6 and r7 are unchanged.
    umlal   r6, r7, r6, r0
    @ Left-shift by r2 mod 32
    and     r2, r2, #31
    movs    r1, #1
    lsls    r1, r2
    @ We do a shift by n bits with a multiplication by 2^n.
    @ Result is in r6:r12
    umull   r6, r12, r6, r1
    umlal   r12, r7, r7, r1
```

In an `fpr_add_sub` operation, we compute $x+y$ and $x-y$. The first shift (for "mantissa phasing") can be mutualized, since it does not depend on the operand signs. The scond shift must be performed twice, though, since the two results need not have the same size as integers.

Rounding. The used rounding policy is "roundTiesToEven", which means that the true mathematical result is rounded to the nearest representable value, and if there is a tie (the value falls right in the middle of the range between two representable values) then the output should use the value whose least significant bit is zero. This rounding rule can be implemented in several ways; we use the following method, which is efficient on the M4. Consider the true mathematical result, which ends with the following sequence of bits (with rightmost bit being the least significant):

$$\ldots b_n b_{n-1} b_{n-2} \ldots b_0$$

To bring the value into the representable mantissa range, it must be right-shifted by n bits, making b_n the new least significant bit. In that case, rounding really means:

1. Right-shift the value by n bits.
2. Add 1 to the resulting shifted value in exactly the following cases:
   - $b_n = 1$ and $b_{n-1} = 1$
   - $b_{n-1} = 1$ and there exists $j \in [0, n-2]$ such that $b_j \neq 0$

   Note that the two cases are not mutually exclusive.

It is equivalent to consider the $n+1$-bit integer $r = b_{n-1} b_{n-2} \ldots b_0 b_n$ (i.e. all n dropped bits, and the least significant non-dropped bit b_n pushed at the *low* position): 1 must be added to the shifted value if and only if $r \geq 2^n + 1$, i.e. $r + (2^n - 1) \geq 2^{n+1}$. We arrange for assembling r in the top bits of a register, so that the comparison is really an addition that may or may not generate a carry, which we propagate into the shifted value. Here is an example, extracted from `fpr_add`; the value to normalize is in `r6:r12` and is in the $[2^{63}, 2^{64} - 1]$ range (the top bit of `r12` is set); the low 7 bits of `r6` are zero; the shift count is $n = 11$. The output is expected in `r0:r1` and the sign and exponent have already been set in `r1`, so the mantissa must be *added* into `r1`. The whole operation executes in 6 cycles:

```
@ The low 12 @ bits of r6 (in high-to-low order) are:
@    b4 b3 b2 b1 b0 0000000
@ After a strict right shift, b4 is the lowest bit. Rounding
@ will add +1 to the value if and only if:
@   - b4 = 0 and b3:b2:b1:b0 >= 1001
@   - b4 = 1 and b3:b2:b1:b0 >= 1000
@ Equivalently, we must add +1 after the shift if and only if:
@   b3:b2:b1:b0:b4 + 01111 >= 100000
lsls    r3, r6, #21           @ top(r3) = b3:b2:b1:b0:00...
lsrs    r0, r6, #11
bfi     r3, r0, #27, #1       @ top(r3) = b3:b2:b1:b0:b4:00...
adds    r3, r3, #0x78000000   @ add 01111 to top bits
adcs    r0, r0, r12, lsl #21  @ propagate carry
adcs    r1, r1, r12, lsr #11  @ ... into the whole value
```

2.5 Signature Verification

As seen on figure 1, the majority of the cost of signature verification is spent on hashing. SHAKE256 is used, which internally calls the Keccak-f function for every 136 bytes absorbed or produced. In the FALCON signature verification, the incoming message and nonce are used as a seed to produce a pseudorandom sequence of 16-bit values, from which are produced n values in $[0, q-1]$ with $q = 12289$, and n being the degree. Rejection sampling is used to ensure uniform selection; this implies that the amount of data extracted from SHAKE may vary. On average, 8.59 invocations of Keccak-f are needed for FALCON-512, and 16.66 for FALCON-1024; the BUFF variant (which was the one used for the statistics of figure 1) also implies hashing the public key, adding 7 and 14 Keccak-f invocations for FALCON-512 and FALCON-1024, respectively.

Our Keccak-f implementation uses the classic method of splitting each 64-bit state word into odd-indexed and even-indexed bits, as two 32-bit words; this allows reducing 64-bit word rotations into 32-bit word rotations (and occasional register exchanges), most of which amenable to "free" execution thanks to the ability of ARMv7 Boolean bitwise opcodes to apply a shift or rotation to one of the operands. We favoured code compacity and thus refrained from unrolling several successive rounds; this makes our code about 15% slower but more compact than the classic XKCP implementation[16]. We also chose to apply the bit splitting and merging operations right before (splitting) and after (merging) running Keccak-f, so that the state structure can also serve as a convenient buffer for the produced data, avoiding the need to perform the bit interleaving for each access.

We also reimplemented in assembly most other operations. The NTT is used twice in each verification, and the inverse NTT is used once. Like the Keccak-f implementation, our NTT and inverse NTT are not unrolled, so they execute in 27087 and 29672 cycles, respectively (at $n = 512$). These figures are certainly not as low as those reported in [3] (15974 and 17047 cycles, respectively), but our code is very compact (the two routines have a footprint of 324 and 372 bytes, respectively, and both support all degrees for implementing all variants of FALCON, including the standard degrees 512 and 1024, but also the "toy" variants 4 to 256).

For operations modulo q, the C implementation uses an internal representation such that an integer modulo q is normalized into the $[1, q]$ range (i.e. zero is represented by q, not by 0). This allows the use of Montgomery reduction, i.e. modular division by 2^{32} modulo q, of a 32-bit value x without "large" multiplications:

```
/* Given 0 < x < 2^32 + 2^16 - (2^16 - 1)*q, return x/2^32 mod q
   in the [1,q] range. */
static inline uint32_t
mq_mred(uint32_t x)
{
    x *= Q1I;
    x = (x >> 16) * Q;
    return (x >> 16) + 1;
}
```

This process was already described in [14] (section 2.4) and is particularly adapted to the ARM Cortex-M0+ CPU (a reduced, ARMv6 CPU) as well as large x86 CPUs with AVX2 support (which can perform 16 multiplications of 16-bit values in parallel). On the M4, how-

ever, we have very efficient large multiplications ($32 \times 32 \rightarrow 64$), which we leverage by first relaxing are representation and accepting the whole $[0, q]$ range (i.e. zero has two possible representations). If registers `r10` and `r14` contain q and $-1/q \bmod 2^{32}$, respectively, then the following two-cycle sequence computes the Montgomery reduction of the value contained in register `r0`:

```
@ Montgomery reduction of r0.
@ Input:
@   r0     value x to reduce
@   r10    q = 12289
@   r14    -1/q mod 2^32 = 4143984639
@ Output:
@   r0     x/2^32 mod q, in the [0,q] range
@   r10 and r14 are unmodified. r12 is modified (scratch).
mul     r12, r0, r14
umaal   r14, r0, r12, r10
```

This process works for the whole 32-bit range. Since $q < 2^{15}$, we can consider values in $[0, q]$ to be signed (but non-negative) and perform multiplications using the DSP opcodes such as `smulbb`, which allows us reading values from RAM by pairs without needing to separate them. Moreover, additions and subtractions modulo q can be performed in three-cycle sequences on two values in parallel; e.g. if `r0` contains values x_0 (low half) and x_1 (high half), and `r1` similarly contains y_0 and y_1, then the modular additions $x_0 + y_0$ and $x_1 + y_1$ (with reduction back into $[0, q]$) can be done in three cycles in total:

```
@ Parallel addition modulo q = 12289.
@ Input:
@   r0     values x0 (low) and x1 (high), both in [0,q]
@   r1     values y0 (low) and y1 (high), both in [0,q]
@   r11    q + (q << 16) = 805384193
@ Output:
@   r0     values x0+y0 and x1+y1, in [0,q]
@   r1 and r11 and unmodified. r12 is modified (scratch).
sadd16  r12, r0, r1     @ r12 <- x0+y0 : x1+y1
ssub16  r0, r12, r11    @ r0 <- x0+y0-q : x1+y1-q
sel     r0, r0, r12     @ select non-negative values
```

Similarly, $x_0 - y_0$ and $x_1 - y_1$ are also computed in three cycles:

```
@ Parallel subtraction modulo q = 12289.
@ Input:
@   r0     values x0 (low) and x1 (high), both in [0,q]
@   r1     values y0 (low) and y1 (high), both in [0,q]
@   r10    constant 0
@   r11    q + (q << 16) = 805384193
@ Output:
@   r0     values x0-y0 and x1-y1, in [0,q]
@   r1, r10 and r11 and unmodified. r12 is modified (scratch).
```

```
    ssub16  r0, r0, r1      @ r0 <- x0-y0 : x1-y1
    sel     r12, r10, r11   @ q if negative, 0 otherwise
    sadd16  r0, r0, r12     @ add q if necessary
```

If several signatures are to be verified against the same public key, then there are two easy optimizations that can be applied:

- The conversion of the public key into NTT representation can be done once, and shared, saving one NTT per verification.
- In the BUFF variant, the public key hash can be computed once, and shared, saving about 103000 cycles for Falcon-512 (206000 for Falcon-1024).

In our API, these possible optimizations are not applied, since we favour a simpler one-call operation in which the public key and signature are provided in their encoded formats.

3 Conclusion

The improvements described in this note, over the previous implementation, consist mostly of algorithmic refinements for the key pair generation (imported from ntrugen[13], in particular the use of fixed-point computations instead of floating-point), and better assembly optimization in signature generation and verification. They show that while Falcon key pair generation and signature generation remain somewhat costly operations in microcontrollers, they are not unusable (even at the low speed of 24 MHz, sub-second performance for signature generation is achieved). For signature verification, Falcon is very fast (faster than ML-DSA, but also faster than the best pre-quantum ECDSA and EdDSA implementations at a similar security level) and uses relatively little RAM.

References

1. A. Abdulrahman, V. Hwang, M. Kannwischer and A. Sprenkels, *Faster Kyber and Dilithium on the Cortex-M4*, Applied Cryptography and Network Security – ACNS 2022, Lecture Notes In Computer Science, vol. 13269, pp. 853-871, 2022.
2. *Procedure Call Standard for the Arm® Architecture*, Arm Limited, version 2024Q3, 2024.
3. J. Choi, S. Yoon and S.C. Seo, *Optimized Falcon Verify on Cortex-M4 for Post-Quantum secure UAV communications*, ICT Express, Elsevier, 2024, https://doi.org/10.1016/j.icte.2024.11.002
4. S. Düzlü, R. Fiedler and M. Fischlin, *BUFFing FALCON without Increasing the Signature Size*, https://eprint.iacr.org/2024/710
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. P. Gajland, J. Janneck and E. Kiltz, *A Closer Look at Falcon*, https://eprint.iacr.org/2024/1769
7. V. Hwang, *Formal Verification of Emulated Floating-Point Arithmetic in Falcon*, Advances in Information and Computer Security: 19th International Workshop on Security – IWSEC 2024, Lecture Notes in Computer Science, vol. 14977, pp. 125-141, 2024.
8. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985, https://doi.org/10.1109%2FIEEESTD.1985.82928
9. Information Technology Laboratory, *Module-Lattice-Based Digital Signature Standard*, National Institute of Standard and Technology, FIPS 204, 2024.
10. D. Goldberg, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys (CSUR), vol. 23, issue 1, pp. 5-48, 1991.
11. 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
12. *Post-Quantum Cryptography*, National Institute of Standard and Technology, https://www.nist.gov/pqcrypto
13. T. Pornin, *New Efficient, Constant-Time Implementations of Falcon*, https://eprint.iacr.org/2019/893
14. T. Pornin, *Efficient Elliptic Curve Operations On Microcontrollers With Finite Field Extensions*, https://eprint.iacr.org/2020/009
15. T. Pornin, *Improved Key Pair Generation for Falcon, BAT and Hawk*, https://eprint.iacr.org/2023/290
16. G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche and R. Van Keer, *eXtended Keccak Code Package*, https://github.com/XKCP/XKCP