Towards Practical Microcontroller Implementation of the Signature Scheme Falcon

Tobias Oder[1], Julian Speith[1], Kira Höltgen[1], and Tim Güneysu[1,2]

[1] Ruhr-University Bochum, Germany
[2] DFKI, Germany
{tobias.oder,julian.speith,kira.hoeltgen,tim.gueneysu}@rub.de

Abstract. The majority of submissions to NIST's recent call for Post-Quantum Cryptography are encryption schemes or key encapsulation mechanisms. Signature schemes constitute a much smaller group of submissions with only 21 proposals. In this work, we analyze the practicability of one of the latter category – the signature scheme `Falcon` with respect to its suitability for embedded microcontroller platforms.
`Falcon` has a security proof in the QROM in combination with smallest public key and signature sizes among all lattice-based signature scheme submissions with decent performance on common x86 computing architectures. One of the specific downsides of the scheme is, however, that according to its specification it is "*non-trivial to understand and delicate to implement*".
This work aims to provide some new insights on the realization of `Falcon` by presenting an optimized implementation for the ARM Cortex-M4F platform. This includes a revision of its memory layout as this is the limiting factor on such constrained platforms. We managed to reduce the dynamic memory consumption of `Falcon` by 43% in comparison to the reference implementation. Summarizing, our implementation requires 682 ms for key generation, 479 ms for signing, and only 3.2 ms for verification for the $n = 512$ parameter set.

Keywords: Ideal lattices, Falcon, Cortex-M, Microcontroller, NIST PQC

1 Introduction

With the progress on quantum computing that has been made in recent years, the possibility of a powerful quantum computer being build in the coming years seems as likely as never before. The impact of the sheer existence of such a machine to current real world cryptography would be disastrous. Of special significance in that regard is Shor's algorithm for prime factorization and discrete logarithms [32], which, given a powerful quantum computer, allows for polynomial time attacks on almost all public-key algorithms that are in use today.

As a result, research in the area of post-quantum cryptography has significantly picked up in speed for the last couple of years. A lot of work is currently being put into the construction of quantum-secure cryptographic schemes that

one day could replace today's most widely distributed algorithms. This effort culminated in the NISTs call for post-quantum cryptographic algorithms [27] that ended in November 2017. While most of the first round submissions focus on key exchange or key encapsulation schemes (KEMs), a few are dealing with the problem of generating cryptographically sound digital signatures. These include the lattice-based schemes `pqNTRUsign` [10], `qTESLA` [7], `Dilithium` [12], `DRS` [28], and `Falcon` [15], but also hash-based algorithms such as `SPHINCS`$^+$ [21] or schemes based on multivariate quadratics.

There are two competing approaches to realize lattice-based signature schemes. While the majority of them is constructed by applying the Fiat-Shamir transform to an authentication scheme, `Falcon` is based on the so-called hash-and-sign approach. The major difference is that while Fiat-Shamir schemes in general have a better performance, hash-and-sign ones can be proven to be secure in the Random Oracle Model (ROM) and even the Quantum Random Oracle Model (QROM). Another advantage of hash-and-sign signatures is that it is possible to construct identity-based encryption schemes out of a signature scheme like `Falcon` [13].

In particular in IoT infrastructures with critical requirements for long-term security, it is important to identify solutions that can still be deployed on contemporary small devices. Embedded systems found in automotive, consumer, or medical applications, for instance, demand an alternative solution that can withstand future attacks in the long run. With the implementation presented in this work, we show that `Falcon` can be a solution in this context.

1.1 Related Work

The majority of practical work on lattice-based NIST PQC candidates for embedded devices focuses on encryption schemes or key encapsulation mechanisms. An implementation of Saber [11] for ARM Cortex-M microcontroller platforms by Karmakar et al. [23] has been published in TCHES'18 and has since then been further optimized by Kannwischer et al. in [22]. There is also a microcontroller and FPGA implementation of Frodo [3] by Howe et al. [20]. Albrecht et al. developed an implementation of Kyber [5] that exploits existing RSA co-processors [2]. Finally, there is a Cortex-M4 implementation of Round5 [6], a scheme that resulted from the merger of Round2 [17] and HILA5 [30], by Saarinen et al. [31].

A detailed list of publications related to microcontroller implementations of NIST PQC candidates is available at the PQCzoo [19]. The most comprehensive collection of ARM Cortex-M4 implementations can be found in the `pqm4` [1] library. Most of these implementations are rather straight-forward portings of their respective reference implementations, but it also features implementations that are described in dedicated publications. The library `pqm4` contains ten KEMs and only three signature schemes, namely `Dilithium`, `qTESLA`, and `SPHINCS`$^+$. `Falcon` has not been included in `pqm4`.

1.2 Contribution

The Falcon web page [16] mentions that the comparatively low memory consumption of Falcon is one of the highlights of the algorithm and states that "Falcon is compatible with small, memory-constrained embedded devices". The reference implementation of the Falcon submission however paints a different picture as it uses 210 kB of dynamic RAM memory during the signing step. In this work, we want to verify the claim of the Falcon submission to be well suited for memory-constrained embedded devices by presenting the first embedded microcontroller implementation of the signature scheme Falcon.

To do so, we apply a number of memory-saving techniques to reduce the dynamic memory consumption in comparison to the reference implementation by 43%. Our implementation on an ARM Cortex-M4 requires 64 kB of RAM and has a runtime of 682 ms for key generation, 479 ms for signing, and only 3.2 ms for verification using Falcon-512.

In its Call for Proposals [27] NIST explicitly states that the flexibility of a proposed scheme is one major evaluation criterion for the standardization process. The document furthermore defines flexibility to include that "algorithms can be implemented securely and efficiently on a wide variety of platforms, including constrained environments". In our work, we show that Falcon fulfills this requirement to some extent and also highlight the limitations of the scheme regarding its implementation on embedded platforms.

2 Preliminaries

In this chapter, we discuss the mathematical background that is crucial for the understanding of this paper.

2.1 Notation

We follow the notation of the Falcon specification [15]. Matrices are written as bold uppercase letter, vectors as bold lowercase letter, and scalars and polynomials as italic letters. An asterisk marks the component-wise adjoint of the transpose of a matrix. Sampling a value a from a Gaussian distribution is written as $a \leftarrow D_{\mathbb{Z},x,\sigma}$ where x denotes the center of the distribution and σ denotes its standard deviation.

2.2 The Falcon Signature Scheme

Due to the complexity of Falcon, a detailed description of all its components is out of the scope of this work. In the following we will broadly describe the key generation, signing, and verification procedures of Falcon and refer to the official specification [15] for more details.

Key generation, as shown in Algorithm 1, can be separated into two distinct parts. First, it generates polynomials $f, g, F, G \in \mathbb{Z}[x]/(\phi)$ that fulfill the NTRU

equation $fG - gF = q \bmod \phi$ using the algorithm NTRUGen. The second part deals with the construction of the `Falcon` tree T using the LDL* decomposition of the matrix $\mathbf{G} = \mathbf{B}\mathbf{B}^*$. Since our optimizations strongly depend on the tree generation algorithm, it can be found in Appendix A. Keygen then returns a public key $\mathsf{pk} = h = gf^{-1} \bmod q$ and a secret key $\mathsf{sk} = (\hat{\mathbf{B}}, \mathsf{T})$.

Algorithm 1 $\mathsf{Keygen}(\phi, q)$

Require: A monic polynomial $\phi \in \mathbb{Z}[x]$, a modulus q
Ensure: A secret key sk, a public key pk

$f, g, F, G, \gamma \leftarrow \mathsf{NTRUGen}(\phi, q)$ ▷ Solving the NTRU equation
$\mathbf{B} \leftarrow \left[\begin{array}{c|c} g & -f \\ \hline G & -F \end{array}\right]$
$\hat{\mathbf{B}} \leftarrow \mathsf{FFT}(\mathbf{B})$
$\mathbf{G} \leftarrow \hat{\mathbf{B}} \times \hat{\mathbf{B}}^*$
$\mathsf{T} \leftarrow \mathsf{ffLDL}^*(\mathbf{G})$ ▷ Computing the LDL* tree
if ϕ is binary **then**
    $\sigma \leftarrow 1.55\sqrt{q}$
else if ϕ is ternary **then**
    $\sigma \leftarrow 1.32 \cdot 2^{1/4}\sqrt{q}$
for each leaf leaf of T **do** ▷ Normalization step
    $\mathsf{leaf.value} \leftarrow \sigma/\sqrt{\mathsf{leaf.value}}$
$\mathsf{sk} \leftarrow (\hat{\mathbf{B}}, \mathsf{T})$
$h \leftarrow gf^{-1} \bmod q$
$\mathsf{pk} \leftarrow h$
return sk, pk

For *signature generation*, Algorithm 2 summarizes the required steps. First it computes a hash value $c \in \mathbb{Z}_q[x]/(\phi)$ of the message m and a salt r. It then uses the secret key sk to compute short values s_1, s_2 such that $s_1 + s_2 h = c \bmod q$ by leveraging its knowledge of f, g, F, G. This is done using the ffSampling algorithm, which is also given in Appendix A. Since s_1 can be reconstructed from s_2, the hash c, and public key h, it suffices to output a compressed version of s_2 as the signature, which also includes the random seed r.

Algorithm 2 $\mathsf{Sign}(\mathsf{m}, \mathsf{sk}, \beta)$

Require: A message m, a secret key sk, a bound β
Ensure: A signature sig of m
1: $\mathsf{r} \leftarrow \{0,1\}^{320}$ uniformly
2: $c \leftarrow \mathsf{HashToPoint}(\mathsf{r} \| \mathsf{m})$
3: $\mathbf{t} \leftarrow (\mathsf{FFT}(c), \mathsf{FFT}(0)) \cdot \hat{\mathbf{B}}^{-1}$
4: **do**
5: $\quad \mathbf{z} \leftarrow \mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$
6: $\quad \mathbf{s} = (\mathbf{t} - \mathbf{z})\hat{\mathbf{B}}$
7: **while** $\|\mathbf{s}\| > \beta$
8: $(s_1, s_2) \leftarrow \mathsf{invFFT}(\mathbf{s})$
9: $\mathbf{s} \leftarrow \mathsf{Compress}(s_2)$
10: **return** $\mathsf{sig} = (\mathsf{r}, \mathsf{s})$

Signature verification as shown in Algorithm 3 is rather straightforward and starts by hashing m and r into the hash value c again. Next, s_1 is recomputed and the algorithm checks whether $||(s_1, s_2)|| \leq \beta$ is satisfied with β being some predefined acceptance bound. Only if that bound holds for the given signature, it is accepted as valid.

Algorithm 3 $\mathsf{Verify}(\mathsf{m}, \mathsf{sig}, \mathsf{pk}, \beta)$

Require: A message m, a signature $\mathsf{sig} = (\mathsf{r}, \mathsf{s})$, a public key $\mathsf{pk} = h \in \mathbb{Z}_q[x]/(\phi)$, a bound β
Ensure: Accept or reject
1: $c \leftarrow \mathsf{HashToPoint}(\mathsf{r} \| \mathsf{m}, q, n)$
2: $s_2 \leftarrow \mathsf{Decompress}(\mathsf{s})$
3: $s_1 \leftarrow c - s_2 h \bmod q$
4: **if** $\|(s_1, s_2)\| \leq \beta$ **then**
5: $\quad$ accept
6: **else**
7: $\quad$ reject

3 Microcontroller Implementation

This section deals with two approaches to implement the `Falcon` signature scheme on our target architecture. The first one combines the tree generation with the fast Fourier sampler to reduce memory requirements, while the second one excludes the key generation from the microcontroller entirely and uses precomputed keys instead.

3.1 Target Platform

The STM32F4DISCOVERY board serves as a constrained target platform for our implementation. Its microcontroller has an 32-bit ARM Cortex-M4F core

that runs with a clock frequency of up to 168 MHz. The board offers 192 kB of RAM as well as 1 MB of flash memory. Furthermore, it features a true random number generator (TRNG) based on analog circuitry and a floating-point unit (FPU). But as the FPU only works with single-precision floating point values, we cannot employ it for our implementation.

3.2 Analysis of the Reference Implementation

The analysis of the reference implementation from the `Falcon` submission package is our starting point for the development of our optimized ARM Cortex-M4 implementation. We first measured the dynamic memory consumption of the reference implementation in terms of stack and heap usage. We determine the stack usage with the help of stack canaries. To employ this technique, we start by filling the stack with a magic number before the operation to be measured is executed. Afterwards we check up to which point the magic numbers have been overwritten and therefrom conclude the stack usage. We determine the heap usage by counting the `malloc()` calls in the reference implementation manually as there are only a few of them in the source code.

The resulting dynamic memory consumption of the reference implementation can be seen in Table 1. The first point to note is that 210 kB are required for the signing operation for $n = 1024$ what clearly would not fit into the 192 kB RAM of our STM32F4DISCOVERY development board. Another issue is that in most use cases cryptographic algorithms are subcomponents of a main application on the microcontroller that employes the security functions to securely transmit, receive, or store data. As a result it is not sufficient to make the implementation barely fit the memory of our target platform, but we also need to reserve space for the main application that will be also placed the microcontroller.

Table 1. Dynamic memory usage of the reference implementation in bytes for $n = 512$ and $n = 1024$.

Operation	**Stack Memory**	**Heap Memory**	**Total Memory**
$n = 512$			
Key Generation	18,624	14,777	33,401
Sign	22,632	94,040	116,672
Verify	13,456	2,464	15,920
$n = 1024$			
Key Generation	24,200	29,113	53,313
Sign	28,696	181,080	209,776
Verify	19,080	2,464	21,544

We identify the large `Falcon` tree used in the fast Fourier sampler during signature generation as the memory bottleneck. Considering the case $n = 1024$, that tree takes up 90 kB of the RAM. To execute `Falcon` on the target architecture, we present two possible solutions: We can either adapt the algorithm in a way that is more memory-conserving, or we may implement only the signature generation and verification while using those algorithms in combination with precomputed keys, which include the `Falcon` tree. The keys can then be stored in Flash memory to unburden the RAM. The latter approach is rather straightforward since one only needs to precompute the keys and load them onto the device. However, for many use cases this is not a satisfiable solution, as we may want to generate new keys over time. Therefore we focus on algorithmic changes for the remainder of this section.

To reduce the memory footprint, our implementation will merge the tree generation and the fast Fourier sampling (cf. Appendix A) into a single algorithm $\mathsf{ffSampling}_n^*$ that is described in Algorithm 4. Referring to signature generation as shown in Algorithm 2, we then replace $\mathsf{ffSampling}_n$ with $\mathsf{ffSampling}_n^*$. The `Falcon` tree is therefore no longer part of the secret key sk and is instead computed on-the-fly during sampling. As the matrix $\mathbf{G}$ is required for the computation of the tree, we additionally need to compute it prior to the sampling step. We can therefore exclude the computation of $\mathbf{G}$ from key generation, since it has no use in that algorithm anymore. That way we only need to keep a small subtree in memory, which is generated whenever the respective part of the tree is required. As a consequence of this memory tradeoff we have to recompute the entire tree for each signature generation with a negative impact on the overall performance. Finally, our embedded implementation natively only supports `Falcon-512` and `Falcon-1024`, though the same concepts can be directly applied for `Falcon-768` as well.

3.3 Memory Optimizations

Our fast Fourier sampler with integrated tree generation is the most expensive operation in terms of memory requirements during the signing procedure. We optimized our implementation such that it only needs 8 kB of temporary space (i.e. n double elements), as the in- and outputs alone already take up 56 kB of RAM for $n = 1024$. The flowchart in Figure 1 shows that it is not possible to perform this operation in-place without overriding the inputs. For the sake of simplicity, the flowchart does not include splitting and merging operations. After the first call to $\mathsf{ffSampling}_n^*$, the first output is already calculated and we therefore cannot use its memory to store intermediate results. Hence we leverage the memory, which in the end will contain the second output, to keep the intermediate results in the meantime. However, we still need to store L_{10} as output of the LDL^* somewhere. Therefore it is inevitable to use temporary memory within the sampler without major algorithmic changes.

Algorithm 4 $\mathsf{ffSampling}_n^*(\mathbf{t}, \mathbf{G})$

Require: $\mathbf{t} = (t_0, t_1) \in \mathsf{FFT}(\mathbb{Q}[x](x^n + 1))^2$ and a full-rank Gram matrix $\mathbf{G} \in \mathsf{FFT}(\mathbb{Q}[x](x^n + 1))^{2\times 2}$, $\sigma \leftarrow 1.55\sqrt{q}$
Ensure: $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}(\mathbb{Z}[x](x^n + 1))^2$

if $(n = 1)$ **then**
  $\sigma' \leftarrow \sigma\sqrt{G_{00}}$
  $z_0 \leftarrow D_{\mathbb{Z},t_0,\sigma'}$
  $z_1 \leftarrow D_{\mathbb{Z},t_1,\sigma'}$
  return $\mathbf{z} = (z_0, z_1)$
$(\mathbf{L}, \mathbf{D}) \leftarrow \mathsf{LDL}^*(\mathbf{G})$ ▷ $\mathbf{L} = \left[\begin{array}{c|c} 1 & 0 \\ \hline L_{10} & 1 \end{array}\right], \mathbf{D} = \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & D_{11} \end{array}\right]$
$d_{10}, d_{11} \leftarrow \mathsf{splittfft}_2(D_{11})$ ▷ Handle right child
$\mathbf{t}_1 \leftarrow \mathsf{splittfft}_2(t_1)$
$\mathbf{G}_1 \leftarrow \left[\begin{array}{c|c} d_{10} & d_{11} \\ \hline xd_{11} & d_{10} \end{array}\right]$
$\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_1, \mathbf{G}_1)$
$z_1 \leftarrow \mathsf{mergefft}_2(\mathbf{z}_1)$
$t_0' \leftarrow t_0 + (t_1 - z_1) \odot L_{10}$ ▷ Handle left child
$d_{00}, d_{01} \leftarrow \mathsf{splittfft}_2(D_{00})$
$\mathbf{t}_0 \leftarrow \mathsf{splittfft}_2(t_0')$
$\mathbf{G}_0 \leftarrow \left[\begin{array}{c|c} d_{00} & d_{01} \\ \hline xd_{01} & d_{00} \end{array}\right]$
$\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_0, \mathbf{G}_0)$
$z_0 \leftarrow \mathsf{mergefft}_2(\mathbf{z}_0)$
return $\mathbf{z} = (z_0, z_1)$

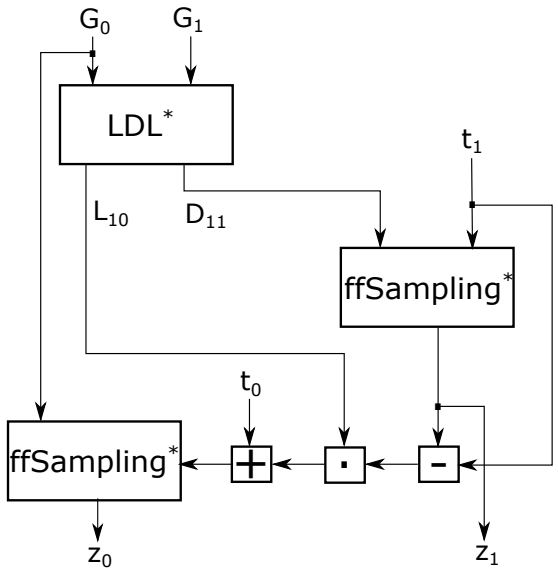

Fig. 1. Flowchart of our fast Fourier sampler with integrated tree generation $\mathsf{ffSampling}_n^*$.

3.4 Timing Analysis

Timing attacks are a fundamental threat to every cryptographic operation involving secret values [25]. With a timing attack an adversary obtains information about the secret key by observing the execution time of the secret key operation, i.e., the signing operation in signature schemes. Timing attacks even work remotely over networks as shown in [8]. The most basic countermeasure against these attacks is to make sure that the execution time of an implementation is completely independent of the secret key, typically referred to as a *constant-time implementation*. Our implementation is currently designed for being **not** constant-time for two reasons. First, many embedded use-cases only require the verification of signatures (e.g., for the verification of authentic firmware updates or other applications where the embedded device is used as authenticated message sink only). Hence, constant time is not an issue for those embedded implementations for which only public data (i.e., the public key) is used. Second, there are particular components in the design of `Falcon` that make a constant-time implementation of the signature generation challenging:

1. Falcon requires to draw samples according to some Gaussian distribution. A lot of research has been focused on developing efficient algorithms for Gaussian sampling [9,26,14,24]. One major difference in comparison to other lattice-based schemes, like KEMs based on the Learning with Errors (LWE) problem, is that in `Falcon` the standard deviation of the Gaussian distribution varies between 1.2 and 1.9 with a precision of 53 bits. Therefore we cannot use constant-time table-based approaches like [24] as a sampling algorithm. Because of the required precision of the sampler, it is also not possible to use the constant-time binomial approach that is utilized in many lattice-based KEMs. The authors of `Falcon` propose to employ a rejection-based approach that is rather inefficient but has an execution time that is independent of the output.
2. Another, more critical obstacle in achieving a constant-time implementation is the use of floating-point arithmetic in `Falcon`. We cannot make use of the floating point unit build into the ARM Cortex-M4F to perform these floating-point operations of `Falcon` as it only works with single precision, while `Falcon` requires double precision operations. Therefore floating point calculations are handled by C runtime library functions, which in turn are usually not constant-time, especially in the case of division or square root operations that are also present in `Falcon`. There are attempts to realize constant-time floating point arithmetic for x86 processors at USENIX'16 [29] and CCS'18 [4]. These works however report a massive performance penalty when their constant-time floating point libraries are utilized resulting in the software being up to one order of magnitude slower than the standard C library functions. However, we are not aware of such libraries for microcontroller platforms and therefore this timing behavior of `Falcon` is one major challenge for its deployment in embedded applications.

4 Results and Comparison

In this section, we discuss the results of our implementation and compare it with others.

4.1 Evaluation Methodology

We evaluate our work by using the *OpenSTM32 System Workbench* (version 2.6), which is based on the development environment `Eclipse` and has specifically been designed to support the development for ARM-based STM32 boards. The IDE uses the GNU ARM Embedded Toolchain (version 7.2) and we set the optimization level to `-O3`. Determining the performance of our implementation is done by using the cycle count register `DWT_CYCCNT` of the *Data Watchpoint and Trace* unit that the Cortex-M4F offers. We assess dynamic RAM consumption by making use of stack canaries as described in Section 3.2.

4.2 Results

Table 2 summarizes the cycle counts of our implementations. We can see from the table that the `Falcon` verification is two orders of magnitude faster than the signing operation or the key generation. For comparability with the `pqm4` library [1] the measurements were obtained at 24 MHz. Translated to 168 MHz, verification would take only 3.2 ms while signing takes 479 ms for $n = 512$ without precomputed keys. Key generation even exceeds the signing operation and requires 682 ms to complete. The cost of the signing operation is dominated by the cost of the fast Fourier sampler as this component accounts for 92% of its total cycle count. In turn, the cost of the fast Fourier sampler heavily depends on the performance of the Gaussian sampler that is executed $2n$ times during the fast Fourier sampling. The $2n$ calls to the Gaussian sampler account for 73% of the cycle count of the entire signature generation.

The Gaussian sampler is therefore the main bottleneck in terms of cycle count of the scheme. Using fixed keys increases the performance of the signing by approximately 10%. This is mainly because we do not have to compute the `Falcon` tree in this case. However, fixed keys do not impact the verification. Another observation is that the FFT, which operates on complex double-precision floating point numbers and is required only during signing and key generation, is one order of magnitude more expensive than the NTT that works on plain integers. Nonetheless, the cost of the FFT is still negligible in comparison to the fast Fourier sampling.

In Table 3 we furthermore present the dynamic memory consumption of our implementations. The signing operation has the highest memory consumption and therefore the total memory consumption of the scheme is equal to the memory requirements of the signing operation. In contrast to the reference implementation we do not allocate memory on the heap and the dynamic memory consumption is therefore entirely determined by the stack usage of the implementation. We reduce the memory requirements of the scheme by 43% for $n = 1024$

in comparison to the reference implementation. Using fixed keys further increases the RAM savings to a total of 55% in comparison to the reference implementation as the keys are stored in Flash memory instead.

Table 2. Clock cycle counts for our ARM implementations of `Falcon` at 24 MHz. All results are averaged over 100 runs. The fast Fourier sampling cycle counts marked with † include the generation of the `Falcon` tree.

	`Falcon`		`Falcon` with fixed keys	
Operation	$n = 512$	$n = 1024$	$n = 512$	$n = 1024$
Key Generation	114,546,135	365,950,978	-	-
Sign	80,503,242	165,800,855	72,261,930	147,330,702
Verify	530,900	1,046,700	529,900	1,083,100
solveNTRU	65,240,266	209,500,594	-	-
Fast Fourier Sampling	$74,433,097^{\dagger}$	$148,600,140^{\dagger}$	64,354,464	130,468,405
$2n$ Gaussian samples	58,541,540	116,768,948	57,947,926	115,855,189
Compute G	583,800	1,131,800	-	-
FFT	772,200	1,716,300	772,800	1,645,100
NTT	75,900	157,700	75,900	159,700

Table 3. Dynamic memory usage in bytes for our ARM Cortex-M4 implementations in comparison with the reference implementation. For our ARM implementations, we only use the stack. We do not allocate extra memory on the heap.

	Reference		M4		Fixed Keys M4	
Operation	$n = 512$	$n = 1024$	$n = 512$	$n = 1024$	$n = 512$	$n = 1024$
Key Gen	33,401	53,313	40,560	51,704	-	-
Sign	116,672	209,776	63,652	120,596	50,508	94,260
Verify	15,920	21,544	6,261	11,893	5,364	10,100

4.3 Comparison

In Table 4 we compare our work with ARM Cortex-M4 implementations of other post-quantum schemes that were either taken from the `pqm4` library [1] or the work of Oder et al. [18]. The security level is given according to the NIST classifications in the Call for Proposals [27]. In this comparison `Falcon` has the lowest execution time for the verification. Even the high-security $n = 1024$

instantiation of Falcon verifies signed messages in about the same time as qTESLA instantiated at a lower security level. Dilithium and qTESLA both have a faster signing and key generation. The major advantage of Falcon over these schemes however is that Falcon comes with a security proof in the ROM and QROM while Dilithium does not have such a proof. qTESLA can be instantiated with "provably-secure" parameters or "heuristic" parameters. The numbers in Table 4 refer to the heuristic instantiation. The minimal security assumptions of SPHINCS$^+$ make it the most conservative choice. The implementation of SPHINCS$^+$ is also the only one from the table that has a data-independent execution time. The signing performance however is four orders of magnitude worse than the signing performance of qTESLA at the same security level. We therefore consider Falcon to be a reasonable trade-off between performance and security.

Table 4. Comparison of our implementation with ARM implementations of other schemes. The given security levels refer to the security categories defined by NIST [27]. For our work, a security level of 1 means that $n = 512$ and level 5 translates to $n = 1024$. The stack memory is given in bytes. The runtime of the key generation, signing, and verification is given in cycles. Our fixed-key implementations are marked by †.

Impl.	Sec.	Stack	Key Gen	Sign	Verify
This work	Level 1	63,652	114,546,135	80,503,242	530,900
	Level 5	120,596	365,950,978	165,800,855	1,046,700
This work†	Level 1	50,508	-	72,261,930	529,900
	Level 5	94,260	-	147,330,702	1,083,100
Dilithium [12]	Level 2	86,568	2,320,362	8,348,349	2,342,191
qTESLA [1]	Level 1	29,336	17,545,901	6,317,445	1,059,370
	Level 3	58,112	30,720,411	11,987,079	2,225,296
SPHINCS$^+$ [1]	Level 1	10,768	4,439,815,208	61,665,898,904	72,326,283

5 Conclusion

In this work, we presented a microcontroller implementation of the lattice-based signature scheme Falcon. Our implementation is memory-efficient and, in contrast to the reference implementation, does fit into the memory of our target platform. We also show that the implementation can be further optimized in terms of performance and memory consumption if the use case does not require to generate a key pair on the device itself. The extremely high performance of the verification makes Falcon a suitable scheme for use cases in which the target device does not have to generate a signature, e.g., for the verification of software

updates. For future work, optimizations of the Gaussian sampler may result in a huge performance gain during signature generation. One obstacle however is that the signing operation cannot easily be realized in constant-time due to the required floating-point operations.

Acknowledgement

We would also like to thank the anonymous reviewers for their very valuable and helpful feedback. The research in this work was supported in part by the European Unions Horizon 2020 program under project number 644729 SAFEcrypto and 780701 PROMETHEUS.

References

1. pqm4 - Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4, accessed: 2018-11-13
2. Albrecht, M.R., Hanser, C., Höller, A., Pöppelmann, T., Virdia, F., Wallner, A.: Learning with errors on RSA co-processors. IACR Cryptology ePrint Archive 2018, 425 (2018), https://eprint.iacr.org/2018/425
3. Alkim, E., Bos, J.W., Ducas, L., Longa, P., Mironov, I., Naehrig, M., Nikolaenko, V., Peikert, C., Raghunathan, A., Stebila, D., Easterbrook, K., LaMacchia, B.: FrodoKEM Learning With Errors key encapsulation. https://frodokem.org/files/FrodoKEM-specification-20171130.pdf, accessed: 2018-11-13
4. Andrysco, M., Nötzli, A., Brown, F., Jhala, R., Stefan, D.: Towards verified, constant-time floating point operations. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 1369–1382. ACM (2018), http://doi.acm.org/10.1145/3243734.3243766
5. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schnack, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/CRYSTALS_Kyber.zip, accessed: 2018-11-30
6. Bhattacharya, S., García-Morchón, Ó., Laarhoven, T., Rietman, R., Saarinen, M.O., Tolhuizen, L., Zhang, Z.: Round5: Compact and fast post-quantum public-key encryption. IACR Cryptology ePrint Archive 2018, 725 (2018), https://eprint.iacr.org/2018/725
7. Bindel, N., Akleylek, S., Alkim, E., Barreto, P.S.L.M., Buchmann, J., Eaton, E., Gutoski, G., Kramer, J., Longa, P., Polat, H., Ricardini, J.E., Zanon, G.: Submission to NIST's post-quantum project: lattice-based digital signature scheme qTESLA. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/qTESLA.zip, accessed: 2018-11-26
8. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks 48(5), 701–716 (2005), https://doi.org/10.1016/j.comnet.2005.01.010

9. Buchmann, J.A., Cabarcas, D., Göpfert, F., Hülsing, A., Weiden, P.: Discrete ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In: Lange, T., Lauter, K.E., Lisonek, P. (eds.) Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8282, pp. 402–417. Springer (2013), https://doi.org/10.1007/978-3-662-43414-7_20
10. Chen, C., Hoffstein, J., Whyte, W., Zhang, Z.: NIST PQ Submission: pqNTRUSign - A modular lattice signature scheme. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/pqNTRUsign.zip, accessed: 2018-11-26
11. D'Anvers, J.P., Karmakar, A., Roy, S.S., Longa, P., Vercauteren, F.: SABER: Mod-LWR based KEM. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SABER.zip, accessed: 2018-11-13
12. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium: A lattice-based digital signature scheme. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018(1), 238–268 (2018), https://doi.org/10.13154/tches.v2018.i1.238-268
13. Ducas, L., Lyubashevsky, V., Prest, T.: Efficient identity-based encryption over NTRU lattices. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 22–41. Springer (2014), https://doi.org/10.1007/978-3-662-45608-8_2
14. Dwarakanath, N.C., Galbraith, S.D.: Sampling from discrete Gaussians for lattice-based cryptography on a constrained device. Appl. Algebra Eng. Commun. Comput. 25(3), 159–180 (2014), https://doi.org/10.1007/s00200-014-0218-3
15. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier lattice-based compact signatures over NTRU. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/Falcon.zip, accessed: 2018-11-26
16. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: Falcon: Fast-Fourier lattice-based compact signatures over NTRU. https://falcon-sign.info/, accessed: 2018-11-26
17. Garcia-Morchon, O., Zhang, Z., Bhattacharya, S., Rietman, R., Tolhuizen, L., Torre-Arce, J.L., Baan, H.: Round2: KEM and PKE based on GLWR. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/Round2.zip, accessed: 2018-11-30
18. Güneysu, T., Krausz, M., Oder, T., Speith, J.: Evaluation of lattice-based signature schemes in embedded systems. In: 25th IEEE International Conference on Electronics Circuits and Systems (2018)
19. Howe, J.: PQCzoo. https://pqczoo.com/, accessed: 2018-11-13
20. Howe, J., Oder, T., Krausz, M., Güneysu, T.: Standard lattice-based key encapsulation on embedded devices. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018(3), 372–393 (2018), https://doi.org/10.13154/tches.v2018.i3.372-393
21. Hulsing, A., Bernstein, D.J., Dobraunig, C., Eichlseder, M., Fluhrer, S., Gazdag, S.L., Kampanakis, P., Kolbl, S., Lange, T., Lauridsen, M.M., Mendel, F., Niederhagen, R., Rechberger, C., Rijneveld, J., Schwabe, P.: SPHINCS+.

https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SPHINCS_Plus.zip, accessed: 2018-11-26

22. Kannwischer, M.J., Rijneveld, J., Schwabe, P.: Faster multiplication in $\mathbb{F}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. IACR Cryptology ePrint Archive 2018, 1018 (2018), https://eprint.iacr.org/2018/1018
23. Karmakar, A., Mera, J.M.B., Roy, S.S., Verbauwhede, I.: Saber on ARM cca-secure module lattice-based key encapsulation on ARM. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018(3), 243–266 (2018), https://doi.org/10.13154/tches.v2018.i3.243-266
24. Karmakar, A., Roy, S.S., Reparaz, O., Vercauteren, F., Verbauwhede, I.: Constant-time discrete Gaussian sampling. IEEE Trans. Computers 67(11), 1561–1571 (2018), https://doi.org/10.1109/TC.2018.2814587
25. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1109, pp. 104–113. Springer (1996), https://doi.org/10.1007/3-540-68697-5_9
26. Micciancio, D., Walter, M.: Gaussian sampling over the integers: Efficient, generic, constant-time. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10402, pp. 455–485. Springer (2017), https://doi.org/10.1007/978-3-319-63715-0_16
27. National Institute of Standards and Technology: Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf, accessed: 2018-11-14
28. Plantard, T., Sipasseuth, A., Dumondelle, C., Susilo, W.: DRS : Diagonal dominant Reduction for lattice-based Signature. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/DRS.zip, accessed: 2018-11-26
29. Rane, A., Lin, C., Tiwari, M.: Secure, precise, and fast floating-point operations on x86 processors. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. pp. 71–86. USENIX Association (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/rane
30. Saarinen, M.J.O.: HILA5. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/Hila5.zip, accessed: 2018-11-30
31. Saarinen, M.J.O., Bhattacharya, S., García-Morchón, Ó., Rietman, R., Tolhuizen, L., Zhang, Z.: Shorter messages and faster post-quantum encryption with Round5 on Cortex M. IACR Cryptology ePrint Archive 2018, 723 (2018), https://eprint.iacr.org/2018/723
32. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Review 41(2), 303–332 (1999), https://doi.org/10.1137/S0036144598347011

A Algorithms

A.1 The Falcon Tree

Please note that there is a typo in the `Falcon` specification [15] in Algorithm 15, Line 3. The description in Algorithm 5 in this section correctly states $n = 2$ instead of $n = 1$.

Algorithm 5 $\mathsf{ffLDL}^*(\mathbf{G})$

Require: A full-rank Gram matrix $\mathbf{G} \in \mathsf{FFT}(\mathbb{Q}[x]/(x^n+1))^{2\times 2}$
Ensure: A binary tree T

1: $(\mathbf{L}, \mathbf{D}) \leftarrow \mathsf{LDL}^*(\mathbf{G})$ ▷ $\mathbf{L} = \left[\begin{array}{c|c} 1 & 0 \\ \hline L_{10} & 1 \end{array}\right], \mathbf{D} = \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & D_{11} \end{array}\right]$
2: $\mathsf{T}.\text{value} \leftarrow L_{10}$
3: **if** $(n = 2)$ **then**
4: $\quad \mathsf{T}.\text{leftchild} \leftarrow D_{00}$
5: $\quad \mathsf{T}.\text{rightchild} \leftarrow D_{11}$
6: $\quad$ **return** T
7: $d_{00}, d_{01} \leftarrow \mathsf{splittfft}_2(D_{00})$
8: $d_{10}, d_{11} \leftarrow \mathsf{splittfft}_2(D_{11})$
9: $\mathbf{G}_0 \leftarrow \left[\begin{array}{c|c} d_{00} & d_{01} \\ \hline x d_{01} & d_{00} \end{array}\right]$
10: $\mathbf{G}_1 \leftarrow \left[\begin{array}{c|c} d_{10} & d_{11} \\ \hline x d_{11} & d_{10} \end{array}\right]$
11: $\mathsf{T}.\text{leftchild} \leftarrow \mathsf{ffLDL}^*(\mathbf{G}_0)$
12: $\mathsf{T}.\text{rightchild} \leftarrow \mathsf{ffLDL}^*(\mathbf{G}_1)$
13: **return** T

A.2 Fast Fourier Sampling

The description can be found in Algorithm 6.

Algorithm 6 $\mathsf{ffSampling}_n(\mathbf{t}, \mathsf{T})$

Require: $\mathbf{t} = (t_0, t_1) \in \mathsf{FFT}(\mathbb{Q}[x]/(x^n+1))^2$ and a Falcon tree T
Ensure: $\mathbf{z} = (z_0, z_1) \in \mathsf{FFT}(\mathbb{Z}[x]/(x^n+1))^2$

if $(n = 1)$ **then**
  $\sigma' \leftarrow \mathsf{T}.\text{value}$
  $z_0 \leftarrow D_{\mathbb{Z},t_0,\sigma'}$
  $z_1 \leftarrow D_{\mathbb{Z},t_1,\sigma'}$
  return $\mathbf{z} = (z_0, z_1)$
$(\mathsf{T}_0, \mathsf{T}_1) \leftarrow (\mathsf{T}.\text{leftchild}, \mathsf{T}.\text{rightchild})$
$\mathbf{t}_1 \leftarrow \mathsf{splittfft}_2(t_1)$
$\mathbf{z}_1 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_1, \mathsf{T}_1)$
$z_1 \leftarrow \mathsf{mergefft}_2(\mathbf{z}_1)$
$t'_0 \leftarrow t_0 + (t_1 - z_1) \odot \mathsf{T}.\text{value}$
$\mathbf{t}_0 \leftarrow \mathsf{splittfft}_2(t'_0)$
$\mathbf{z}_0 \leftarrow \mathsf{ffSampling}_{n/2}(\mathbf{t}_0, \mathsf{T}_0)$
$z_0 \leftarrow \mathsf{mergefft}_2(\mathbf{z}_0)$
return $\mathbf{z} = (z_0, z_1)$