Fly, you fool! Faster Frodo for the ARM Cortex-M4

Joppe W. Bos[1], Simon Friedberger[1,2], Marco Martinoli[3], Elisabeth Oswald[3], and Martijn Stam[3]

[1] NXP Semiconductors joppe.bos@nxp.com
[2] KU Leuven - iMinds - COSIC simon.friedberger@esat.kuleuven.com
[3] University of Bristol, United Kingdom
marco.martinoli, elisabeth.oswald, martijn.stam@bristol.ac.uk

Abstract. We present an efficient implementation of FrodoKEM-640 on an ARM Cortex-M4 core. We leverage the single instruction, multiple data paradigm, available in the instruction set of the ARM Cortex-M4, together with a careful analysis of the memory layout of matrices to considerably speed up matrix multiplications. Our implementations take up to 79.4% less cycles than the reference. Moreover, we challenge the usage of a cryptographically secure pseudorandom number generator for the generation of the large public matrix involved. We argue that statistically good pseudorandomness is enough to achieve the same security goal. Therefore, we propose to use xoshiro128** as a PRNG instead: its structure can be easily integrated in FrodoKEM-640, it passes all known statistical tests and greatly outperforms previous choices. By using xoshiro128** we improve the generation of the large public matrix, which is a considerable bottleneck for embedded devices, by up to 96%.

Keywords: LWE · Frodo · ARM Cortex-M4 · SIMD · PRNG

1 Introduction

FrodoKEM-640 is a Key Encapsulation Mechanism (KEM) submitted to the NIST post-quantum standardisation effort [10] and designed to be conservative yet practical. Its security is based on the hardness of the (plain) Learning With Errors (LWE) problem [11]. On the one hand this means that security follows from the hardness of certain problems over generic “unstructured” lattices. On the other, the main operation to achieve such strong security guarantees and the desired functionality is multiplication by large matrices. This makes FrodoKEM-640 less attractive from a performance perspective compared to, for example, the ring [8] and module [6] “algebraically structured” variants.

Performance of the implementation informs deployment in the real world. Constrained environments, included in the Internet of Things (IoT) framework, are especially challenging platforms. For instance, a naïve implementation of FrodoKEM-640 exceed the resources available on such platforms, hence requiring particular care and analysis.

A popular choice for implementing cryptography on embedded devices is the ARM Cortex-M4. We target this platform and showcase the fastest implementation of FrodoKEM-640 to date on it. The most delicate and resource intense operations are the expansion of a public pseudorandom matrix $\mathbf{A}$ from a seed and various matrix multiplications by smaller matrices. We exploit a combination of on-the-fly expansion of $\mathbf{A}$, originally proposed in the specifications [10], with a particular set of instructions available in our target platform through the Digital Signal Processing (DSP) extension.

These instructions fall into the Single Instruction, Multiple Data (SIMD) paradigm. As the ARM Cortex-M4 is a 32-bit architecture, it can hold up to 32 bits in each internal register.

Conveniently, FrodoKEM-640 is formed of matrices defined over $\mathbb{Z}_q$ with $q = 2^{15}$, which we embed in $\mathbb{Z}_{2^{16}}$ for convenience of computation. This implies that each element can be stored as a halfword in a register, and this is where SIMD instructions turn out to be most useful: when four values are correctly stored in the four halves of two registers, it is possible to operate on them in parallel with a single instruction. We will make a particular heavy use of the `smlad` instruction, which multiplies corresponding halfwords from two registers, adds them together, accumulates the result to a third register and stores the final output in a fourth one, all in one instruction.

Furthermore, we analyse a suit of algorithms and memory layouts of matrices: with which function **A** is generated, either cSHAKE128 or AES128, and whether it is involved in a multiplication as a left or as a right operand are all slightly different variants which require different optimisations.

Despite our carefully tailored optimisations, however, performance of matrix multiplications involving **A** is still dominated by its generation. A fine grained benchmark does indeed show that the greatest number of cycles is spent to perform AES128 and cSHAKE128. We analyse the rationale behind the choice of such cryptographically secure PseudoRandom Number Generators (PRNGs), and conclude that they are over-conservative for the task of generating a public matrix from a public seed. We therefore suggest to use a different, non-cryptographic PRNG in order to speed up the generation of **A**. We choose to implement FrodoKEM-640 using the PRNG xoshiro128**, which produces high quality pseudorandomness at a fraction of the cycle count. We show how to embed it in FrodoKEM-640 and benchmark it against the above two PRNGs.

Our contributions. We improve multiplication to the right of **A** by up to 62.8%, and by up to 78.9% when multiplication is to the left of **A**, compared to reference. We achieve such results by carefully analysing the layout in memory of small matrices, and how portions of **A** are generated and stored. We coded tailored algorithms for each situation, while having the use of SIMD instruction as a common design rationale for all our implementations. We also applied some of our algorithms to multiplication between "small" matrices, i.e. when **A** is not an operand: even in these cases we obtained improvements up to 78.9%, compared to reference.

With faster multiplication routines, generating the matrix **A** became even more dominant than before. For instance, the number of cycles at a clock frequency of 24 MHz spent to generate **A** account for the 86% of the whole execution of FrodoKEM-640: the remaining 14% can be as optimised as possible, but the difference will be barely noticeable anyway. These cryptographically secure PRNGs are not suitable, we instead suggest the usage of xoshiro128**. Generation of **A** suddenly becomes one of the cheapest operations in FrodoKEM-640, as it gets reduced by up to 96%.

We will soon release all our implementations to the public domain.

Related works. Howe et al. [3] recently implemented FrodoKEM-640 on the same target platform, yet did not use SIMD techniques. Secondly the PQM4 project [5], which we partially used as the backbone of our implementation, is a unified framework where to evaluate post-quantum candidates on an ARM Cortex-M4. We dedicate part of Section 5 to extensively compare our work with these two.

Very recently, Kannwischer et al. [4] implemented a plethora of schemes submitted to the NIST standardisation effort on an ARM Cortex-M4. The common denominator of all schemes is that their use of polynomials in $\mathbb{Z}_{2^m}[x]$. Kannwischer et al. created a tool which automatically explores different divide-and-conquer multiplication approaches and generates assembly code for these different algorithms. The DSP assembly instructions are also adopted.

An implementation for the ARM Cortex-M4 of a lattice-based scheme using different moduli has also been implemented: Alkim et al. [1] show how NewHope does benefit from an optimised implementation.

Structure of the paper. We introduce FrodoKEM-640 and the ARM Cortex-M4 in Section 2. Our optimisations are extensively described in Section 3. Then we discuss the proposal of using a non-cryptographically secure PRNG in Section 4, where we also introduce xoshiro128**. We finally combine everything together in Section 5, where we first describe in details several metrics we benchmarked our code with, then we show how our implementations in different contexts perform, and we conclude by comparing them with the aforementioned relevant previous works. We draw conclusions in Section 6.

2 Preliminaries

We describe our target algorithm, FrodoKEM-640, and some characteristics of the target platform, an ARM Cortex-M4 core. The specification of FrodoKEM-640 [10] provides two parameter sets, targeting different levels of security. We study embedded devices used in the IoT ecosystem, therefore we focus on the parameter set which targets NIST level 1 (matching or exceeding the brute-force security of AES128), described in Section 2.2.

The ARM Cortex-M4 core is a popular choice for microcontroller usage and has become a representative platform to benchmark cryptographic application for usage in the IoT ([1,3,4,5]). We target the ARM Cortex-M4 core as well to allow for easy comparison against previous applied cryptographic research, and we discuss it in Section 2.3.

2.1 Notation

We denote vectors and matrices by lowercase boldface letters and uppercase boldface letters, respectively. We use subscript notation to access them, e.g. v_i is the ith element of vector $\mathbf{v}$ and $M_{i,j}$ is the element on row i and column j of matrix $\mathbf{M}$. Concatenation of bit strings, including binary representation of vectors, matrices and integers, is denoted by $\|$.

In the official implementation of FrodoKEM-640, matrices are stored and operated upon as arrays. The convention used to linearise a two-dimensional structure like a matrix into one dimension has a huge impact on performance on embedded devices. It changes which elements are adjacent in memory, affecting how they are loaded. There is no universal convention which the matrices in FrodoKEM-640 follow, rather it comes down to convenience on a case-by-case basis. Throughout this work, we use teletype font to denote arrays obtained from matrices, e.g. $\mathtt{a}$ is the array corresponding to matrix $\mathbf{A}$. Indexes start from 0, and we will make use of pointer arithmetic notation derived from the C programming language to make notation easier when handling arrays. If an $n \times n$ matrix $\mathbf{A}$ is stored in $\mathtt{a}$ row-wise, for instance, then $\mathtt{a} + n$ denotes a pointer to the first element of the second row of $\mathbf{A}$, as it lies n positions away from the base address (always pointing to the top-left element of a matrix: $A_{0,0}$).

Algorithm 1 FrodoKEM-640.KeyGen

Input: None.
Output: Key pair $pk = (\text{seed}_{\mathbf{A}}, \mathbf{B}) \in \{0,1\}^{128} \times \mathbb{Z}_q^{n \times \overline{n}}$ and $sk = (\mathbf{s}, \mathbf{S}) \in \{0,1\}^{128} \times \mathbb{Z}_q^{n \times \overline{n}}$.
Parameters: $n = 640$, $\overline{n} = 8$, $q = 2^{15}$.

```
1: $\mathbf{s}, \text{seed}_{\mathbf{E}}, \mathbf{z} \xleftarrow{\$} \{0,1\}^{128}$
2: $\text{seed}_{\mathbf{A}} \leftarrow \text{cSHAKE128}(\mathbf{z})$
3: $\mathbf{A} \leftarrow \text{Frodo.Gen}(\text{seed}_{\mathbf{A}})$
4: $\mathbf{S}, \mathbf{E} \leftarrow \text{Frodo.Sample}(\text{seed}_{\mathbf{E}})$
5: $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E} \pmod{q}$
6: return public key $pk = (\text{seed}_{\mathbf{A}}, \mathbf{B})$ and secret key $sk = (\mathbf{s}, \mathbf{S})$
```

Algorithm 2 FrodoKEM-640.Encaps

Input: Public key $pk = (\text{seed}_{\mathbf{A}}, \mathbf{B}) \in \{0,1\}^{128} \times \mathbb{Z}_q^{n \times \overline{n}}$.
Output: Ciphertext $c = (\mathbf{B}', \mathbf{C}, \mathbf{d}) \in \mathbb{Z}_q^{\overline{m} \times n} \times \mathbb{Z}_q^{\overline{m} \times \overline{n}} \times \{0,1\}^{128}$ and shared secret $\mathbf{ss} \in \{0,1\}^{128}$.
Parameters: $n = 640$, $\overline{n} = 8$, $\overline{m} = 8$, $q = 2^{15}$.

```
 1: $\mu \xleftarrow{\$} \{0,1\}^{128}$
 2: $\text{seed}_{\mathbf{E}'}, \mathbf{k}, \mathbf{d} \leftarrow \text{cSHAKE128}(pk \| \mu)$
 3: $\mathbf{S}', \mathbf{E}' \leftarrow \text{Frodo.Sample}(\text{seed}_{\mathbf{E}'})$
 4: $\mathbf{A} \leftarrow \text{Frodo.Gen}(\text{seed}_{\mathbf{A}})$
 5: $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}' \pmod{q}$
 6: $\mathbf{E}'' \leftarrow \text{Frodo.Sample}(\text{seed}_{\mathbf{E}'})$
 7: $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' \pmod{q}$
 8: $\mathbf{C} \leftarrow \mathbf{V} + \text{Frodo.Encode}(\mu) \pmod{q}$
 9: $\mathbf{ss} \leftarrow \text{cSHAKE128}(\mathbf{B}' \| \mathbf{C} \| \mathbf{k} \| \mathbf{d})$
10: return Ciphertext $c = (\mathbf{B}', \mathbf{C}, \mathbf{d})$ and shared secret $\mathbf{ss}$
```

2.2 Details of Frodo

FrodoKEM-640 is a key encapsulation mechanism whose security is based on the LWE problem [11]. Intuitively, security is derived from the hardness of finding solutions to systems of linear equations which are perturbed by small amounts of additive noise, typically from a distribution close to Gaussian. Matrix multiplication is therefore the main operation, which becomes problematic when the matrix sizes become prohibitively large for embedded devices. All operations are performed modulo $q = 2^D$. In the parameter set we target $D = 15$ and we work with residues in $\mathbb{Z}_{2^{15}}$. In practice it is more efficient to use a redundant representation and work with residues in $\mathbb{Z}_{2^{16}}$ (e.g., the usual 16-bit datatypes). Converting from the redundant representation can be done by simply ignoring the most significant bit. The only other relevant parameters we make use of in this work are $n = 640$ and $\overline{n} = \overline{m} = 8$, which will determine the sizes of all handled matrices.

Algorithms 1, 2 and 3 describe the functionality of FrodoKEM-640, almost exactly as described in the specification [10]. Whenever something is drawn randomly, we assume a source of randomness is present on the target device. We use the Random Number Generator (RNG) built into our setup, see Section 2.3 for more details. The function cSHAKE128 is used multiple times to expand true randomness into longer pseudorandom sequences, which can be interpreted as matrices or bitstrings depending on the requirements. The only other discrepancy between Algorithms 1,2 and 3 and the official specification is that the latter includes two additional functions called Frodo.Pack/Unpack, which turn matrices into bitstrings and vice-versa. We omit them as they are irrelevant for our purposes.

Algorithm 3 FrodoKEM-640.Decaps

Input: Ciphertext $c = (\mathbf{B}', \mathbf{C}, \mathbf{d}) \in \mathbb{Z}_q^{\overline{m}\times n} \times \mathbb{Z}_q^{\overline{m}\times\overline{n}} \times \{0,1\}^{128}$ and secret key $sk = (\mathbf{s}, \mathbf{S}) \in \{0,1\}^{128} \times \mathbb{Z}_q^{n\times\overline{n}}$.
Output: Shared secret $\mathbf{ss} \in \{0,1\}^{128}$.
Parameters: $n = 640$, $\overline{n} = 8$, $\overline{m} = 8$, $q = 2^{15}$.

```
1:  $\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S} \pmod q$
2:  $\mu' \leftarrow$ Frodo.Decode($\mathbf{M}$)
3:  $\text{seed}_{\mathbf{E}'}, \mathbf{k}', \mathbf{d}' \leftarrow$ cSHAKE128($pk \| \mu'$)
4:  $\mathbf{S}', \mathbf{E}' \leftarrow$ Frodo.Sample($\text{seed}_{\mathbf{E}'}$)
5:  $\mathbf{A} \leftarrow$ Frodo.Gen($\text{seed}_{\mathbf{A}}$)
6:  $\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}' \pmod q$
7:  $\mathbf{E}'' \leftarrow$ Frodo.Sample($\text{seed}_{\mathbf{E}'}$)
8:  $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}'' \pmod q$
9:  $\mathbf{C}' \leftarrow \mathbf{V} +$ Frodo.Encode($\mu'$) $\pmod q$
10: if $\mathbf{B}' = \mathbf{B}''$ and $\mathbf{C} = \mathbf{C}'$ and $\mathbf{d} = \mathbf{d}$ then
11:     return $\mathbf{ss} \leftarrow$ cSHAKE128($\mathbf{B}' \| \mathbf{C} \| \mathbf{k}' \| \mathbf{d}'$)
12: else
13:     return $\mathbf{ss} \leftarrow$ cSHAKE128($\mathbf{B}' \| \mathbf{C} \| \mathbf{s} \| \mathbf{d}'$)
```

Algorithm 4 Frodo.Gen-AES128

Input: Seed $\text{seed}_{\mathbf{A}} \in \{0,1\}^{128}$.
Output: Pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$.
Parameters: $n = 640$, $q = 2^{15}$.

```
1: for $0 \le i < n$ do
2:     for $0 \le j < n$, $j \leftarrow j + 8$ do
3:         $\mathbf{p} \leftarrow i \| j \| 0 \| \ldots \| 0 \in \{0,1\}^{128}$
4:         $c_{i,j} \| \ldots \| c_{i,j+7} \leftarrow \text{AES128}_{\text{seed}_{\mathbf{A}}}(\mathbf{p})$ where $c_{i,k} \in \{0,1\}^{16}$
5:         for $0 \le k < 8$ do
6:             $A_{i,j+k} \leftarrow c_{i,j+k} \pmod q$
7: return $\mathbf{A}$
```

The two functions Frodo.Encode and Frodo.Decode turn a bit string into a matrix with coefficients in $\mathbb{Z}_q$ in such a way that the original bits can be recovered even if some bounded noise is introduced. We refer the interested reader to the original publication for more information on this procedure [10].

Secret and error matrices are generated by expanding a seed into a pseudorandom sequence with cSHAKE128, and then by applying inversion sampling to this sequence. This way, their elements follow a Gaussian-like distribution called χ, whose outputs are smaller than 11 in absolute value. Since this part is not central to our work, we hide all the details behind the Frodo.Sample function.

Seed expansion. Of particular interest to us is the Frodo.Gen function in Algorithms 1, 2 and 3, which expands a seed to generate $\mathbf{A}$. $\mathbf{A}$ is a matrix of size $n \times n$, hence requiring a total of $16n^2$ bits, i.e. 800 KB for $n = 640$. As we will describe in Section 2.3, our target device does not have enough memory to fully generate and load $\mathbf{A}$, which is why the FrodoKEM-640 team implemented an on-the-fly generate-and-multiply routine to be used in constrained devices. The function Frodo.Gen can be instantiated with two different PRNGs: AES128 or cSHAKE128. It is very important to analyse each case separately as elements of $\mathbf{A}$ are filled in a different orders, making any optimisation dependent on this choice.

Algorithm 5 Frodo.Gen-cSHAKE128

Input: Seed $\text{seed}_{\mathbf{A}} \in \{0,1\}^{128}$.
Output: Pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.
Parameters: $n = 640$, $q = 2^{15}$.

```
1: for $0 \leq i < n$ do
2:     $c_{i,1}\| \dots \| c_{i,n} \leftarrow \text{cSHAKE128}(\text{seed}_{\mathbf{A}} \| (2^8 + i))$ where $c_{i,j} \in \{0,1\}^{16}$
3:     for $0 \leq j < n$ do
4:         $A_{i,j} \leftarrow c_{i,j} \pmod q$
5: return $\mathbf{A}$
```

Algorithm 4 uses AES128 and works as follows. Two indexes i, j are converted to binary and padded to form a 128-bit plaintext. AES128 is then applied with $\text{seed}_{\mathbf{A}}$ as a key and the resulting 128-bit ciphertext is interpreted as eight 16-bit numbers modulo q and stored in $A_{i,j}, \dots, A_{i,j+7}$. This is repeated for all rows, index by i, and every eight columns, index by j.

The second possibility is using cSHAKE128 and is described in Algoritm 5. In this case cSHAKE128 is initialised with $\text{seed}_{\mathbf{A}}$ and with a customisation value which only depends on the row index, and produces n 16-bit numbers modulo q which are stored as one full row of $\mathbf{A}$.

Algorithms 4 and 5 are different and not compatible with one another. A fair comparison in performance of the two algorithms is also quite hard to establish due to the large number of possible implementations of the two PRNGs. On the one hand, AES128 is exceptionally fast when the underlying platform has dedicated instructions to run it, e.g. AES-NI instructions on Intel platforms. When they are not available, however, cSHAKE128 seems to offer better performance according to the Frodo specification [10]. For this work, we choose a highly optimised implementation of cSHAKE128 in ARM assembly made by the KECCAK team, also used in PQM4 [5] and by Howe et al. [3]. With the latter we also share the implementation of AES128, proposed by Schwabe and Stoffelen [12].

Remark. We noticed that there are ways of speeding up the computation of AES128 and cSHAKE128 by exploiting the fact that the inputs are partly fixed for each iteration. By reformatting how bitstrings are given, it is possible to compute and store parts of the internal state. For example, the plaintext of AES128 is mostly formed of zeros, hence portions of the first and second rounds could be precomputed and reused at every iteration. Similarly, since $\text{seed}_{\mathbf{A}}$ is fixed, the first few absorption of cSHAKE128 can be computed in advance. We do not explore these ideas further, as doing so would mean to enter in the details of how the two PRNGs are implemented. We instead decided to use them as black boxes and focus on the operations in FrodoKEM-640.

2.3 Fast Arithmetic on an ARM Cortex-M4 Core

In recent works the ARM Cortex-M4 core has been considered a representative platform when benchmarking post-quantum primitives targeting embedded applications [1,3,4,5]. This platform has a word-size of 32 bits. The Cortex-M4 has specific instructions which can work on two half-words (of size 16 bits) in parallel following the single instruction, multiple data (SIMD) paradigm. This has recently been explored by Kannwischer et al. [4] to optimise multiplication of polynomials in $\mathbb{Z}_{2^m}[x]$. Let us recall the most relevant instructions we use in the implementation.

- `ldmia` loads multiple (up to eight) full-word values from consecutive memory into the corresponding amount of registers and, optionally, updates the pointer to the memory accordingly.
- `smlad` multiplies four half-word sized values stored in two word registers and adds them to an accumulator. More specifically,

  $$\texttt{smlad} \quad d, a, b, c$$

  computes

  $$d = \left(a \bmod 2^{16}\right) \cdot \left(b \bmod 2^{16}\right) + \left\lfloor \frac{a}{2^{16}} \right\rfloor \cdot \left\lfloor \frac{b}{2^{16}} \right\rfloor + c \bmod 2^{32}.$$

- `ldrh` and `strh` are useful for loading and storing half-word sized values from memory in one instruction, without needing to load a full-word combined with masking or shifting.
- `bfi` copies a bit field from one register into another one. This is useful for merging two half-word values for using the aforementioned SIMD instruction.

Given the amount of available registers these instructions allow multiplying two matrices five values at a time, in a pretty straightforward way. This is under the assumption that the left matrix is given in row-major order and the right matrix in column-major order such that they can both be accessed linearly. As outlined in Section 3 this is not always the case, hence the need for dedicated subroutines accessing non-adjacent portions of a linearised matrix.

The ARM Cortex-M4 in our setup is mounted on a STM32F407 board, equipped with 1 MB of flash and 192 KB of memory. The default clock frequency is 168 MHz. The board is equipped with a True Random Number Generator (TRNG) that derives entropy from analog noise. A Linear Feedback Shift Register (LFSR) is seeded with the noise and with a dedicated clock. Its output is stored in the RNG_DR register, which is read when randomness is needed. Checks to verify the absence of abnormalities in both the noise-derived seed and clock are present too.

The matrix $\mathbf{A}$ is the main obstacle to any implementation of FrodoKEM-640 on embedded devices. Apart from being resource intensive to generate, it also simply does not fit in memory. Our platform is ideal to test on-the-fly solutions, because we are forced to generate $\mathbf{A}$ in chunks.

3 FrodoKEM-640 Optimisations

Despite the clear differences outlined in Algorithms 4 and 5, AES128 and cSHAKE128 do show a certain degree of similarity, allowing us to write some common subroutines in ARM assembly to be reused in both cases. They will also turn useful when we describe how we optimised other matrix multiplications other than those by $\mathbf{A}$, namely the ones between $n \times \overline{n}$ matrices in Encaps and Decaps.

We opted for addressing different multiplications in different sections, rather than dividing the discussion between PRNGs. Therefore, Section 3.1 describes the common subroutines, which are used in Sections 3.2 and 3.3 that describe $\mathbf{AS}$ and $\mathbf{S'A}$, respectively. We conclude with multiplications between small matrices in Section 3.4.

3.1 Common Subroutines

Both AES128 and cSHAKE128 can generate a full row of $\mathbf{A}$, although in slightly different ways. This perfectly fits the case when $\mathbf{A}$ is the left operand in a multiplication, because it implies that some of its rows are fully stored in adjacent memory locations, and happens during KeyGen only. Even more conveniently, $\mathbf{S}$ is stored column-wise, effectively opening the way to an inner product function based on SIMD instructions.

We call our first subroutine `ip_n`, as it computes the inner product between two vectors of length n, which we assume have their elements stored in adjacent locations in memory. The inner functionality and a description of all instructions follow.

```
                ip_n
1   mov %[r], #0
2   ldmia %[s]!, {r0, r1, r2, r3, r4}
3   ldmia %[a]!, {r5, r6, r7, r8, r9}
4   smlad %[r], r0, r5, %[r]
5   smlad %[r], r1, r6, %[r]
6   smlad %[r], r2, r7, %[r]
7   smlad %[r], r3, r8, %[r]
8   smlad %[r], r4, r9, %[r]
9   (lines 4-10 are repeated 64 times)
```

1 The register holding the final result, named `%[r]`, is initialised to zero.

2-3 The instruction `ldmia` is used to load multiple registers at once. We load five registers, i.e. 20 bytes, exploiting the fact that all elements we are interested in are adjacent in memory. The exclamation mark after the address registers indicates that pointers, stored in registers `%[s]` and `%[a]`, are also updated, hence no other instructions will be needed in the next iteration.

4-8 Five SIMD instructions are everything we need to perform 10 multiplications and 10 additions (accumulations) on the output register `%[r]`.

9 Since the above lines compute over 10 elements at the time, we need to repeat them $n/10 = 64$ times.

The function `ip_n` crucially relies on the vectors to be multiplied to have adjacent positions in memory such that their addresses can be loaded once and then updated directly by the loading instruction. Unfortunately such an optimal scenario only happens when $\mathbf{AS}$ is performed during KeyGen. Apart from there, $\mathbf{A}$ has to be generated during Encaps and for the re-encapsulation part of Decaps. In these cases it is multiplied to the left by the matrix $\mathbf{S}'$, while still being generated row-wise for consistency and correctness. On the bright side, $\mathbf{S}'$ is generated row-wise.

The function we developed for the matrix multiplication $\mathbf{S}'\mathbf{A}$ is denoted `row_by_chunk` and is depicted in Figure 1. Four registers, are loaded with eight consecutive elements in a column of $\mathbf{A}$ (we represent only four of them in Figure 1 for the sake of compactness). As this procedure differs between cSHAKE128 and AES128, we defer the details and the adopted instructions for later, while for now referring to those elements as the vector `a_temp`. Then, for every row of $\mathbf{S}'$, the corresponding eight elements are loaded, SIMD-multiplied with `a_temp` and the result is accumulated in the corresponding positions of the output matrix. The colour code in Figure 1 visualises them. The code and a line by line description follow.

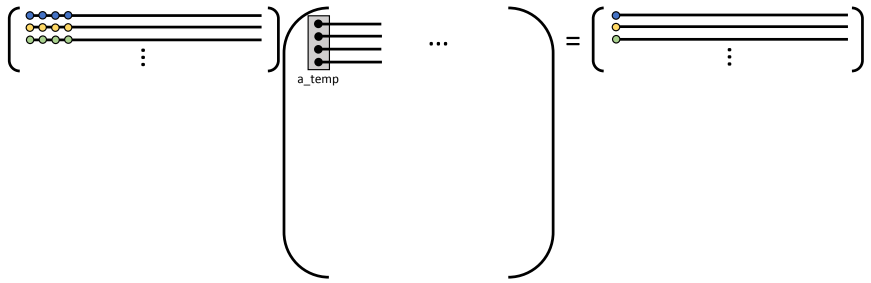

Fig. 1: Visualisation of the `row_by_chunk` function. Circles refer to elements, where for compactness we depicted `a_temp` holding four elements, while in reality it holds eight. Lines show how elements are disposed in memory. Finally, the colour code simply highlights how multiplication by `a_temp` works.

`row_by_chunk`

```
1   mov r0, %[s]
2   mov r9, %[o]
3   ldmia r0, {r5,r6,r7,r8}
4   ldrh r10, [r9, #0]
5   smlad r10, r1, r5, r10
6   smlad r10, r2, r6, r10
7   smlad r10, r3, r7, r10
8   smlad r10, r4, r8, r10
9   strh r10, [r9, #0]
10  add r0, r0, #1280
11  (lines 3-10 are repeated 8 times)
```

1 The address of the matrix $\mathbf{S}'$ is held in `%[s]`.

2 The address of the matrix $\mathbf{B}'$ is held in `%[o]`. We use hard-coded offsets to access the correct position: each element in a column of $\mathbf{B}'$ is $2n = 1280$ bytes away from the top element in the same column.

3 Four registers are filled with eight elements of $\mathbf{S}'$. Note that we do not use the exclamation mark because we manually modify the address to access non adjacent memory locations (line 10).

4 One element of $\mathbf{B}'$ is loaded: this is a 16-bit value, hence the instruction to load a halfword is used.

5-8 Four SIMD instructions are used to perform 8 multiplications and 8 additions (accumulations). We assume the elements of `a_temp` are stored in registers `r1` up to `r4`.

9 The partially updated element of $\mathbf{B}'$ is stored back in place.

10 The address of $\mathbf{S}'$ is shifted by $2n = 1280$ bytes, therefore it now points to the second row of $\mathbf{S}'$.

11 Lines 3 to 8 are repeated a total of $\overline{n} = 8$ times, hence partially updating a full column of $\mathbf{B}'$. The only caveat is that after four columns, the register `r9` holding the pointer to $\mathbf{B}'$ must be updated by $4 \cdot 2n = 5120$ bytes with an extra `add` instruction, otherwise the offset would exceed the maximum allowed by the architecture.

Algorithm 6 AS multiplication

Input: Seed $\text{seed}_{\mathbf{A}} \in \{0,1\}^{128}$, output vector $\texttt{b} \in \mathbb{Z}_q^{n\overline{n}}$ initialised with error matrix $\mathbf{E}$, and secret vector $\texttt{s} \in \mathbb{Z}_q^{n\overline{n}}$.
Output: $\texttt{b} \leftarrow \texttt{b} + \mathbf{AS}$.
Parameters: $n = 640$, $\overline{n} = 8$, $q = 2^{15}$.

```
 1: $\texttt{a\_rows} \leftarrow \{0\}^{4n}$
 2: for $0 \le i < n, i \leftarrow i + 4$ do
 3:     for $0 \le j < 4$ do
 4:         if PRNG = AES128 then
 5:             for $0 \le k < n, k \leftarrow k + 8$ do
 6:                 $\mathbf{p} \leftarrow (i + j) \| k \| 0 \| \dots \| 0 \in \{0,1\}^{128}$
 7:                 $\texttt{a\_rows} + j \cdot n + k \leftarrow \text{AES128}_{\text{seed}_{\mathbf{A}}}(\mathbf{p})$
 8:         else if PRNG = cSHAKE128 then
 9:             $\texttt{a\_rows} + j \cdot n \leftarrow \text{cSHAKE128}(\text{seed}_{\mathbf{A}} \| (2^8 + i + j))$
10:     for $0 \le k < \overline{n}$ do
11:         for $0 \le j < 4$ do                                        ▷ Unrolled loop
12:             $\texttt{b}[k + (i + j)\overline{n}] \leftarrow \texttt{b}[k + (i + j)\overline{n}] + \texttt{ip\_n}(\texttt{s} + k \cdot n, \texttt{a\_rows} + j \cdot n)$
```

3.2 Optimising the Matrix Multiplication AS

This setting is straight-forward and we can tackle it independently from the PRNG adopted. The matrix $\mathbf{A}$ is the left operand which we store row-wise. We can apply cSHAKE128 as specified in Algorithm 5 whenever we need a row, while in the case of AES128 we can fix the index i and run the j-loop from Algorithm 4. Once one or more rows are generated, we can simply run the `ip_n` subroutine directly.

In Algorithm 6 four rows of $\mathbf{A}$ are genearted at-a-time, which works slightly different depending if AES128 or cSHAKE128 is being used. In the first case, an extra loop over the columns, eight by eight, is needed; while the latter PRNG generates full rows straight away. Next, we multiply each of the generated rows by all columns of $\mathbf{S}$ and accumulate the output vector. Note that `ip_n` returns a value, which is added in to `b`.

3.3 Optimising the Matrix Multiplication S′A

In the setting of the matrix multiplication $\mathbf{S'A}$ it is crucial to understand how $\mathbf{A}$ is placed in memory and how AES128 differs from cSHAKE128. The `row_by_chunk` subroutine assumes eight consecutive elements in a column of $\mathbf{A}$, which do need to be adjacent in memory, are stored in the `a_temp` array. Next, this is multiplied by corresponding portions of $\mathbf{S'}$, accumulated and stored directly in memory. We denote the portion of $\mathbf{A}$ being generated each time as `a_cols`, which is the counterpart of `a_rows` from Section 3.2.

Figure 2 visualises the process in both the AES128 (left) and the cSHAKE128 (right) cases. In the former, a $n \times 8$ submatrix is generated by performing the whole i-loop with a fixed j in Algorithm 4. Performing the same operation with an incremented j yields to the next submatrix, hence shifting `a_cols` to the right by 8 positions. Since cSHAKE128 can only generate full rows `a_cols` is simply filled with the first eight of them (only four are shown for compactness). Dashed arrows show in which direction `a_cols` moves in both cases. Note that Figure 2 represents only four elements in `a_temp` for compactness.

Once `a_cols` is generated and stored, values corresponding to `a_temp` have to be loaded to be used by `row_by_chunk`. This is where cSHAKE128 and AES128 differ: `a_temp` contains eight

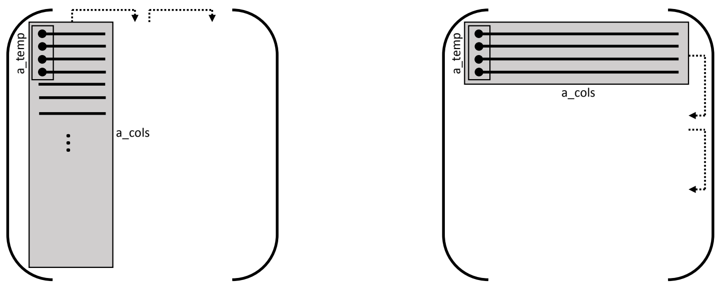

Fig. 2: The matrix **A** as generated and accessed by AES128 (left) and cSHAKE128 (right). Dashed arrows show in which order `a_cols` moves to different portions of **A**.

consecutive values in one column, and moves to the next column for the next iteration. Once there are no more columns available in `a_cols`, `a_temp` jumps to the subsequent eight rows. Such a wrap around happens after 8 columns for AES128 and n columns for cSHAKE128. In particular, in the former case `a_temp` jumps more often, but the next eight rows are present in memory; in the latter `a_temp` covers the full eight rows present in memory, then `a_cols` has to be filled with other eight rows. The following ARM assembly snippet illustrates how we dealt with such a discrepancy.

load_a_temp

```
__________aes_________|_________cshake________
ldrh r1, [%[a], #0]   | ldrh r1, [%[a], #0]
ldrh r5, [%[a], #16]  | ldrh r5, [%[a], #1280]
bfi r1, r5, #16, #16  | bfi r1, r5, #16, #16
ldrh r2, [%[a], #32]  | ldrh r2, [%[a], #2560]
ldrh r5, [%[a], #48]  | ldrh r5, [%[a], #3840]
bfi r2, r5, #16, #16  | bfi r2, r5, #16, #16
                      | add %[a], %[a], #5120
ldrh r3, [%[a], #64]  | ldrh r3, [%[a], #0]
ldrh r5, [%[a], #80]  | ldrh r5, [%[a], #1280]
bfi r3, r5, #16, #16  | bfi r3, r5, #16, #16
ldrh r4, [%[a], #96]  | ldrh r4, [%[a], #2560]
ldrh r5, [%[a], #112] | ldrh r5, [%[a], #3840]
bfi r4, r5, #16, #16  | bfi r4, r5, #16, #16
```

The code snippet `load_a_temp` reports both AES128 (left) and cSHAKE128 (right) versions, with differences highlighted in red. Starting from the address stored in `%[a]`, even positions are loaded to the bottom half of each register, while odd ones are first loaded to the bottom half of the temporary register `r5` and subsequently moved to the top half of the designated register (`r1` up to `r4`) thanks to the `bfi` instruction. Such a procedure is common to both halves of the snippet.

What differs is the offset when loading values from `%[a]`. In the case of AES128 (left) the offset among values in the same column is a multiple of 16 bytes because `a_cols` is a $n \times 8$ submatrix of **A**. Instead, elements in the same column of `a_cols` when cSHAKE128 is used are $2n = 1280$ bytes apart from each other, hence offsets in the right half of the snippet are multiples of 1280 (cf. Figure 2). This introduces an extra complication: offsets are not allowed

Algorithm 7 $\mathbf{S'A}$ multiplication

Input: Seed $\text{seed}_{\mathbf{A}} \in \{0,1\}^{128}$, output vector $\texttt{bp} \in \mathbb{Z}_q^{n\overline{n}}$ initialised with error matrix $\mathbf{E}'$, and secret vector $\texttt{sp} \in \mathbb{Z}_q^{n\overline{n}}$.
Output: $\texttt{bp} \leftarrow \texttt{bp} + \mathbf{S'A}$.
Parameters: $n = 640$, $\overline{n} = 8$, $q = 2^{15}$.

```
1:  $\texttt{a\_cols} \leftarrow \{0\}^{8n}$
2:  for $0 \le k < n, k \leftarrow k + 8$ do
3:      if PRNG = AES128 then
4:          for $0 \le i < n$ do
5:              $\mathbf{p} \leftarrow i \| k \| 0 \dots 0 \in \{0,1\}^{128}$
6:              $\texttt{a\_cols} + 8 \cdot i \leftarrow \text{AES128}_{\text{seed}_{\mathbf{A}}}(\mathbf{p})$
7:          for $0 \le i < n, i \leftarrow i + 8$ do
8:              for $0 \le j < 8$ do
9:                  $\texttt{a\_temp} \leftarrow \texttt{load\_a\_temp}(\texttt{a\_cols} + 8 \cdot i + j)$
10:                 $\texttt{row\_by\_chunk}(\texttt{sp} + i, \texttt{a\_temp}, \texttt{bp} + k + j)$
11:     else if PRNG = cSHAKE128 then
12:         for $0 \le j < 8$ do                                        ▷ Unrolled loop
13:             $\texttt{a\_cols} + j \cdot n \leftarrow \text{cSHAKE128}(\text{seed}_{\mathbf{A}} \| (2^8 + k + j))$
14:         for $0 \le i < n$ do
15:             $\texttt{a\_temp} \leftarrow \texttt{load\_a\_temp}(\texttt{a\_cols} + i)$
16:             $\texttt{row\_by\_chunk}(\texttt{sp} + k, \texttt{a\_temp}, \texttt{bp} + i)$
```

to exceed 4095, hence we have to spend an extra `add` instruction, highlighted in red, to make the address in `%[a]` point to the first position of the fourth row.

Algorithm 7 incorporates both AES128 and cSHAKE128 variants of the $\mathbf{S'A}$ multiplication. We denoted by `load_a_temp_aes` and `load_a_temp_cshake` the two halves of the `load_a_temp` snippet. Once the appropriate values are loaded, the `row_by_chunk` function is applied. Note that it does not return any value, as the memory location of `bp` is an input, hence results are immediately stored back. Loop structures make sure that submatrices are traversed in the correct order, depending on the situations.

3.4 Small Matrix Multiplication Optimisations

Multiplications involving $\mathbf{A}$ are by far the most time consuming. There are however multiplications between smaller matrices ($\overline{n} \times n$ and vice-versa) which take a much smaller portion of the overall computation time, but can still benefit from some of the subroutines outlined in Section 3.1.

During Decaps, the function `ip_n` can be used to efficiently perform $\mathbf{B'S}$, since both matrices are conveniently stored in memory. Differently than before, they are fully present in memory, and the matrix $\mathbf{B}'$ is not initialised to an error matrix because it comes from the ciphertext. The result of multiplication is therefore saved in a different vector and is later subtracted from $\mathbf{C}$ using a different function.

Algorithm 8 shows how to use `ip_n` for the $\mathbf{B'S}$ multiplication. Effectively we efficiently perform the inner product along the bigger dimension n, while the two smaller dimensions $\overline{n}$ are taken care in a loop and in an unrolled fashion.

Similarly, we can compute the function $\mathbf{V} \leftarrow \mathbf{S'B} + \mathbf{E}''$ in both Encaps and Decaps using the `row_by_chunk` function. In this case, $\mathbf{B}$ is a $n \times 8$ matrix stored row-wise, precisely like `a_cols` in the AES128 case because $\overline{n} = 8$. It is worth noticing that $\mathbf{B}$ is generated

Algorithm 8 $\mathbf{B'S}$ multiplication

Input: Uninitialised output vector `out` $\in \mathbb{Z}_q^{\overline{n}\overline{n}}$, vectors `bp`, `s` $\in \mathbb{Z}_q^{n\overline{n}}$ containing $\mathbf{B}'$ and $\mathbf{S}$, respectively.
Output: `out` $\leftarrow \mathbf{B'S}$.
Parameters: $n = 640$, $\overline{n} = 8$, $q = 2^{15}$.

```
1: for $0 \le i < \overline{n}$ do
2:     for $0 \le j < 8$ do                                ▷ Unrolled loop
3:         $\texttt{out}[i \cdot \overline{n} + j] \leftarrow \texttt{ip\_n}(\texttt{bp} + in, \texttt{s} + j \cdot n)$
```

Algorithm 9 $\mathbf{S'B}$ multiplication

Input: Output vector `v` $\in \mathbb{Z}_q^{\overline{n}\overline{n}}$ initialised with elements of $\mathbf{E}''$, vectors `b`, `sp` $\in \mathbb{Z}_q^{n\overline{n}}$ containing $\mathbf{B}$ and $\mathbf{S}'$, respectively.
Output: `v` $\leftarrow$ `v` $+ \mathbf{S'A}$.
Parameters: $n = 640$, $\overline{n} = 8$, $q = 2^{15}$.

```
1: for $0 \le i < n, i \leftarrow i + 8$ do
2:     for $0 \le j < \overline{n}$ do
3:         $\texttt{b\_temp} \leftarrow \texttt{load\_a\_temp\_aes}(\texttt{b} + 8i + j)$
4:         $\texttt{row\_by\_chunk}(\texttt{sp} + i, \texttt{b\_temp}, \texttt{v} + j)$
```

during KeyGen, being it part of the public key, and then used both in Encaps and Decaps as rightmost operand in matrix multiplications. Therefore a column-wise layout in memory would be more beneficial, but we do not explore this idea further being the impact on the overall computation time too marginal. Algorithm 9 shows the pseudocode, which is exactly the same as in lines 6 to 9 of Algorithm 7.

We postpone the showcase of performances our optimisations achieve to Section 5, where a unified view on benchmarks, together with comparisons with other relevant works, will be given. In the coming section, instead, we focus our attention on how $\mathbf{A}$ is generated. So far we have been compliant with the design choices in the specification of FrodoKEM-640 [10], and optimised the usage of AES128 and cSHAKE128 inside matrix multiplications. We are about to offer an alternative o them greatly outperforming both.

4 Faster PRNG for A: xoshiro128**

Generating the large matrix $\mathbf{A}$ is typically problematic on embedded devices. More generally, the competitiveness of FrodoKEM-640 on any device is hampered by the need to deal with big matrices rather than small polynomials, as happens in its ring and module variants.

A step in the right direction has already been taken with the design choice of expanding true randomness with a PRNG. When it comes to generating large matrices, both randomness cost and performance improve. Setting aside the unavoidable cost of generating a seed from random, the next question is which PRNG to choose. As we pointed out several times, this choice impacts on how portions of a matrix are actually generated, thus on performance.

To remedy the performance penalty incurred by FrodoKEM-640 for generating $\mathbf{A}$, we propose a third option for the PRNG in this section by challenging the need for a cryptographically secure PRNG to generate a public matrix. The purpose of cryptographically secure PRNGs is to provide streams of pseudorandom numbers achieving some form of security. When the seed of the sequence is secret, an adversary learning the first k bits should

Algorithm 10 xoshiro128**

Input: State $\mathtt{s} \in (\{0,1\}^{32})^4$.
Output: Pseudorandom number $r \in \{0,1\}^{32}$.
Parameters: None.

```
1: $r \leftarrow (((\mathtt{s}[0] \cdot 5) << 7) \vee ((\mathtt{s}[0] \cdot 5) >> 25)) \cdot 9$
2: $t \leftarrow \mathtt{s}[1] << 9$
3: $\mathtt{s}[2] \leftarrow \mathtt{s}[2] \oplus \mathtt{s}[0]$
4: $\mathtt{s}[3] \leftarrow \mathtt{s}[3] \oplus \mathtt{s}[1]$
5: $\mathtt{s}[1] \leftarrow \mathtt{s}[1] \oplus \mathtt{s}[2]$
6: $\mathtt{s}[0] \leftarrow \mathtt{s}[0] \oplus \mathtt{s}[3]$
7: $\mathtt{s}[2] \leftarrow \mathtt{s}[2] \oplus t$
8: $\mathtt{s}[3] \leftarrow (\mathtt{s}[3] << 11) \vee (\mathtt{s}[3] >> 21)$
9: **return** $r$
```

not be able to predict the $(k+1)$th bit. If the internal state is compromised, reverting the computation upstream until the seed is disclosed should also be infeasible.

Therefore, the whole point of a cryptographically secure PRNG is that, as long as something inside remains secret, be it the seed or some internal state, an adversary cannot tell the difference between the output stream and a truly random sequence, nor recover anything that would allow reconstructing part of the sequence. Crucially, disclosing the secret reveals the deterministic nature of the sequence, hence making any security notion useless.

For the above reasons, we suggest the usage of a non-cryptographic PRNG over a secure one for the generation of the public matrix $\mathbf{A}$, whose seed is part of the public key. This is advantageous because non-cryptographic PRNGs are usually faster and have smaller internal states, since they are only designed to achieve good statistical properties. The latter is still a stringent property of $\mathbf{A}$, as clear patterns and other generic statistical weaknesses could potentially weaken the underlying LWE problem.

4.1 Description of xoshiro128**

We propose to use xoshiro128** [2] as the designated PRNG for our implementation in FrodoKEM-640. There are several reasons behind this decision. As the name says, it has a 128 bit state, which is precisely the length of $\text{seed}_{\mathbf{A}}$. Moreover, the state is seen as an array of four 32 bit values, which matches the word size of the target architecture. However, larger versions achieving the same statistical properties are available [2].

Statistical quality and performance are the main advantages of xoshiro128**. The design is inspired by the xorshift family of PRNGs [9], which are linear functions and therefore do not always pass all statistical tests. To fix this issue a *scrambler* is used, that is to say a non-linear layer to avoid linear dependencies in output. The authors of xoshiro128** use a sequence of multiplication-rotation-multiplication operations called a $**$ scrambler, hence the name xoshiro128**. One word of the state is multiplied by constants of the form $2^s + 1$ and rotated. Constants are chosen to be efficiently implemented as one shift and one addition. Algorithm 10 shows the pseudocode of xoshiro128**, where $<<$ and $>>$ indicate left and right rotations, respectively. Firstly the output of the current iteration r is computed, then the internal state $\mathtt{s}$ is updated.

Outputs of xoshiro128** achieve very good statistical quality. According to the original publication [2], xoshiro128** passes the BigCrush test from the TestU01 suite [7]. It is in fact one of the fastest doing so. On top of that, the authors designed a test to discover statistical bias in the Hamming weight of w-bit words generated by a PRNG, showing that xoshiro128** and closely related PRNGs succeed.

4.2 Frodo and xoshiro128**, Love at First Sight

We used xoshiro128** to generate $\mathbf{A}$ inside Frodo. We left everything else as it was, including the fact that cSHAKE128 is used to generate seed$_\mathbf{A}$ and all secret and error matrices. For the latter, security of the PRNG is indeed mandatory, as the seeds are supposed to be kept secret. In the case of seed$_\mathbf{A}$, adopting a cryptographically secure PRNG fed with true randomness makes sure that tampering with the seed of $\mathbf{A}$ is not feasible, and that xoshiro128** is seeded correctly. Since an adversary does not choose nor can tamper with the seed, statistical quality harming the underlying LWE security is the only potential problem. Since xoshiro128** has very good statistical properties [2], no such issue holds.

We opted for changing the order in which $\mathbf{A}$ is stored in memory: instead of having row values adjacent in memory, we generate $\mathbf{A}$ column-wise. This is motivated by the fact that the former convention is only convenient in KeyGen, while being problematic for Encaps and Decaps. This way, the situation is reverted.

Column-wise layout in memory has another advantage: its symmetry with respect to the AES128 and cSHAKE128 generate $\mathbf{A}$. We can therefore simply swap the subroutines described in Section 3.1 for the multiplications by $\mathbf{A}$: `ip_n` can be used for $\mathbf{S}'\mathbf{A}$ and `row_by_chunk` can be used in $\mathbf{AS}$. The former returns a value, which is then added to the output matrix in the wrapping C code, but note that the latter stores results directly to memory. Therefore there is one caveat to take care of: `row_by_chunk` was originally designed to output $\mathbf{B}'$, which is saved in rows of length n, while here we want it to output $\mathbf{B}$, whose rows are of length $\overline{n} = 8$. A simple change of offsets in the loading and storing instructions is then enough to solve the issue.

We realised three implementations: a first one in portable C where $\mathbf{A}$ is fully pre-generated, an implementation in portable C where $\mathbf{A}$ is generated and multiplied on-the-fly which we use as our reference, and finally our optimised implementation.

In the next section we describe how we benchmarked our optimised xoshiro128** implementation only, alongside with reference and optimised implementations from Section 3. We provide several metrics to understand and, more importantly, contextualise results. Finally we compare our work with existing papers in the literature.

5 Results and Comparison with Previous Works

Benchmarks are never straightforward to carry out, and interpreting their results to derive recommendations is an even more delicate task. In this section we collect all performance results of our implementations, as well as compare them with relevant previous works.

As we mentioned in Section 2, the default clock frequency of our target development board is 168 MHz. However when the main interest is how the implementation of a specific function performs, i.e. how many clock cycles the *operations* in that function take, then it is useful to run benchmarks at a lower clock frequency to compensate for memory accesses. If there are any inside the benchmarked function, the final number of clock cycles computed at a normal frequency might not be representative, as a good part of them might have been simply wasted by the function waiting for memory to retrieve the queried values. Memory is indeed notoriously slower than the microprocessor. On the other hand, not contextualising

Function	cSHAKE128		AES128		xoshiro128**
	Ref [10]	**This work**	Ref [10]	**This work**	**This work**
KeyGen	99,762,353	88,288,589	105,892,559	95,593,272	14,205,132
Encaps	118,213,358	92,814,363	110,258,517	99,800,669	14,927,835
Decaps	118,686,081	93,776,021	110,699,504	100,7471,40	15,731,086

Table 1: Cycle counts averaged over 100 executions and obtained at 168 MHz.

benchmarks at a lower frequency is equally, if not more, misleading, and can yield to erroneous conclusions on the actual performance.

The STM32F407 development board offers a very neat and simple way of precisely computing cycle count, thanks to the Data Watchpoint and Trace (DWT) registers. We reset and read the DWT_CYCCNT register around functions to benchmark to have a confident measure of how many cycles they took. We use the default 168 MHz clock frequency for benchmarking time of algorithms, and the lower 24 MHz when benchmarking cycle count of internal operations. This seems to be a popular choice in the literature [3,4,5], thus making comparisons fairer.

We compare our implementations against the portable C implementation of FrodoKEM-640 denoted as "optimised" in the official specifications [10]. The "reference" implementation cannot possibly fit in our setup because **A** is fully generated, and it would occupy $2n^2 = 819,200$ bytes of memory, i.e. 800 KB against the 192 KB available. The "optimised" implementation, instead, generates and multiplies chunks of **A** on-the-fly. Since the latter is the only official implementation we can refer to for benchmarking purposes, we call it reference for the rest of the paper.

We use the same implementation of cSHAKE128 as in the PQM4 framework [5], which has been optimised in ARM assembly. Since AES128 is not part of the framework, we opted for the Schwabe and Stoffelen [12] implementation, also optimised for the ARM Cortex-M4 and used by Howe et al. [3] too. Finally, we deploy the implementation of xoshiro128** in C code, available at `http://xoshiro.di.unimi.it/xoshiro128starstar.c`.

5.1 Benchmarks at 168 MHz

We benchmark both the reference [10] and our optimised implementations of FrodoKEM-640 for all three PRNGs. We set the clock frequency to the default 168 MHz, and execute the whole protocol 100 times, measuring cycle counts of KeyGen, Encaps and Decaps each time. Results are shown in Table 1.

Overall, we improve the cycle counts of FrodoKEM-640 on the ARM Cortex-M4 by 18.4% when using cSHAKE128, and by 9.4% when using AES128. However, our optimised implementation using xoshiro128** is on average 84.3% faster than the above two, which is emblematic of how many cycles are spent in the generation of a publicly known matrix whose only goal is to look random.

We can fairly accurately estimate the time taken by FrodoKEM-640 in the three cases by dividing the total cycle count in Table 1 by $168 \cdot 10^6$ to obtain a time expressed in seconds. Our optimised implementations take around 1.64 seconds when cSHAKE128 is adopted, 1.76 second with AES128 and 0.27 seconds with xoshiro128**.

Function	cSHAKE128		AES128		xoshiro128**
	Ref [10]	**This work**	Ref [10]	**This work**	**This work**
AS	90,053,180	78,068,108	48,764,566	38,449,619	10,811,476
Gen. **A**	73,612,819	71,948,659	32,315,402	32,327,961	2,867,639
Mult.	16,423,218	6,114,896	16,423,218	6,114,896	7,527,058
$\mathbf{S'A}$	105,905,509	80,789,865	50,544,249	40,292,651	9,192,436
Gen. **A**	73,614,739	73,615,219	32,469,159	32,469,079	3,072,351
Mult.	35,955,867	7,578,258	17,240,744	7,041,221	6,115,378
KeyGen	93,301,265	81,299,689	52,012,652	41,681,042	14,042,899
Encaps	111,616,496	86,255,368	56,255,552	45,758,155	14,657,940
Decaps	112,084,038	87,212,153	56,684,122	46,720,217	15,456,163

Table 2: Cycle count of the reference and our optimised implementations, obtained at 24 MHz and with data cache disabled.

5.2 Benchmarks at 24 MHz

For the second part of our results showcase, we downclock the microprosessor to 24 MHz, because we are interested in studying the number of cycles taken by our implementations to perform its operations, thus excluding time spent waiting for memory from the picture. On top of this, we also disable data cache: this forces the Schwabe and Stoffelen [12] implementation of AES128 to run in constant time, and slightly simplifies the flow of execution. Since memory is not a bottleneck at this clock frequency, keeping data in cache is not beneficial anyway.

Table 2 summarises results of all our implementations, in terms of clock cycles measured at 24 MHz and with data cache disabled. Several aspects are worth noticing. When it comes to the official PRNGs, our multiplication routines improve the reference ones by between 59.2% ($\mathbf{S'A}$ with AES128) and 78.9% ($\mathbf{S'A}$ with cSHAKE128). The most prominent fact about our version using xoshiro128**, instead, is the immensely better generation of **A**: within the **AS** multiplication, it takes 91.1% less cycles than AES128 and 96.0% less than cSHAKE128.

A further point of interest, which backs up our initial discussion on benchmarking at different clock frequencies, is in the performance in clock cycles of AES128 at 24 MHz in Table 2 when compared to those listed in Table 1 at 168 MHz. Despite the former numbers are smaller than those of cSHAKE128, FrodoKEM-640 turns out to be slightly slower when using AES128. This is explained by the crucial role that memory access plays in AES128, being based on tables, which is in turns almost neglected by counting cycles at 24 MHz.

We omitted matrix multiplications between small matrices from Table 2, as they do not depend on the PRNG. We improved also in this area: $\mathbf{S'B}$ passed from 369,439 cycles to 111,103 (69.9% better), while $\mathbf{B'S}$ from 410,222 to only 84,461 (79.4% better).

5.3 Comparison with Previous Works

There are two relevant previous works we can compare our results against. The first one is the PQM4 project [5], which we also used as a framework to embed and evaluate our code on the board. Secondly, Howe et al. [3] also recently proposed an implementation of Frodo on the same microcontroller. Table 3 compares the cycle counts of KeyGen, Encaps, Decaps, as well as of some internal functions, across different PRNGs. All numbers, from all sources, have been obtained on the same board running at 24 MHz. Howe et al. [3] also disabled data cache

PRNG	Function	PQM4 [5]	Howe et al. [3]	**This work**
cSHAKE128	KeyGen	$94,119,511$	$85,585,315$	$81,299,689$
	Encaps	$106,992,266$	$112,103,350$	$86,255,368$
	Decaps	$107,505,670$	$112,442,770$	$87,212,153$
	$\mathbf{AS}$		$82,256,529$	$78,068,108$
	$\mathbf{S'A}$		$106,178,196$	$80,789,865$
AES128	KeyGen		$44,603,160$	$41,681,042$
	Encaps		$47,742,966$	$45,758,155$
	Decaps		$48,051,929$	$46,720,217$
	$\mathbf{AS+E}$		$41,308,745$	$38,449,619$
	$\mathbf{S'A+E'}$		$41,833,535$	$40,292,651$
xoshiro128**	KeyGen			$14,042,899$
	Encaps			$14,657,940$
	Decaps			$15,456,163$
	$\mathbf{AS+E}$			$10,811,476$
	$\mathbf{S'A+E'}$			$9,192,436$

Table 3: Cycle count of KeyGen, Encaps, Decaps and other functions as reported by previous works and compared to ours. Numbers were obtained on the same board, running at 24 MHz. Blank spaces refer to data not available.

since the used the same implementation of AES128, thus the comparison holds. The impact of this on the implementation based on cSHAKE128 is negligible, hence also the comparison with PQM4 [5] is meaningful.

PQM4 [5] does not come with a M4-specific implementation of Frodo, hence performance benchmarks were done on the code that we used as a reference in this work. Moreover, only cSHAKE128 is currently implemented and benchmarked.

The attentive reader will notice a small discrepancy between the results of PQM4 in Table 3 and those of our reference implementation in Table 2. This is due to a recent update of the PQM4 framework (commit #23), in which the authors moved to a more recent version of GCC for ARM on Arch Linux, i.e. `arm-none-eabi-gcc` version 8.2.0. Since our setup is based on a different distribution we use `arm-none-eabi-gcc` version *7-2018-q2-update* as provided by ARM at `https://developer.arm.com/open-source/gnu-toolchain/gnu-rm`. For the sake of comparison, before said commit #23 PQM4 reported that KeyGen took $94,191,951$ cycles, Encaps took $111,688,861$ cycles and Decaps took $112,156,317$ cycles. These numbers are indeed much closer to how the reference implementation [10] performs on our board, according to Table 2.

Howe et al. [3] implemented both the AES128 and the cSHAKE128 flavours, including optimised routines in ARM assembly. Several differences with our work exist. First and foremost, they do not optimise the $\mathbf{S'A}$ multiplication when cSHAKE128 is used, which makes our implementation the first of its kind and explains the bigger gap in the first section of Table 3. Note indeed that their Encaps and Decaps cycle count when using cSHAKE128 is extremely close to the reference implementation in Table 2.

Secondly, they change the memory layout of the matrix **S** in the **AS** multiplication, for the sake of optimising loading patterns in that specific case. However, unless amended somewhere else in the code, this makes their implementation incompatible with the reference one, while we chose to make **AS** and **S′A** interchangeable with the reference.

Finally, they did not use SIMD instructions in their ARM assembly code, instead optimised for load/store operations. SIMD instructions do offer a speed-up, but require registers to be filled in a precise way, see Section 2.3, hence the design of the multiplication needs to be tailored around them. For instance, adjacent elements in memory can be loaded in multiple registers using the `ldmia` instruction, but then SIMD instructions can be used only if the two values contained in each register can be multiplied by values in the corresponding halves of other registers, and optionally also accumulated. Since they did not have such a (mild) restriction, they instead optimised for memory access patterns. Unfortunately, it is hard to give precise comparisons in terms of overall performance as their cycle counts lack contextualisation for different frequencies.

6 Conclusions

We showed how SIMD instructions in the ARM Cortex-M4 and a careful analysis of memory layout are valuable tools to implement matrix multiplication. We showcased the performance gain by improving FrodoKEM-640, but we believe that similar techniques can be applied to any scheme based on the standard LWE problem [11] (and variants, as alread demonstrated by Kannwischer et al. [4]). Our **AS** function is 62.7% faster than the reference, and we speeded up **S′A** by 59.2% when AES128 is used, and by 78.9% when using cSHAKE128.

Our watchful analysis of performance at different clock frequencies sheds light on the pitfalls of deriving strong conclusions after benchmarking. Our results apply to the ARM Cortex-M4 mounted on a STM32f4 discovery board: different processors, and even different packagings of the same processor, might change the picture unpredictably.

We can firmly state, however, how unsatisfactory the currently adopted PRNGS are. Generating **A** is by far the most resources hungry operation in FrodoKEM-640, which sounds paradoxical considering that is a public matrix, hence adversaries know all about it: seed, internal state during generation and final result.

Moved by such considerations, we challenged the use of a cryptographically secure PRNG, and instead put forward the idea of using a PRNG achieving pseudorandomness of good statistical quality but not meeting any security goal. Our final suggestion is xoshiro128**: it is extremely fast, it passes all known statistical tests according to the original authors [2] and has an easy-to-integrate structure. We therefore improved generation of **A** of up to 96.0%, suddenly making it one of the least demanding operations in FrodoKEM-640.

Acknowledgements

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme Marie Skłodowska-Curie ITN ECRYPT-NET (Project Reference 643161). Furthermore, Elisabeth Oswald was partially funded by H2020 grant SEAL (Project Reference 725042).

References

1. Erdem Alkim, Philipp Jakubeit, and Peter Schwabe. NewHope on ARM Cortex-M. In *Security, Privacy, and Applied Cryptography Engineering*, pages 332–349. Springer International Publishing, 2016.
2. David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. *CoRR*, abs/1805.01407, 2018.
3. James Howe, Tobias Oder, Markus Krausz, and Tim Güneysu. Standard lattice-based key encapsulation on embedded devices. *IACR TCHES*, 2018(3):372–393, 2018. `https://tches.iacr.org/index.php/TCHES/article/view/7279`.
4. Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. Cryptology ePrint Archive, Report 2018/1018, 2018. `https://eprint.iacr.org/2018/1018`.
5. Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. `https://github.com/mupq/pqm4`.
6. Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, Jun 2015.
7. Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4):22:1–22:40, August 2007.
8. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.
9. George Marsaglia. Xorshift RNGs. *Journal of Statistical Software, Articles*, 8(14):1–6, 2003.
10. Michael Naehrig, Erdem Alkim, Joppe Bos, Leo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, Ananth Raghunathan, and Douglas Stebila. Frodokem. Technical report, National Institute of Standards and Technology, 2017. available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions`.
11. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.
12. Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 180–194. Springer, Heidelberg, August 2016.