Dilithium for Memory Constrained Devices

Joppe W. Bos, Joost Renes, and Amber Sprenkels

NXP Semiconductors
{joppe.bos,joost.renes}@nxp.com
amber@electricdusk.com

Abstract. We investigate the use of the Dilithium post-quantum digital signature scheme on memory-constrained systems. Reference and optimized implementations of Dilithium in the benchmarking framework pqm4 (Cortex-M4) require $50-100$ KiB of memory, demonstrating the significant challenge to use Dilithium on small IoT platforms. We show that compressing polynomials, using an alternative number theoretic transform, and falling back to the schoolbook method for certain multiplications reduces the memory footprint significantly. This results in the first implementation of Dilithium for which the recommended parameter set requires less than 7 KiB of memory for key and signature generation and less than 3 KiB of memory for signature verification. We also provide benchmark details of a portable implementation in order to estimate the performance impact when using these memory reduction methods.

1 Introduction

Digital signatures are one of the essential building blocks in our cybersecurity infrastructure. These cryptographic algorithms need to be computed on different platforms in the ecosystem: ranging from high-end cloud servers to resource constrained embedded devices. Especially the rise of the multitude of Internet-of-Things (IoT) devices, which has steadily outgrown the number of humans living on this planet and is expected to keep increasing [20], highlights the importance of being able to compute security primitives on such constrained devices.

Currently, the most commonly used digital signature schemes are RSA [26] and variants of (EC)DSA [13,21]. However, with the possibility of a quantum computer being realized, the security of RSA and (EC)DSA is threatened. Cryptography designed to run on our current platforms and which is secure against such a quantum threat is called *post-quantum* or *quantum safe* cryptography (PQC). In an effort to standardize such algorithms the US National Institute of Standards and Technology (NIST) put out a call for proposals [22] to submit candidate algorithms in 2016. As of July 2020, seven out of fifteen remaining candidates have been marked as finalists, of which a subset is expected to be selected for standardized before April 2022. One of the three finalists for digital signatures is CRYSTALS–Dilithium [5,18], which will be the focus of this paper.

One can observe that Dilithium signing has two main practical drawbacks for embedded devices. The first one is the variable signing time, which follows a geometric distribution. When using the parameter set targeting NIST security level 3, the probability that the signing time is more than twice the expected average is approximately 14 percent. This is significant and will have a real impact on many performance requirements for various use-cases. The second drawback relates to the memory requirements which are significantly higher for virtually all PQC schemes compared to the classical digital signature counterparts. This is not only attributed to relatively large key and signature sizes, but also heavy use of stack space for the storage of intermediate data. For example, the embedded benchmarking platform *pqm4* [10,9] (which executes on the ARM Cortex-M4) initially reported memory requirements for Dilithium in the range of $50-100$ KiB for both the original reference as well as the optimized implementations.

Dilithium has received a significant amount of attention from the cryptographic community. One direction of study comes from an applied cryptographic engineering perspective: how can one realize efficient implementations in practice for a selected target platform. Often the single most important optimization criteria is latency: the algorithm needs to execute as fast as possible, at the

possible expense of other important metrics. Examples include the AVX2-based implementations from [5] and [6]; or the implementation from [24], which requires up to 175 KiB of memory.

Instead, we target platforms which have significantly less memory and computational power. Typical examples are platforms which are based on ARM Cortex-M0(+) cores. Such platforms are typical for a large family of IoT applications. Products in this range include the LPC800 series by NXP (4 − 16 KiB of SRAM), STM32F0 by ST (4 − 32 KiB of SRAM), or the XMC1000 by Infineon (16 KiB of SRAM). It is clear that PQC algorithms with memory requirements of well over 50 KiB do not fit on these platforms and will not be able to sign nor verify digital post-quantum signatures.

In this paper we investigate if it is possible to execute Dilithium on such memory constrained devices that often have up to 8 KiB of SRAM and, if so, which performance penalty is incurred. Recently, there have been promising results in this direction [7,1]. Most notably the third strategy from [7] manages to run the recommended parameter set of Dilithium in just below 10 KiB: this is a remarkable achievement but still too large for many devices with only 8 KiB of SRAM, especially when you take into account that other applications will require memory as well.

2 Preliminaries

In this paper, we follow the same notation as the Dilithium specification [18]. We let R and R_q respectively denote the rings $\mathbb{Z}[X]/(X^n + 1)$ and $\mathbb{Z}_q[X]/(X^n + 1)$, for q an integer. Throughout this document, the value of n will always be 256 and q will be the prime $8380417 = 2^{23} - 2^{13} + 1$. Scalars and polynomials are written in a regular font (a), vectors are written in lowercase bold ($\mathbf{a}$), and matrices are written in uppercase bold ($\mathbf{A}$). All values in the NTT domain are written with a hat ($\hat{a}$), $\circ$ denotes coefficient-wise multiplication between two polynomials, and $\|a\|_\infty$ denotes the infinity norm of a.

In the remaining part of this work we describe optimization strategies for the Dilithium signature algorithm. One of the techniques used is to reuse memory space that is used by another value at some point during the computation to reduce the overall memory requirement. In the context programming languages, a variable's *lifetime* is the time from which it is *initialized*, until the last time it is *used*. After a variable has been used for the last time, it is *dead*. This is not the same as *allocation*, which means that a certain part of memory has been reserved for the storage of a variable. When the lifetimes of two variables overlap this means that, at some point in the algorithm, both variables are alive at the same time. As a consequence, such overlapping lifetimes mean that both variables cannot share the same memory location.

2.1 Dilithium

One approach to construct digital signatures comes from the realm of lattice-based cryptography. A particular problem used as a security foundation is known as the Learning With Errors (LWE) problem [25] introduced by Regev, which relates to solving a "noisy" linear system modulo a known integer. This problem can be used as the basis for a signature scheme, as shown by Lyubashevsky [17], by improving on their idea to apply Fiat-Shamir with aborts [16,12] to lattices.

The Ring Learning With Errors (R-LWE) [19,23] is a variant of this approach which works in the ring of integers of a cyclotomic number field and offers significant storage and efficiency improvements compared to LWE. Yet another approach to address certain shortcomings in both LWE and R-LWE and allows one to interpolate between the two. This Module Learning with Errors problem (M-LWE) [4,15] takes the R-LWE problem and, informally, replaces the single ring element with module elements over the same ring. Using this intuition, R-LWE can be seen as M-LWE with module rank 1. A practical instantiation of M-LWE with various (practical) improvements resulted in the CRYSTALS–Dilithium [5,18] digital signature algorithm.

The key generation is listed in Algorithm 1, the signature generation in Algorithm 2, and the signature verification in Algorithm 3.

Algorithm 1 Dilithium key generation (taken from [18])

Output: A public/secret key pair (pk, sk).
1: $\zeta \leftarrow \{0,1\}^{256}$
2: $(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} := \mathsf{H}(\zeta)$
3: $\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated in NTT domain as $\hat{\mathbf{A}}$
4: $(\mathbf{s_1}, \mathbf{s_2}) \in S_\eta^\ell \times S_\eta^k := \mathsf{ExpandS}(\rho')$
5: $\mathbf{t} := \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$ ▷ Compute $\mathbf{A}\mathbf{s_1}$ as $\mathsf{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathsf{NTT}(\mathbf{s_1}))$
6: $(\mathbf{t_1}, \mathbf{t_0}) := \mathsf{Power2Round}_q(\mathbf{t}, d)$
7: $tr \in \{0,1\}^{256} := \mathsf{H}(\rho \parallel \mathbf{t_1})$
8: **return** $(pk = (\rho, \mathbf{t_1}), sk = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0}))$

Algorithm 2 Dilithium signature generation (taken from [18])

Input: Secret key sk and a message M.
Output: Signature $\sigma = \mathsf{Sign}(sk, M)$.
1: $\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated in NTT domain as $\hat{\mathbf{A}}$
2: $\mu \in \{0,1\}^{512} := \mathsf{H}(tr \parallel M)$
3: $\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$
4: $\rho' \in \{0,1\}^{512} := \mathsf{H}(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing)
5: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do** ▷ Pre-compute $\hat{\mathbf{s}}_\mathbf{1} := \mathsf{NTT}(\mathbf{s_1})$, $\hat{\mathbf{s}}_\mathbf{2} := \mathsf{NTT}(\mathbf{s_2})$, and $\hat{\mathbf{t}}_\mathbf{0} := \mathsf{NTT}(\mathbf{t_0})$
6: $\quad \mathbf{y} \in S_{\gamma_1}^\ell := \mathsf{ExpandMask}(\rho', \kappa)$
7: $\quad \mathbf{w} := \mathbf{A}\mathbf{y}$ ▷ $\mathbf{w} := \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \cdot \mathsf{NTT}(\mathbf{y}))$
8: $\quad \mathbf{w_1} := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$
9: $\quad \tilde{c} \in \{0,1\}^{256} := \mathsf{H}(\mu \parallel \mathbf{w_1})$
10: $\quad c \in B_\tau := \mathsf{SampleInBall}(\tilde{c})$ ▷ Store c in NTT representation as $\hat{c} = \mathsf{NTT}(c)$
11: $\quad \mathbf{z} := \mathbf{y} + c\mathbf{s_1}$ ▷ Compute $c\mathbf{s_1}$ as $\mathsf{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_\mathbf{1})$
12: $\quad \mathbf{r_0} := \mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s_2}, 2\gamma_2)$ ▷ Compute $c\mathbf{s_2}$ as $\mathsf{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{s}}_2)$
13: $\quad$ **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta$ **then**
14: $\quad\quad (\mathbf{z}, \mathbf{h}) := \perp$
15: $\quad$ **else**
16: $\quad\quad \mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t_0}, \mathbf{w} - c\mathbf{s_2} + c\mathbf{t_0}, 2\gamma_2)$ ▷ Compute $c\mathbf{t_0}$ as $\mathsf{NTT}^{-1}(\hat{c} \cdot \hat{\mathbf{t}}_\mathbf{0})$
17: $\quad\quad$ **if** $\|c\mathbf{t_0}\|_\infty \geq \gamma_2$ **or** the # of 1's in $\mathbf{h}$ is greater than ω **then**
18: $\quad\quad\quad (\mathbf{z}, \mathbf{h}) := \perp$
19: $\quad \kappa := \kappa + \ell$
20: **return** $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

Algorithm 3 Dilithium signature verification (taken from [18])

Input: Public key pk, message M and signature σ.
Output: Signature verification $\mathsf{Verify}(pk, M, \sigma)$: true if σ is a valid signature of M using pk or false otherwise.
1: $\mathbf{A} \in R_q^{k\times\ell} := \mathsf{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated in NTT Representation as $\hat{\mathbf{A}}$
2: $\mu \in \{0,1\}^{512} := \mathsf{H}(\mathsf{H}(\rho \parallel \mathbf{t_1}) \parallel M)$
3: $\mathsf{SampleInBall}(\tilde{c})$
4: $\mathbf{w}_1' := \mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c\mathbf{t_1} \cdot 2^d, 2\gamma_2)$
5: **return** $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ **and** $[\![c = \mathsf{H}(\mu \parallel \mathbf{w}_1')]\!]$ **and** $[\![\#$ of 1's in $\mathbf{h}$ is $\leq \omega]\!]$

Table 1. Overview of the relevant Dilithium parameters (taken from [18]).

NIST security level	2	3	5
q [modulus]	8380417	8380417	8380417
(k, ℓ) [dimensions of $\mathbf{A}$]	$(4, 4)$	$(6, 5)$	$(8, 7)$
η [secret key range]	2	4	2
τ [hamming weight of c]	39	49	60
d [dropped bits from $\mathbf{t}$]	13	13	13
γ_1 [$\mathbf{y}$ coefficient range]	2^{17}	2^{19}	2^{19}
γ_2 [low-order rounding range]	$(q-1)/88$	$(q-1)/32$	$(q-1)/32$
β [range of $c\mathbf{s_1}$ and $c\mathbf{s_2}$]	78	196	120
public key size (bytes)	1312	1952	2592
secret key size (bytes)	2528	4000	4864
signature size (bytes)	2420	3293	4595

The main mathematical building blocks in Dilithium are polynomials in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$. These polynomials consist of $n = 256$ elements modulo $q < 2^{23}$. So far, all previous Dilithium implementations have stored polynomials in a buffer of 256 (`u`)`int32_t`s. This counts up to 1024 bytes $\hat{=}$ 1 KiB per polynomial. This explains why Dilithium implementations generally require a significant amount of memory: all of KeyGen, Sign, and Verify use $k \cdot \ell$ polynomials to store $\mathbf{A}$; add to that four vectors of length k, and you have already reached a total memory use of $\{32, 54, 88\}$ KiB.

It should be noted that not all polynomials require this much memory. The situation is different for $(\mathbf{s_1}, \mathbf{s_2}) \in S_\eta^\ell \times S_\eta^k$ which are uniformly random numbers in the range $\{-\eta, \ldots, +\eta\}$. Since $\eta = \{2, 4, 2\}$ for Dilithium-{2,3,5} this means each coefficient can be stored using $\{3, 4, 3\}$ bits respectively. Hence, the polynomial $\mathbf{s}_{\mathbf{x},i}$ requires only $\{96, 128, 96\}$ bytes of memory. However, when this polynomial is stored naively in the NTT domain then one cannot use this property and the full 1024 bytes are needed: requiring $\{10.7, 8.0, 10.7\}$ times more memory.

3 Dilithium Signature Generation

The digital signature generation in Dilithium requires a significant amount of memory. To illustrate, the fastest implementation reported on the benchmark results from pqm4[1] requires ≈ 49, ≈ 80 and ≈ 116 KiB for Dilithium-{2,3,5}. In this section we outline the proposed techniques to reduce the memory requirements.

3.1 Streaming A and y

In Dilithium's signature generation algorithm the matrix $\mathbf{A}$ requires $k \cdot \ell$ KiB: by far the largest contributor to memory. A straight-forward optimization is to not generate the entire matrix $\mathbf{A}$ but only generate the elements of $\mathbf{A}$ and $\mathbf{y}$ on-the-fly when they are needed. This approach was proposed and discussed already in [7, Strategy 3].

The expected memory reduction of this optimization is $k \cdot \ell$ KiB for $\mathbf{A}$, and ℓ KiB for $\mathbf{y}$; in practice this means a saving of $\{20, 35, 63\}$ KiB, for Dilithium-{2,3,5} respectively. This optimization comes at a performance price: the matrix $\mathbf{A}$ needs to be regenerated again from ρ on every iteration of the rejection-sampling loop. Moreover, $\mathbf{y}$ needs to be generated twice during each iteration of the rejection-sampling loop; once for computing $\mathbf{w} = \mathbf{A}\mathbf{y}$, and once for computing

[1] Accessed February 14, 2022 using revision `3bfbbfd3`.

$\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ later on. In [7], the authors report a slowdown factor of around 3.3 − 3.9 compared to precomputing $\mathbf{A}$ and $\mathbf{y}$ completely.[2]

3.2 Compressing w

Another significant contributor to the overall memory requirements is the vector $\mathbf{w}$. This could be resolved if one could compute and use a single element at a time during the signature generation. Unfortunately, this is not possible due to the overlapping lifetimes of $\mathbf{w}$ and c, as identified in [7]. In line 9 of Algorithm 2, c is computed from $\mathbf{w_1} = \mathsf{HighBits}(\mathbf{w})$. On lines 12 and 16, the values $\mathbf{r_0}$ and $\mathbf{h}$ depend on c, and the complete vector $\mathbf{w}$. This means that either *all* elements of $\mathbf{w}$ must be retained between computing c and computing $\mathbf{r_0}$ and $\mathbf{h}$, or $\mathbf{w} = \mathbf{Ay}$ must be computed twice during each iteration of the rejection-sampling loop. Recomputing the matrix-vector multiplication in every loop iteration will roughly double the execution time of the signing algorithm: although a viable direction to reduce memory we deemed this performance impact too large. Instead, we explore the other option where all elements of $\mathbf{w}$ have to be alive at the same time at the cost of storing k polynomials.

One polynomial has $n = 256$ coefficients, which are all bounded by $q = 2^{23} - 2^{13} + 1$. In previous works, implementations have always used 32-bit data-types for storing these coefficients. Instead, we use a *compressed* representation for storing $\mathbf{w}$. Instead of using 32-bit registers for storing $\mathbf{w}$ coefficients, the approach is to explicitly reduce them modulo q, reducing each coefficient to 24 bits and next pack the 256 24-bit coefficients into a 768-byte array. The (24-bit) compressed coefficients reduce the amount of storage that is used for storing $\mathbf{w}$ from $k \cdot 1024$ bytes to $k \cdot 768$ bytes, which results in a reduction of {1.0,1.5,2.0} KiB for Dilithium-{2,3,5}, respectively. Packing and unpacking coefficients of $\mathbf{w}$ adds a little overhead during the matrix-vector multiplication.

It should be noted that one could compress each coefficient into 23 bits instead of 24 bits. This would save an additional 32 bytes per polynomial. However, working with the 23-bit format (packing and unpacking) is significantly more cumbersome and therefore slower compared to the 24-bit format for alignment reasons and the need for more expensive reductions during the computation of $\mathbf{w}$. This explains why we compress for the results presented in Section 5 to 24 bits.

3.3 Compressing $c \cdot \mathbf{s_1}$, $c \cdot \mathbf{s_2}$, and $c \cdot \mathbf{t_0}$

The multiplication of the challenge $c \in B_\tau$ with the polynomials $\mathbf{s_1} \in S_\eta^\ell$, $\mathbf{s_2} \in S_\eta^k$, and $\mathbf{t_0} \in S_{2^d}^k$, is typically computed using NTTs (see line 11, line 12 and line 16 of Algorithm 2). As the values of $\mathbf{s_1}$, $\mathbf{s_2}$ and $\mathbf{t_0}$ are static throughout a whole signing computation, it is computationally most efficient to pre-compute the NTTs on all these elements, and store $\hat{\mathbf{s}}_\mathbf{1}$, $\hat{\mathbf{s}}_\mathbf{2}$ and $\hat{\mathbf{t}}_\mathbf{0}$ in memory before entering the rejection-sampling loop. Avoiding the storage of these elements reduces the total memory used by $2k + \ell$ KiB; i.e., {12,17,23} KiB for Dilithium-{2,3,5}, respectively. Indeed, this would naively imply a performance loss as the NTTs need to be computed several times (at least once for each aborted signature). However, the routine using (inverse) NTTs on-the-fly needs at least 1.75 KiB of space: 1 KiB is needed to compute the (inverse) NTT for one operand, while 0.75 KiB is needed to store the other operand in (24-bit) compressed form. We find that, for the computation of $\mathbf{s_1}$, $\mathbf{s_2}$ and $\mathbf{t_0}$ we do not necessarily need to use the regular NTT. For most values involved, there is a lot of structure that can be exploited. In the remainder of this section we discuss three different ideas to compute $c \cdot \mathbf{s_1}, c \cdot \mathbf{s_2}$ and $c \cdot \mathbf{t_0}$ with lower memory requirements than using regular NTTs.

Schoolbook multiplication. The most obvious choice for polynomial multiplication is the schoolbook approach. At first glance, using schoolbook multiplication actually requires *more* memory compared to NTT-based multiplication because one cannot do the multiplication in-place. However, when using schoolbook multiplication, one does not need to store the right-hand-side operand polynomials ($\mathbf{s_1}$, $\mathbf{s_2}$, and $\mathbf{t_0}$) completely. We can multiply their coefficients in a streaming fashion, unpacking them “lazily” from the secret key. Apart from using a small buffer we have now removed

[2] These numbers are for NIST round-2 Dilithium and do not directly apply to the round-3 version.

the need to store any full element from $\mathbf{s_1}$, $\mathbf{s_2}$ and $\mathbf{t_0}$. Although one still needs 1.0 KiB for the accumulator polynomial, only 68 bytes are required for storing the challenge c; and a small buffer of 32 bytes, which is used to unpack polynomial coefficients more efficiently from the secret key. This adds up to 1124 bytes in total: a reduction of a factor 1.37 compared to using an NTT.

Furthermore, one can reduce the computational as well as the memory complexity by exploiting the regular structure of the challenge c. Recall that the challenge polynomial has exactly τ non-zero coefficients that are either ± 1, where $\tau \in \{39, 49, 60\}$ depending on the Dilithium parameter set. Therefore, when multiplying c with some other polynomial, one really only needs to multiply each coefficient from the right-hand side operand with τ coefficients in c. Skipping the multiplications with the zero-elements is not a security concern (e.g., from a timing leakage perspective) since the challenge value c is public.

Using this property, one can use a data structure for c that allows for fast iteration over all the non-zero coefficients. We use a single 64-bit datatype which indicates for each of the τ non-zero positions whether it is a $+1$ or a -1; and an array of τ bytes which stores positions of the non-zero coefficients. The benefit of storing the indices of all non-zero coefficients, as opposed to storing a bit-string with bits set for each non-zero coefficient, is the fast iteration over the non-zero indices. If we store a bit for every coefficient in c, we have to do a conditional addition/subtraction of the coefficient in the other operand for *every* coefficient of c, i.e., n times. By only storing the non-zero indices, we only have to do the addition/subtraction τ times and avoid computing any multiplications. Hence, this polynomial multiplication with c can be done using $\tau \cdot n$ additions or subtractions only.

Alternative Number Theoretic Transforms. When computing $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ one can use a different-sized NTT over a smaller prime as described in [1]. The idea is that all coefficients of both $c\mathbf{s_1}$ and $c\mathbf{s_2}$ are bounded by $\pm\tau \cdot \eta = \pm\beta$. This allows computing the polynomial product with modulus $q' = 257$ for Dilithium-{2,5}, and $q' = 769$ for Dilithium-3. Since the coefficients in the product are bounded by $\pm\beta$, they will not overflow when computing them modulo $q' \geq 2\beta$. In [1], this improves the performance of the NTT-based multiplications because—with $q' = 257$—some of the multiplications with twiddle factors become cheaper. Moreover, [1] still uses 32-bit registers for all values, which provides so much headroom that it eliminates the need for any intermediate Barrett reductions in both NTT algorithms. However, the small-modulus NTTs also allows one to store all coefficients in 16-bit variables: computing an NTT in half the amount of memory at the cost of reintroducing the intermediate Barrett reductions. When using this technique, the memory requirement of $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ is reduced to 1.0 KiB: 0.5 KiB for the first operand and product, and another 0.5 KiB for the second operand.

Kronecker Substitution. By applying (generalizations of) Kronecker substitution [14,8] to $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ one can reduce the polynomial multiplication to the integer multiplications $c(2^\lambda) \cdot \mathbf{s_1}(2^\lambda)$ and $c(2^\lambda) \cdot \mathbf{s_2}(2^\lambda)$ modulo $2^{256\lambda} + 1$. The application of Kronecker substitution to lattice-based cryptography has been studied [2,3], but its use for $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ has not been considered yet. In order to retrieve the coefficients of the resulting polynomial, we require that $2^\lambda \geq 2\beta$. This means we can select $\lambda = 8$ for Dilithium-{2,5} and $\lambda = 9$ for Dilithium-3, reducing to 2048-bit and 2304-bit multiplications respectively. This requires 256 or 288 bytes for each of the two inputs and result polynomials: assuming the result can overwrite one of the inputs this means 512 or 576 bytes in total. Additionally one can use the more general Kronecker+ method [3] to improve the performance further (the optimal setting depending on the platform).

Although Kronecker substitution works perfectly well on the regular central processing unit it is particularly suitable for small systems that typically have dedicated hardware to perform (public-key) cryptographic operations in a timely manner. For RSA or elliptic-curve cryptography (ECC), such co-processors come in the form of large-integer multipliers that are heavily optimized for performing integer (modular) multiplications.

3.4 Variable Allocation

After applying the memory optimizations described above we analyzed efficient memory allocation schemes during the Dilithium signature generation algorithm. This showed that one can reuse the 1 KiB memory location that is used every time to compute with a non-compressed polynomial. On top of that, we need 128 bytes for storing μ and ρ'; and 68 bytes for storing c, as described earlier ($\tilde{c}$ is stored solely in the output buffer). The complete memory allocation of the signature generation algorithm is listed in Figure 1.

When looking at Figure 1 one can observe that the memory bottle-neck is shared between multiple subroutines. We see no trivial way to further optimize the allocation of variables in memory. The only time-memory tradeoff that could still be performed is to keep a single element of $\mathbf{w}$ at a time. Following the recommendations from [7] we dismiss this approach because it requires to compute all elements of $\mathbf{w}$ *twice* during each iteration of the rejection sampling loop. This would not only require expanding all elements of $\mathbf{A}$ *and* $\mathbf{y}$ twice, one would also need to recompute $\hat{\mathbf{y}} = \mathsf{NTT}(\mathbf{y})$ and $\mathbf{w} = \mathsf{NTT}^{-1}(\hat{\mathbf{w}})$. Because the matrix-multiplication is already a dominating factor in the signing algorithm, this optimization would likely result in another slowdown by a factor two. Its gains in terms of memory consumption would be $(k-1) \cdot 768$ bytes, i.e., $\{2.25, 3.75, 5.25\}$ KiB for Dilithium-{2,3,5}, so it might be worthwhile if one can compensate for or cope with this performance penalty.

4 Dilithium Key Generation and Signature Verification

Both the Dilithium key generation and verification algorithms are fundamentally different from the signature algorithm with the most important difference that there is no rejection-sampling loop. Therefore, there is no performance benefit to precomputing the matrix $\mathbf{A}$ in these algorithms, which already reduces the memory requirement naturally. Moreover, in both KeyGen and Verify there are no polynomials for which it makes sense to precompute the NTT representation to speed-up the algorithms. This makes both algorithms significantly more lightweight in terms of memory compared to the signature generation, even without any further optimizations.

It is common that the key generation algorithm is executed on the same device where one performs the signature generation algorithm. Therefore, we do not attempt to reduce the memory footprint of KeyGen to the maximum extent, but instead try and minimize the memory footprint of $\max(\mathsf{KeyGen}, \mathsf{Sign})$. In other words, we optimize the memory use of KeyGen, until it is at least as low as the memory use of Sign which we try to optimize as much as possible.

4.1 Key Generation

When following the same strategy for computing the multiplication $\mathbf{A} \cdot \mathbf{s_1}$ in the key generation algorithm as in the signing algorithm one can already remove the need for ℓ different memory slots for polynomials. Using this optimization in combination with careful scheduling the other memory (see Fig. 2) already means that all variables used in KeyGen use less memory than the signature generation algorithm. Hence, there is no reason to sacrifice any performance to optimize the KeyGen algorithm further.

Let us outline some memory improvements for the interested reader who has requirements to reduce the memory even further. One path comes from the observation that one can transpose the order in which the multiplication in $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$ is performed. Recall that in the Sign algorithm, the lifetime of c overlaps with the lifetimes of all elements in $\mathbf{w}$ (where $\mathbf{w}$ is the output of the matrix-vector multiplication) which limits the potential to reduce memory. However, in the KeyGen algorithm there is no (equivalent to) c, i.e., there is no variable that causes the lifetimes of the elements in $\mathbf{t}$ to overlap. Hence, the elements in $\mathbf{t}$ do not have to be alive at the same time and can be computed in a streaming fashion. With this optimization one can reduce the memory by $(k-1)$ KiB, saving $\{3.0, 4.5, 6.0\}$ for Dilithium-{2,3,5}, respectively.

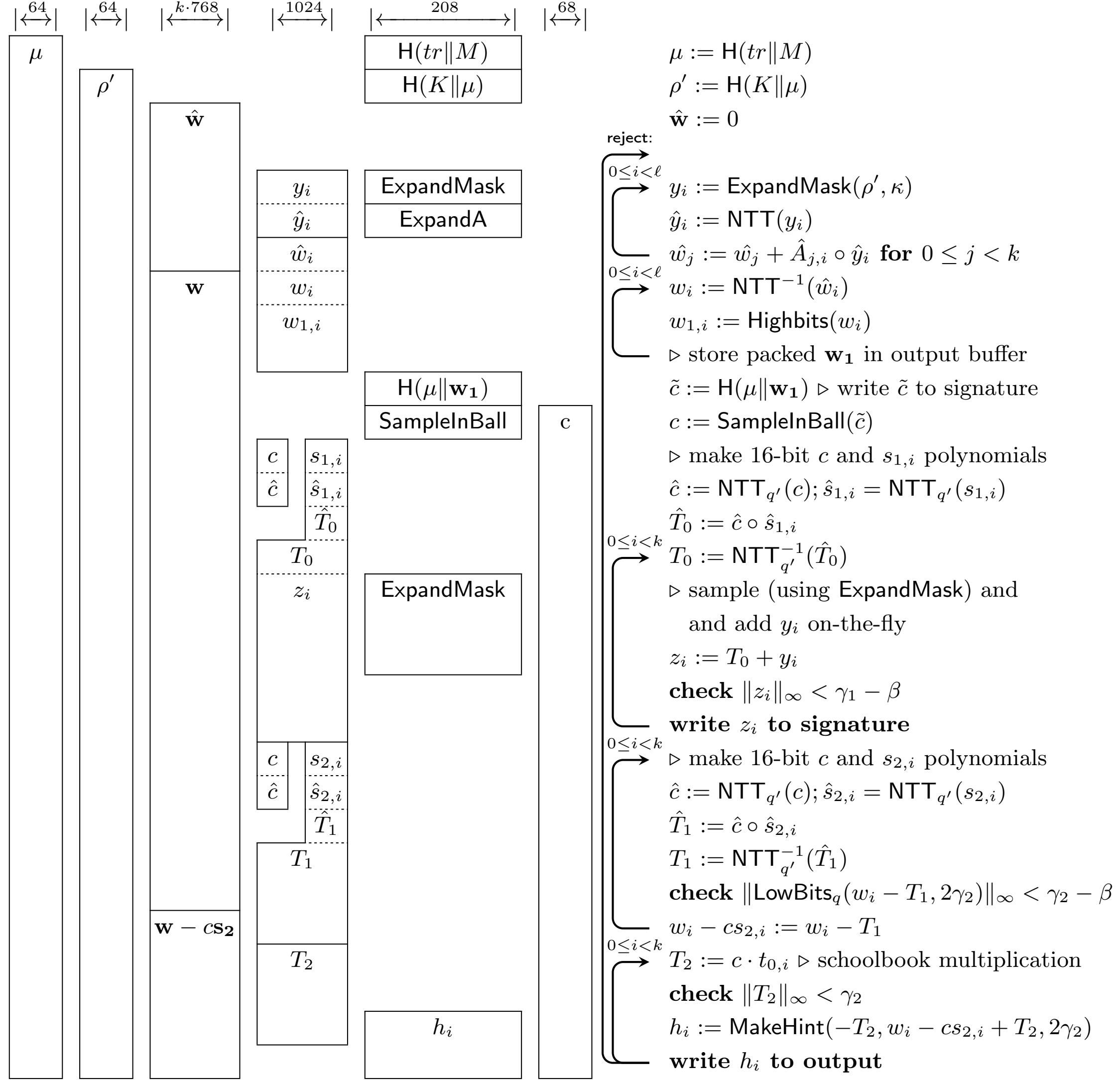

Fig. 1. Memory allocation of the Dilithium signature generation algorithm. Horizontal direction shows the memory slots that are used. Vertical direction shows the progression in time. The boxes indicate the lifetimes of the variables used in the algorithm. Dotted barriers denote that a variable is *renamed*, i.e., it is modified in-place. Arrows in the algorithm indicate loops that iterate over some range, except for the loop annotated by reject:, which indicates which code is repeated when a signature in the Sign algorithm is aborted. All temporary values are denoted by a T.

4.2 Signature Verification

In the setting of the Dilithium signature verification algorithm we are interested in minimizing the memory usage as much as possible. There are many embedded applications that only use signature verification: e.g., secure boot implementations or in the case of public-key infrastructures.

The optimizations one can apply to the signature verification algorithm follow the same pattern as those of Sign and KeyGen. In particular, it is possible to verify any signature using only two slots for storing polynomials, of which one is 1.0 KiB and one is 768 bytes, using the optimizations from Section 4.1. Apart from the 1.75 KiB for storing two polynomials, one still need twice the space for storing the SHA-3 state (208 bytes) plus one compressed challenge polynomial c (68 bytes). This sums up to a minimum of 2276 bytes of required memory for such an approach in

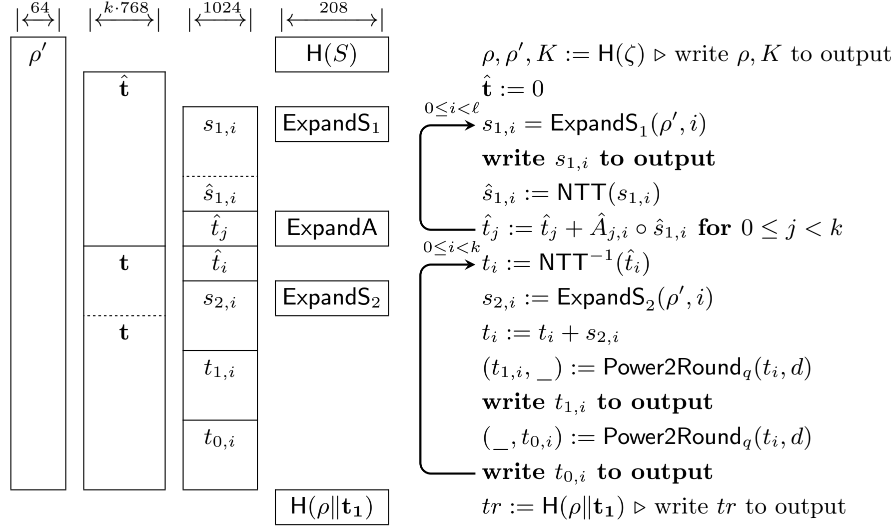

Fig. 2. Memory allocation of the Dilithium key generation algorithm. Horizontal direction shows the memory slots that are used. Vertical direction shows the progression in time. The boxes indicate the lifetimes of the variables used in the algorithm. Dotted barriers denote that a variable is *renamed*, i.e., it is modified in-place. Arrows in the algorithm indicate loops that iterate over some range.

the Dilithium verification algorithm. In contrast to the KeyGen and Sign algorithms, the memory usage of the Verify algorithm is independent from any of the Dilithium parameters.

5 Results & Discussion

Our implementation. Using the Dilithium reference implementation[3] as a starting point, we wrote a new implementation for Dilithium, in which we applied the techniques described in Sections 3 and 4. Because we are only interested in validating the memory reduction techniques and *not* focused on performance we have opted to write a cross-platform implementation in pure C. Correspondingly, our implementation does not include any architecture-specific optimizations.

Our implementation introduces many new internal data types that are optimized for a lower memory footprint: like compressed polynomials (with 24-bit coefficients and 16-bit coefficients) and the compressed challenge. We implemented the $q' \in \{257, 769\}$ NTTs for $c \cdot \mathbf{s_1}$ and $c \cdot \mathbf{s_2}$ multiplications, and we implemented the schoolbook multiplication for the $c \cdot \mathbf{t_0}$ and $c \cdot \mathbf{t_1}$ multiplications. We improved the implementation such that parts can be called in a streaming fashion. For example, the matrix-vector multiplication and ExpandA routines have been merged into a single non-buffering function; and almost all packing/unpacking functions have been refactored to allow for (un)packing polynomials in small chunks. Because of the tight memory budget we have removed some local stack allocations from all internal Dilithium routines. Instead, one memory block is allocated on the stack in the root functions (i.e., `dilithium_keygen`, `dilithium_signature`, and `dilithium_verify`) and passed to the internal functions.

As opposed to the previous works that only support a single Dilithium variant at a time, selected using C preprocessor macros at compile time, our implementation integrates all variants at the same time, and the variant is selected by the user at runtime as in typical in cryptographic software libraries.

Results. We integrated our implementation into a local fork of the benchmarking framework pqm4 [9].[4] We compared the memory footprint and the execution times of our implementation to

[3] `https://github.com/pq-crystals/dilithium`
[4] Commit hash `e47864b3`, forked on 8 Oct 2021.

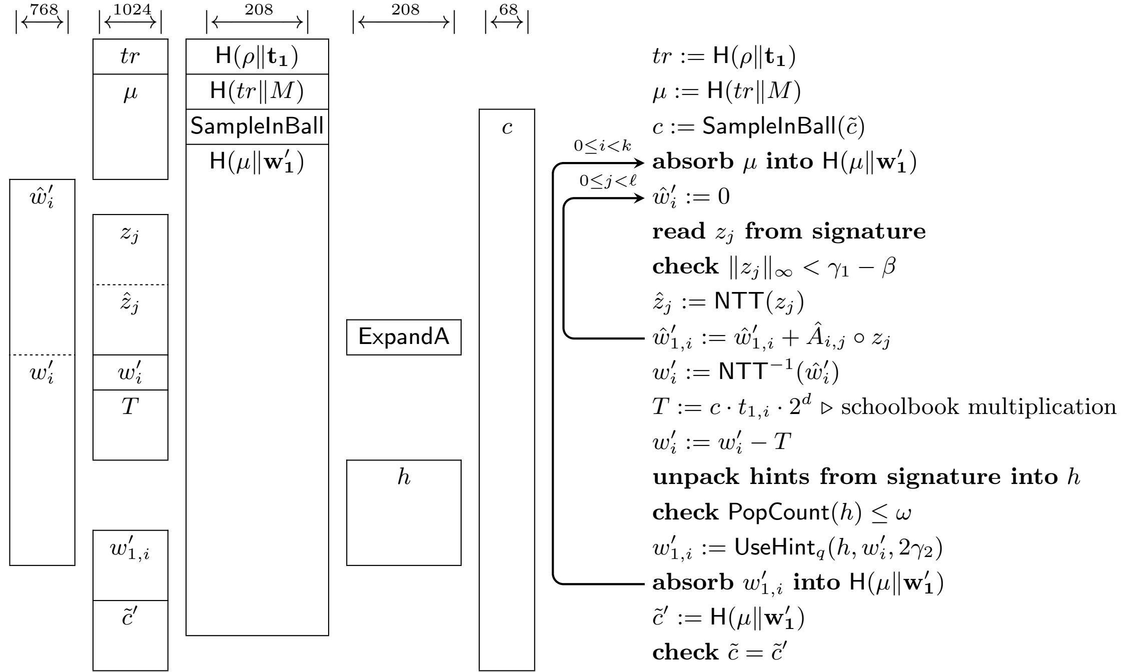

Fig. 3. Memory allocation of the Dilithium signature verification algorithm. Horizontal direction shows the memory slots that are used. Vertical direction shows the progression in time. The boxes indicate the lifetimes of the variables used in the algorithm. Dotted barriers denote that a variable is *renamed*, i.e., it is modified in-place. Arrows in the algorithm indicate loops that iterate over some range. All temporary values are denoted by a T.

those of the Dilithium implementation in PQClean [11], the Dilithium-round-3 updated port of [7] in pqm4, and the recent implementation results from [1].[5]

It should be noted that all of these implementations have different goals and implementation methods, so evaluating the benchmarking results is not as straightforward as just comparing performance numbers. Firstly, the PQClean implementation has been published as a "clean" implementation of Dilithium. Its main goal is to provide an implementation of Dilithium, written purely in C, that works cross-platform and follows best coding practices. It has been written with performance in mind and ensures a running time independent of secret-key related material. However, it does not include any platform-optimized assembly code which has the potential to greatly improve the performance. On the other side, there are the pqm4 ([7]) and [1] implementations. These implementations are specifically hand-crafted for the ARM Cortex-M4 platform and are highly optimized for performance (i.e., reducing the number of required cycles) and large parts of these implementations are written in Armv7 assembly. Some attention is paid to reducing memory in "Strategy 3" from [7]; unfortunately it is hard to compare directly since the paper presents numbers for the round-2 parameters of Dilithium which are significantly different compared to the latest (Round 3) ones. As an indication, the round-2 Dilithium-3 memory usage of signature verification and generation using this strategy are in both settings 10 KiB: significantly less compared to previous work but still too large for the embedded devices we target in this paper.

Our implementation is designed with a different goal in mind: it is a cross-platform C implementation that optimizes in the first place for memory usage to ensure it can execute on memory constrained ($\leq$ 8KiB) platforms. It makes a significant amount of sacrifices in terms of performance and does not contain any routines that are specially optimized for the Cortex-M4 (the techniques presented in this paper are platform independent). Therefore we expect the pqm4 im-

[5] As of early 2022, this implementation has replaced the port of [7] in pqm4.

Table 2. Memory usage and cycle counts for Dilithium in kibibytes (KiB) and kilocycles (kcc). K, S, and V correspond to the signing primitives KeyGen, Sign, and Verify respectively. All cycle counts were averaged over 10 000 iterations.

	variant		Dilithium-2			Dilithium-3			Dilithium-5		
			KiB	total ≤ 8	kcc	KiB	total ≤ 8	kcc	KiB	total ≤ 8	kcc
with asm	[7] (pqm4)	K	37.1		1 602	59.6		2 835	95.7		4 835
		S	47.9		4 219	72.3		6 742	112.3		8 960
		V	35.2		1 579	56.6		2 700	90.8		4 718
	[1]	K	37.1		1 598	59.6		2 830	95.7		4 828
		S	47.9		4 083	67.4		6 624	113.3		8 726
		V	35.2		1 572	56.6		2 692	90.8		4 707
C only	PQClean	K	37.4		2 025	59.4		3 504	–[a]		–[a]
		S	50.7		8 034	77.7		12 987	–[a]		–[a]
		V	35.4		2 223	56.4		3 666	–[a]		–[a]
	This work	K	**4.9**	✓	**2 927**	**6.4**	✓	**5 112**	**7.9**	✓	**8 609**
		S	**5.0**	✓	**18 470**	**6.5**	✓	**36 303**	**8.1**		**44 332**
		V	**2.7**	✓	**4 036**	**2.7**	✓	**7 249**	**2.7**	✓	**12 616**

[a] Implementation disabled because the device does not have enough RAM to support it.

plementation from [7] and the implementation from [1] to outperform our implementation on the Cortex-M4: we use a slower approach and a generic implementation. In order to assess the impact of the proposed techniques we remove the optimized assembly implementation from the equation and compare to the generic PQClean implementation. We include the performance figures of the other implementations for the sake of completeness.

An overview of the results is provided in Tables 2. The testing platform that we used is the STM32F4 Discovery board, which is based on the STM32F407 microcontroller. Our implementation was benchmarked using the pqm4 framework. To obtain the cycle counts we measured 10 000 executions and computed the average. The results for the pqm4 ([7]) and [1] implementations are based on the results listed in [1].

The pqm4 method for measuring a scheme's memory usage is to first fill the stack with dummy values, then run the algorithm, and count how many dummy values were overwritten. The speed of the scheme is measured by measuring how much the SysTick timer has advanced while running the algorithm. The SysTick timer is clocked with the same frequency as the CPU, so this gives us the algorithm's latency in number of cycles (cc). In order to eliminate the influence of the chip's flash latency on the benchmarking results, the STM32F4 chip is clocked at 24 MHz, and the flash wait-states are set to zero. The code was compiled using GCC version 9.2.1,[6] with optimization level `-Os`.

In Table 3 we have listed the code sizes for all the implementations that we compare in Table 2. We have measured these code sizes using the same settings as for the memory/performance measurements. Because the [7] pqm4 and the [1] implementations are optimized for speed, we have listed their code sizes for the optimization levels `-O3` and `-Os`. In these metrics, the contribution of symmetric primitives—e.g., the size of the SHAKE code—has been excluded.

Discussion. The memory footprints reported in Table 2 for the presented techniques are close to the lower bounds provided earlier. The discrepancy in memory use is around 0.4 KiB of memory for all algorithms. The largest contributor to this additional memory use is the execution of SHAKE.

[6] `arm-none-eabi-gcc (15:9-2019-q4-0ubuntu1) 9.2.1 20191025 (release) [ARM/arm-9-branch revision 277599]`

Table 3. Code sizes of the implementations from Table 2 expressed in bytes. Opt-level denotes the optimization level that was used. Contribution of Keccak and AES code is excluded from all implementations.

implementation	opt-level	Dilithium-2	Dilithium-3	Dilithium-5
[7] (pqm4)	-O3	10 564	10 092	–[a]
[1]	-O3	18 448	19 916	18 262
[7] (pqm4)	-Os	9 700	9 276	–[a]
[1]	-Os	17 408	19 012	17 234
PQClean	-Os	6 986	6 534	–[b]
This work	-Os	**10 091**[c]	**10 091**[c]	**10 091**[c]

[a] Not reported by pqm4.
[b] Implementation disabled because the device does not have enough RAM to support it.
[c] Implementation includes support for all Dilithium variants.

The SHAKE code, which has been unadapted from the Dilithium reference implementation uses around 300 bytes of stack. The last 100 bytes are found in call-tree information and temporary buffers used during the packing and unpacking of polynomials into bit-arrays.

Table 2 clearly shows that the proposed techniques pay off. The states of both Dilithium-2 and Dilithium-3 for signature generation, verification and key generation easily fit into 8 KiB. It should be noted that none of the other high-speed implementations can execute on devices even with 32 KiB of memory. The amount of headroom arguably allows for plenty of other tasks to run on the device; 3.0 KiB in the case of Dilithium-2 and 1.4 KiB for Dilithium-3. The memory footprint of Dilithium-5 signing *just* exceeds 8 KiB. For Verify, the memory footprint is reduced to 2.7 KiB.

This is of course only half of the story. The memory reduction techniques have a clear impact on the performance of the scheme. When comparing cycle counts to those of the PQClean implementation (which is the implementation most similar to ours), one observes a factor $2.3-2.8$ slowdown for Sign and a factor $1.8-2.0$ slowdown for Verify. For both algorithms, the difference in performance is due to the overhead from the (24-bit) bit-packing operations in the matrix-vector multiplication, and the slower schoolbook method for multiplying $c\mathbf{t_0}$. For Dilithium-3 signing there is some additional overhead, because the $q' = 769$ NTTs are somewhat slower than the $q' = 257$ NTTs in the other variants.

Optimization efforts from [7] and [1] have lead to a $43\%-44\%$ reduction of cycles in Sign compared to the PQClean implementation. Similarly, one can expect that future performance enhancements will be able to improve the performance of our implementation of the memory reduction techniques as well. Depending on the platform, integrating more optimized assembly implementations for SHAKE, (inverse) NTT, and challenge multiplication could result in significant performance gains. In particular, many of the values in the challenge multiplication are 8 bits, This is suitable for parallel computation using SIMD instructions, which are not used in our C-implementation.

More importantly, many of the memory constrained devices come equipped with dedicated cryptographic coprocessors for symmetric primitives (such as SHAKE) as well as for big-number arithmetic. When one is able to make use of these coprocessors, the execution times could be reduced drastically: especially because SHAKE remains a dominating component of the Dilithium execution time as well as the polynomial multiplication [3].

Although the reduction of the run-time state has a big impact on the execution speed of the algorithm, we see from the results in Table 3 that this is not the case for the code size. The code for our new implementation is slightly bigger than the PQClean code, but about the same size as

the optimized implementations.[7] Moreover, we must take into account that our implementation supports *all* variants of Dilithium at the same time, so a slight increase is actually expected.

6 Conclusion

Although there is considerable performance impact when implementing Dilithium in a low-memory environment, we have shown that such low-memory Dilithium implementations are feasible in practice. In particular, we broke the 8 KiB memory barrier for Dilithium-2 and Dilithium-3. Dilithium-5 uses a *little* bit more memory than 8 KiB but we have shown that there are still time-memory tradeoffs that can be applied, even though these tradeoffs are relatively expensive in terms of performance.

When earlier work (like [24]) was published, it was not clear whether Dilithium was a scheme that could even be considered for memory constrained devices. Then [7] showed that the Dilithium algorithms could reasonably fit into 16 KiB of memory. In this paper, we show that most variants of Dilithium can even fit into 8 KiB without a very drastic impact on performance. More so, we reduced the memory footprint for Dilithium verification below 3 KiB. For memory-constrained devices, storing Dilithium's public keys and signatures has arguably become a bigger challenge than storing its run-time state.

References

1. Abdulrahman, A., Hwang, V., Kannwischer, M.J., Sprenkels, D.: Faster Kyber and Dilithium on the Cortex-M4. Cryptology ePrint Archive, Report 2022/112 (2022), https://eprint.iacr.org/2022/112
2. Albrecht, M.R., Hanser, C., Hoeller, A., Pöppelmann, T., Virdia, F., Wallner, A.: Implementing RLWE-based schemes using an RSA co-processor. IACR TCHES **2019**, 169–208 (Nov 2018). https://doi.org/10.13154/tches.v2019.i1.169-208, https://tches.iacr.org/index.php/TCHES/article/view/7338
3. Bos, J.W., Renes, J., van Vredendaal, C.: Polynomial multiplication with contemporary co-processors: Beyond Kronecker, Schönhage-Strassen & Nussbaumer (to appear). In: USENIX Security Symposium. USENIX Association (2022)
4. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) ITCS 2012. pp. 309–325. ACM (Jan 2012). https://doi.org/10.1145/2090236.2090262
5. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Dilithium: A lattice-based digital signature scheme. IACR TCHES **2018**(1), 238–268 (2018). https://doi.org/10.13154/tches.v2018.i1.238-268, https://tches.iacr.org/index.php/TCHES/article/view/839
6. Faz-Hernández, A., Kwiatkowski, K.: Introducing CIRCL: An Advanced Cryptographic Library. Cloudflare (Jun 2019), available at https://github.com/cloudflare/circl. v1.1.0 Accessed Feb 2022
7. Greconici, D.O.C., Kannwischer, M.J., Sprenkels, D.: Compact Dilithium implementations on Cortex-M3 and Cortex-M4. IACR TCHES **2021**(1), 1–24 (2021). https://doi.org/10.46586/tches.v2021.i1.1-24, https://tches.iacr.org/index.php/TCHES/article/view/8725
8. Harvey, D.: Faster polynomial multiplication via multipoint Kronecker substitution. Journal of Symbolic Computation **44**(10), 1502–1510 (2009), https://doi.org/10.1016/j.jsc.2009.05.004
9. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, https://github.com/mupq/pqm4
10. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. Workshop Record of the Second PQC Standardization Conference (2019)
11. Kannwischer, M.J., Schwabe, P., Stebila, D., Wiggers, T.: PQClean (to appear). Cryptology ePrint Archive, Report 2022/XXX (2022), https://eprint.iacr.org/2022/XXX

[7] It is clear that the hand-written assembly in the [7] and [1] implementations—which is very aggressively loop-unrolled—comes at a significant cost in code size.

12. Kiltz, E., Lyubashevsky, V., Schaffner, C.: A concrete treatment of Fiat-Shamir signatures in the quantum random-oracle model. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 552–586. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78372-7_18
13. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**, 203–209 (1987)
14. Kronecker, L.: Grundzüge einer arithmetischen Theorie der algebraischen Grössen. Journal für die reine und angewandte Mathematik **92**, 1–122 (1882), https://doi.org/10.1515/9783112342404-001
15. Langlois, A., Stehlé, D.: Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography **75**(3), 565–599 (2015), https://doi.org/10.1007/s10623-014-9938-4
16. Lyubashevsky, V.: Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 598–616. Springer, Heidelberg (Dec 2009). https://doi.org/10.1007/978-3-642-10366-7_35
17. Lyubashevsky, V.: Lattice signatures without trapdoors. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 738–755. Springer, Heidelberg (Apr 2012). https://doi.org/10.1007/978-3-642-29011-4_43
18. Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D., Bai, S.: CRYSTALS-DILITHIUM. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
19. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 1–23. Springer, Heidelberg (May / Jun 2010). https://doi.org/10.1007/978-3-642-13190-5_1
20. Maayan, G.D.: The IoT rundown for 2020: Stats, risks, and solutions. https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx
21. Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO'85. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (Aug 1986). https://doi.org/10.1007/3-540-39799-X_31
22. National Institute of Standards and Technology: Post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization
23. Peikert, C., Regev, O., Stephens-Davidowitz, N.: Pseudorandomness of ring-LWE for any ring and modulus. In: Hatami, H., McKenzie, P., King, V. (eds.) 49th ACM STOC. pp. 461–473. ACM Press (Jun 2017). https://doi.org/10.1145/3055399.3055489
24. Ravi, P., Gupta, S.S., Chattopadhyay, A., Bhasin, S.: Improving speed of Dilithium's signing procedure. In: Belaïd, S., Güneysu, T. (eds.) Smart Card Research and Advanced Applications – CARDIS. LNCS, vol. 11833, pp. 57–73. Springer (2019). https://doi.org/10.1007/978-3-030-42068-0_4
25. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) 37th ACM STOC. pp. 84–93. ACM Press (May 2005). https://doi.org/10.1145/1060590.1060603
26. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the Association for Computing Machinery **21**(2), 120–126 (1978), https://dl.acm.org/doi/10.1145/359340.359342