Improving Speed of Dilithium's Signing Procedure

Prasanna Ravi[1,2], Sourav Sen Gupta[2], Anupam Chattopadhyay[1,2], and Shivam Bhasin[1]

[1] Temasek Laboratories, Nanyang Technological University, Singapore
[2] School of Computer Science and Engineering
Nanyang Technological University, Singapore

prasanna.ravi@ntu.edu.sg sg.sourav@ntu.edu.sg anupam@ntu.edu.sg
sbhasin@ntu.edu.sg

Keywords: Dilithium, Early Evaluation, *pqm4*, Digital Signatures, Lattice-based cryptography, Post-Quantum Cryptography

Abstract. Dilithium is a round 2 candidate for digital signature schemes in NIST initiative for post-quantum cryptographic schemes. Since Dilithium is built upon the "Fiat Shamir with Aborts" framework, its signing procedure performs rejection sampling of its signatures to ensure they do not leak information about the secret key. Thus, the signing procedure is iterative in nature with a number of rejected iterations, which serve as unnecessary overheads hampering its overall performance. As a first contribution, we propose an optimization that reduces the computations in the rejected iterations through *early-evaluation* of the conditional checks. This allows to perform an early detection of the rejection condition and reject a given iteration as early as possible. We also incorporate a number of standard optimizations such as *unrolling* and *inlining* to further improve the speed of the signing procedure. We incorporate and evaluate our optimizations over the software implementation of Dilithium on both the Intel Core i5-4460 and ARM Cortex-M4 CPUs. As a second contribution, we identify opportunities to present a more refined evaluation of Dilithium's signing procedure in several scenarios where pre-computations can be carried out. We also evaluate the performance of our optimizations and the memory requirements for the pre-computed intermediates in the considered scenarios. We could yield speed-ups in the range of *6%* upto *35%*, considering all the aforementioned scenarios, thus presenting the fastest software implementation of Dilithium till date.

1 Introduction

It has been known for quite sometime that modern public-key cryptography that is being used today, is not secure against attacks by large-scale quantum computers [13]. With continued advances in the field of quantum computing [3], it will probably not be long before we have the world's first large scale quantum computer, that can break modern day public-key cryptography. This prompted

NIST to initiate a standardization process for public-key cryptographic schemes (public-key encryption, digital signatures, and key establishment schemes) that are secure against quantum computers [11]. NIST received 69 valid submissions for the first round of the standardization process. After intense scrutiny by NIST and based on public feedback, NIST selected 26 algorithms (17 public-key encryption and 9 digital signature schemes) for the second round out of the 69 valid submissions from the first round of the standardization process. The Dilithium lattice-based signature scheme, part of the CRYSTALS (Cryptographic Suite for Algebraic Lattices) package is one of the leading second-round candidates for digital signatures [8]. Dilithium offers both good security and efficiency guarantees with its security based on the efficient Module-Learning With Errors (MLWE) problem. Thus, most if not all computations in Dilithium involve operations over polynomials in a structured cyclotomic ring that allows use of the efficient Number Theoretic Transform (NTT) for polynomial multiplication.

However, one of the main features of Dilithium is that it is built upon the well-known *Fiat-Shamir with Aborts* framework [7]. The signing procedure performs *rejection sampling* of certain intermediate variables through a number of conditional checks. This is done to ensure that the generated signatures do not leak the distribution of the secret key. Thus, the signing procedure is iterative in nature and goes through a number of rejected iterations until it outputs a valid signature. For example, the signing procedure for recommended parameter sets of Dilithium has an average repetition rate of 6.6 [8] and hence the computations performed in all except the last iteration (5.6 iterations) are just un-necessary overheads. Thus, the repetition rate severely hampers the performance of Dilithium's signing procedure.

As a *first contribution*, we propose an optimization to perform *early-evaluation* of the conditional checks, so as to perform optimal number of computations to reject an iteration. Our high-level optimizations simply involve reorganization of computations within each iteration and hence can be adopted to speed-up both SW and HW implementations. Moreover, our optimizations could also be applicable to other lattice-based schemes built upon a similar framework. We also further enhance the performance of the signing procedure through techniques such as *unrolling* and *inlining* optimizations. The proposed optimizations do not create any secret key related timing dependency. We also identified opportunities to refine the approach to evaluate the signing performance of Dilithium in certain realistic scenarios where pre-computations are possible. We mainly consider two scenarios - (1) pre-computed intermediates in case of static public-private key pairs and (2) partitioning the signing procedure in case of the randomized variant of Dilithium. Thus as a *second contribution*, we perform a detailed evaluation of the performance improvements and the memory requirements for the above mentioned scenarios. We present results for the optimized signing procedure for different scenarios on both the Intel(R) Core(TM) i5-4460 CPU and observe speed-ups of *upto 31%* across all updated parameter sets of Dilithium. We also present the fastest software results for Dilithium on the ARM Cortex-M4F by optimizing the open-source implementation of Dilithium available in the open source *pqm4* library and observe speed-ups in the range of *6%* upto *35%*, thus demonstrating the portability of our optimizations across implementation platforms.

2 Preliminaries

Notation: Elements in the integer ring $\mathbb{Z}_q$ are denoted by regular font letters viz. $a, b \in \mathbb{Z}_q$, where q is a prime. We denote $x \xleftarrow{\$} X$ to denote sampling x uniformly in random from set X. We denote the polynomial ring $\mathbb{Z}_q[X]/\langle X^n + 1\rangle$ as R_q. For an element $\mathbf{a} \in R_q$, we define $\|\mathbf{a}\|_\infty = \max_{0 \le i \le n-1} |a(i) \pmod q|$. For a given $\eta \in \mathbb{N}$, define $S_\eta = \{\boldsymbol{a} \in R_q \mid \|\boldsymbol{a}\|_\infty \le \eta\}$. Multiplication of two polynomials $\mathbf{a}, \mathbf{b} \in R_q$ is denoted as $\mathbf{a} \cdot \mathbf{b}$ or $\mathbf{ab} \in R_q$. Matrices and vectors of polynomials in R_q are referred to as *modules* and are denoted using bold letters viz. $\mathbf{a} \in R_q^{k \times l}, \mathbf{b} \in R_q^l$. Each polynomial element of module $\mathbf{b} \in R_q^l$ is denoted as $\mathbf{b}[i]$ for $i \in [0, l-1]$.

Lattice-based Cryptography: The security of most efficient lattice-based cryptographic schemes are based on the hardness of two average case-hard problems known as the Ring-Learning With Errors problem (RLWE) [9] and the Ring-Short Integer Solutions problem (RSIS) [10]. Both these problems reduce to corresponding worst-case instances of hard problems over ideal lattices. For a public key $(\mathbf{a}, \mathbf{t}) \in (R_q, R_q)$, an RLWE attacker is asked to solve for polynomials $\mathbf{s}_1, \mathbf{s}_2 \in S_\eta$ with $\eta \ll q$ such that $\mathbf{t} = \mathbf{a} \cdot \mathbf{s}_1 + \mathbf{s}_2$. Given m uniformly random elements $\mathbf{a}[i] \in R_q$ for $i \in [0, m-1]$, an MSIS attacker is required to solve for a short non-zero vector $\mathbf{z} = (\mathbf{z}[0], \mathbf{z}[1], \ldots, \mathbf{z}[m-1]) \in S_\eta^m$ such that $\sum_i^m \mathbf{a}[i] \cdot \mathbf{z}[i] = 0 \in R_q$.

The RLWE and RSIS problems generalize to the corresponding Module-LWE (MLWE) and Module-SIS (MSIS) problems respectively, where computations are performed over matrices and vectors of polynomials in the space $R_q^{k \times \ell} = \mathbb{Z}_q^{k \times \ell}[X]/(X^n + 1)$ for $k, \ell > 1$ (as opposed to R_q for their ring variants). The generalized module version of the LWE and SIS problems also provide better security guarantees compared to their corresponding ring variants. A change in security of a scheme based on MLWE or MSIS only requires to alter the module dimensions k, ℓ while keeping the underlying operating ring fixed. Thus, change in security can be easily achieved through very minimal changes in the underlying implementation.

2.1 Dilithium

The security of Dilithium is based on the MLWE and MSIS problems. While the property of indistinguishability of the public key comes from the MLWE problem, security against existential forgery under the quantum random oracle model is based on MSIS hardness assumption[8]. Based on how the ephemeral nonce in the signing procedure is generated, Dilithium comes in two variants (i.e) deterministic or probabilistic.

In the following discussion, we discuss the details of the Dilithium signature scheme with more focus on its signing procedure[8]. The signature scheme is based on the "Fiat-Shamir with Aborts" framework [7] while the scheme itself derives from the lattice-based signature scheme proposed by Bai and Galbraith[2]. The scheme operates over the base ring R_q with $n, q = (256, 8380417)$ while offering flexibility with the module parameters (k, ℓ) allowing to operate over varying dimensions

$(k \times \ell)$ in four different security levels henceforth referred to as Dilithium1 (Weak), Dilithium2 (Medium), Dilithium3 (Recommended) and Dilithium4 (Very High).

Algorithm 1: Dilithium Signature scheme

1 **Procedure** Sign(sk, M)
2 $\quad \mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
3 $\quad \mu = \mathsf{CRH}(\mathsf{tr} \| M)$
4 $\quad \kappa = 0, (\mathbf{z}, \mathbf{h}) = \perp$
5 $\quad \rho' \in \{0,1\}^{384} := \mathsf{CRH}(K \| \mu)$(or $\rho' \leftarrow \{0,1\}^{384}$ for randomized signing)
6 $\quad$ **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
7 $\quad\quad \mathbf{y} \in S_{\gamma_1 - 1}^{\ell} := \mathsf{ExpandMask}(\rho' \| \kappa)$
8 $\quad\quad \mathbf{w} = \mathbf{A} \cdot \mathbf{y}$
9 $\quad\quad (\mathbf{w}_1, \mathbf{w}_0) = \mathsf{D}_q(\mathbf{w}, 2\gamma_2)$
10 $\quad\quad \mathbf{c} \in B_{60} = H(\mu \| \mathbf{w}_1)$
11 $\quad\quad \mathbf{z} = \mathbf{y} + \mathbf{c} \cdot \mathbf{s}_1$
12 $\quad\quad (\mathbf{r}_1, \mathbf{r}_0) := \mathsf{D}_q(\mathbf{w} - \mathbf{c} \cdot \mathbf{s}_2, 2\gamma_2)$
13 $\quad\quad$ **if** $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$ or $\mathbf{r}_1 \neq \mathbf{w}_1$ **then**
14 $\quad\quad\quad (\mathbf{z}, \mathbf{h}) = \perp$
15 $\quad\quad$ **else**
16 $\quad\quad\quad \mathbf{h} = \mathsf{MH}_q(-\mathbf{c} \cdot \mathbf{t}_0, \mathbf{w} - \mathbf{c} \cdot \mathbf{s}_2 + \mathbf{c} \cdot \mathbf{t}_0, 2\gamma_2)$
17 $\quad\quad\quad$ **if** $\|\mathbf{c} \cdot \mathbf{t}_0\|_\infty \geq \gamma_2$ or $\mathsf{wt}(\mathbf{h}) > \omega$ **then**
18 $\quad\quad\quad\quad (\mathbf{z}, \mathbf{h}) = \perp$
19 $\quad\quad$ **end**
20 $\quad\quad \kappa = \kappa + 1$
21 $\quad$ **end**
22 $\quad$ **return** $\sigma = (\mathbf{z}, \mathbf{h}, \mathbf{c})$

Key Generation: The main operation of the key generation procedure is to generate the MLWE instance that forms the public-private key pair (pk, sk) of Dilithium. An LWE instance $\mathbf{t} = \mathbf{a} \cdot \mathbf{s}_1 + \mathbf{s}_2$ is created where $\mathbf{a} \in R_q^{k \times \ell}$ is the public parameter while the secret and error modules $\mathbf{s}_1 \in R_q^{\ell}$ and $\mathbf{s}_2 \in R_q^{k}$ are small modules sampled from S_η^{ℓ} and S_η^{k} respectively. The LWE instance is not directly output as the public key but is decomposed into $\mathbf{t}_0, \mathbf{t}_1$ such that $\mathbf{t}_0$ consists of the d lower order bits of all coefficients of $\mathbf{t}$ while $\mathbf{t}_1$ consists of its remaining higher order bits. Subsequently, $\mathbf{t}_1$ is published as part of the public key pk while $\mathbf{t}_0$ along with $\mathbf{s}_1, \mathbf{s}_2$ form part of the secret key sk.

Signing: Refer to Alg.1 for the signing procedure of Dilithium. The signing procedure is iterative in nature (While loop from Line 6 to 21 of Sign in Alg.1) with a number of conditional checks (Line 13 and 17) and it exits with a valid signature only when all the conditional checks are successfully passed. Moreover, these selective rejections in the signing procedure together ensure both security and 100% correctness of the signature scheme.

The most important component of the signing procedure (apart from the secret key) is the ephemeral nonce $\mathbf{y} \in R_q^\ell$. Knowledge of a single value of $\mathbf{y}$ or reuse of $\mathbf{y}$ for different messages leads to a trivial break of the signature scheme. Moreover, the method of generation of the ephemeral nonce $\mathbf{y}$ also determines the deterministic nature of the signature scheme. In deterministic Dilithium, $\mathbf{y} \in R_q^\ell$ is deterministically generated using the ExpandMask function which takes as inputs, the message μ to be signed, a random secret key component $K \in \{0,1\}^{256}$ and the iteration count k (Line 5 and 7). But, in case of probabilistic Dilithium, $\mathbf{y}$ is randomly generated using the same ExpandMask function but with inputs, ρ' and the iteration count k where ρ' is sampled randomly from $\{0,1\}^{384}$ (Line 5).

Once $\mathbf{y}$ is sampled, the product $\mathbf{w} = \mathbf{a} \cdot \mathbf{y} \in R_q^k$ is computed and further decomposed into $\mathbf{w}_1$ and $\mathbf{w}_0$ such that $\mathbf{w} = \mathbf{w}_1 \cdot 2\gamma_2 + \mathbf{w}_0$. Further, a sparse challenge polynomial $\mathbf{c}$ (only 60 non-zero coefficients in either ± 1) is generated by hashing together the message, ephemeral nonce and public key information. Using $\mathbf{c}$ and $\mathbf{y}$, the signer generates the primary signature component $\mathbf{z}$ as $\mathbf{z} = \mathbf{s}_1\mathbf{c} + \mathbf{y}$. Finally, a hint vector $\mathbf{h} \in R_q^k$ with coefficients in $\{0,1\}$ is generated and is also published along with $\mathbf{z}, \mathbf{c}$ as the signature. This hint vector $\mathbf{h}$ is actually used by the verifier along with the signature to recover the value of $\mathbf{w}_1$ which is used to verify the authenticity of the challenge polynomial $\mathbf{c}$. We do not discuss the verification procedure and the reader is referred to [8] for more details.

3 Early Evaluation Optimization

Referring to the Sign procedure in Alg.1, we provide the following terminologies for the various conditional/rejection checks. It is important to note that all these checks have to be passed together in a single iteration, in order to output a valid signature.

- $\|\mathbf{z}\|_\infty \leq \gamma_1 - \beta$: Chk_Norm($\mathbf{z}$) (Line 13 of Sign in Alg.1)
- $\|\mathbf{r}_0\|_\infty \leq \gamma_2 - \beta$: Chk_Norm($\mathbf{r}_0$) (Line 13)
- $\|\mathbf{ct}_0\|_\infty \leq \gamma_2$: Chk_Norm($\mathbf{ct}_0$) (Line 17)
- $wt(\mathbf{h}) < w$: Chk_Weight($\mathbf{h}$) (Line 17)
- $\mathbf{r}_1 \neq \mathbf{w}_1$ (Line 13)

We make a couple of observations about implementation of the rejection checks in the reference implementation of Dilithium submitted to the NIST standardization process [8]. For reference, we consider the same code snippet in Fig.1 which contains operations corresponding to the computation of $\mathbf{z}$ followed by its corresponding rejection check Chk_Norm($\mathbf{z}$)[1].

- **Observation-1:** Out of the five rejection checks, the three rejection checks Chk_Norm($\mathbf{z}$), Chk_Norm($\mathbf{r}_0$) and Chk_Norm($\mathbf{ct}_0$) contribute to more than 99% of the rejections in the signing procedure. They are all *infinity_norm* checks (Chk_Norm) over modules with multiple polynomials.

[1] The code snippet shown in Fig.1 is in its static single assignment form. In the static single assignment code, the result of an operation is always written to a new variable. In the original implementation, all of $\mathbf{z}_i$ for $i = \{0, \ldots, 3\}$ refer to a single variable $\mathbf{z}$. The single assignment form is used for better illustration of our idea.

```
1   for(i = 0; i < L; ++i)
2   {
3     poly_pointwise_invmontgomery(z1.vec+i, &chat, s1.vec+i);
4     poly_invntt_montgomery(z2.vec+i,z1.vec+i);
5   }
6   polyvecl_add(&z3, &y, &z2);
7   polyvecl_freeze(&z,$z3);
8   if(polyvecl_chknorm(&z, GAMMA1 - BETA))
9     goto rej;
```

Fig. 1: C-Code snippet corresponding to computation of $\mathbf{z}$ according to the reference implementation in static single assignment form

Infinity norm checks are necessary conditions and computed one coefficient at a time. Considering $\mathsf{Chk\_Norm}(\mathbf{z})$, all individual polynomials $\mathbf{z}[i]$ for $i \in \{0, L-1\}$ of $\mathbf{z}$ are supposed to pass the check, for the complete module $\mathbf{z}$ to be considered valid. Hence, an iteration can be immediately rejected upon detecting a violation in any of the polynomials of $\mathbf{z}$. Lets assume a case where the first polynomial of $\mathbf{z}[0]$ violates $\mathsf{Chk\_Norm}(\mathbf{z})$. Though the violation can be detected just by computing the first polynomial $\mathbf{z}[0]$, analysis of the reference implementation of Dilithium revealed that all polynomials of $\mathbf{z}$ are computed before the conditional check over the whole module of $\mathbf{z}$ is performed. If $\mathbf{z}[0]$ can be computed independently and checked immediately, then one can immediately reject the iteration saving the un-necessary computations of $\mathbf{z}[1], \ldots, \mathbf{z}[L-1]$. The same applies to the other two $\mathsf{Chk\_Norm}$ conditions over $\mathbf{r}_0$ and $\mathbf{ct}_0$ in the signing procedure.

Hence, we alternately propose to *compute and check* $\mathbf{z}$ *one polynomial at a time* (instead of *one module at a time* in the reference implementation). Only if the check over a particular polynomial $\mathbf{z}[i]$ is passed, the next polynomial $\mathbf{z}[i+1]$ is computed, else the iteration is rejected immediately. We also make the following observation.

- **Observation-2:** The module $\mathbf{z}$ is computed over a series of computations ($\mathbf{z}_1 \rightarrow \mathbf{z}_2 \rightarrow \mathbf{z}_3 \rightarrow \mathbf{z}$) with each computation ($\mathsf{poly\_pointwise\_invmontgomery}$, $\mathsf{poly\_invntt\_montgomery}$, $\mathsf{polyvecl\_add}$ and $\mathsf{polyvecl\_freeze}$) operating over the entire module. But, all these computations preceding the rejection check can also independently operate over single polynomials and do not have any dependency over other polynomials in the same module.

This enables us to *chain* these computations corresponding to single polynomials and compute $\mathbf{z}$ one polynomial at a time. The same technique can also be applied to the computation of $\mathbf{r}_0$ and $\mathbf{ct}_0$ pertaining to the two other rejection checks (though the computations involved are slightly different). For a better illustration, the compute chain of $\mathbf{z}$ corresponding to the original implementation can be depicted as in Eqn.1 as follows:

$$(\mathbf{z}_1[0] \rightarrow \mathbf{z}_1[1] \rightarrow \ldots \rightarrow \mathbf{z}_1[L-1]) \rightarrow (\mathbf{z}_2[0] \rightarrow \mathbf{z}_2[1] \rightarrow \ldots \rightarrow \mathbf{z}_2[L-1]) \rightarrow \ldots$$
$$(\mathbf{z}_3[0] \rightarrow \mathbf{z}_3[1] \rightarrow \ldots \rightarrow \mathbf{z}_3[L-1]) \rightarrow (\mathbf{z}[0] \rightarrow \mathbf{z}[1] \rightarrow \ldots \rightarrow \mathbf{z}[L-1]) \quad (1)$$

From Eqn.1, we can see that a particular computation is performed over every polynomial in the module before starting the next computation. But our optimized technique computes $\mathbf{z}$ according to the compute chain as depicted in Eqn.2:

$$(\mathbf{z}_1[0] \rightarrow \mathbf{z}_2[0] \rightarrow \mathbf{z}_3[0] \rightarrow \mathbf{z}[0]) \rightarrow (\mathbf{z}_1[1] \rightarrow \mathbf{z}_2[1] \rightarrow \mathbf{z}_3[1] \rightarrow \mathbf{z}[1]) \rightarrow \ldots$$
$$(\mathbf{z}_1[2] \rightarrow \mathbf{z}_2[2] \rightarrow \mathbf{z}_3[2] \rightarrow \mathbf{z}[2]) \rightarrow \ldots \rightarrow$$
$$(\mathbf{z}_1[L-1] \rightarrow \mathbf{z}_2[L-1] \rightarrow \mathbf{z}_3[L-1] \rightarrow \mathbf{z}[L-1]) \quad (2)$$

In fact, the above compute chain is not always fully computed and is *halted* at the earliest possible instance as every polynomial $(\mathbf{z}[0], \ldots, \mathbf{z}[L-1])$ is immediately checked after it is computed. This is in contrast to the reference implementation where the compute chain is always fully computed before the whole of $\mathbf{z}$ is checked. Going one step further, we also observe that the set of consecutive computations including the rejection check (i.e) (polyvecl_add, polyvecl_freeze and polyvecl_chknorm) are actually point-wise operations which operate over single coefficients. Thus, it is possible to combine these consecutive operations into a single composite operation, thus bringing our optimization from the *polynomial level* down to the *coefficient level*. Refer to Fig.2 for the code-snippet of the optimized computation of $\mathbf{z}$, wherein computations are performed *one polynomial at a time*. Furthermore, the identified consecutive point-wise operations are further fused into a single function (poly_add_freeze_chk_norm) which computes and immediately checks each coefficient before moving onto the next. These optimizations also directly apply to the other rejection checks involving $\mathbf{r}_0$ and $\mathbf{ct}_0$. We will henceforth refer to it as the *Early-Eval* optimization throughout the paper. Since it mainly works to remove un-necessary computations, we can clearly see that it will benefit serial implementations much more than parallel implementations. While we expect to observe maximum speed-up for serial implementations (HW/SW) which iterate over computations corresponding to *one polynomial at a time*, we would only observe negligible/no speed-up in *embarrassingly parallel* HW implementations which parallelize computations corresponding to all polynomials of the module.

3.1 Note on Timing Attacks:

Any given iteration of our signing procedure in our optimized implementation is immediately *rejected* as soon as a coefficient that violates a conditional check is computed. Thus, any adversary with access to the timing side-channel may potentially derive information about the position of the coefficient which resulted in rejection. However, the probability of a given coefficient violating the bound is independent of the secret key and thus knowledge of the position of the coefficient that resulted in rejection does not leak any exploitable information about the secret key. Thus, to the best of our knowledge, our *Early-Eval* optimization does not bring in any additional exploitable timing vulnerabilities.

```
1   for(i = 0; i < L; ++i)
2   {
3     poly_pointwise_invmontgomery(z.vec+i, &chat, s1.vec+i);
4     poly_invntt_montgomery(z.vec+i);
5     if(poly_add_freeze_chk_norm(z.vec+i, z.vec+i,
6         y.vec+i, GAMMA1 - BETA))
7           goto rej;
8   }
```

Fig. 2: C-Code snippet of computation of **z** improved using our *Early-Eval* optimization

3.2 Additional optimizations

While implementing the proposed optimization on the public code of *pqm4* library, we observed some potential scope for further optimizations. Though these optimizations might be intuitively known and not necessarily novel, we included these optimizations to test the limits of speed-up that can be achieved. We observed that the reference implementation of Dilithium consists of a large number of functions which operate over single coefficients. These functions were implemented in separate files and were compiled into separate object files and hence the compiler couldn't *inline* them automatically. With these computations spanning over multiple polynomials each of degree 256, the overhead from just function calls (branch to and from the functions) in these point-wise functions are significant. Hence, we resorted to manually *inlining* all the point-wise functions used in the implementation. Though inlining doesn't result in very elegant code, it avoids the un-necessary overhead from branching to and from the function for every coefficient.

We also incorporated another standard optimization of *unrolling* the loops in all the small functions that computed over single coefficients. We limited the unroll factor to 8 for all such loops within these functions so as to maintain the readability and simplicity of the code. We henceforth refer to these optimizations as the *Impl-Level* optimizations throughout the paper. It is important to note that the *Impl-Level* optimizations are applied to all point-wise/coefficient-wise operations within the Dilithium signature scheme, while our *Early-Eval* optimizations only apply to the few operations preceding the conditional checks within the signing procedure. Though the *Impl-level* optimizations speed up all the three procedures of Dilithium (KeyGen, Sign and Verify), we limit our focus only to the performance improvements of its signing procedure.

4 Experimental Results

In this section, we perform an experimental evaluation of our optimizations over the Dilithium's signing operation on two software platforms (1) Intel Core i5 CPU and (2) ARM Cortex-M4 MCU. Our optimizations were incorporated over the updated reference implementation of Dilithium submitted to the *second* round of the ongoing NIST standardization process. It is possible to independently

employ both the *Early-Eval* and *Impl-Level* optimizations and thus we present two different optimized implementations of the signing operation (Refer Tab.1). While the proposed Opt-1 variant demonstrates the speed-up only due to the *Early-Eval* optimization, the Opt-2 variant demonstrates the speed-up from the combination of both the *Early-Eval* and *Impl-Level* optimizations.

Table 1: Different variants of Dilithium's signing procedure based on the employed optimizations

Variant	Optimization Used
Ref	None
Opt-1	*Early-Eval*
Opt-2	*Early-Eval* & *Impl-Level*

4.1 A Refined Evaluation Approach

While experimenting with the implementation of Dilithium's signing procedure, we found that it can be further refined when considering its practical usage in certain realistic scenarios. The main factor we consider is the *cryptoperiod* of the public-private key pair. According to the NIST SP 800-57 Part-1 document on "Recommendation for Key Management", "a *cryptoperiod* is the time span during which a specific key is authorized for use by legitimate entities, or the keys for a given system will remain in effect." NIST dedicates a complete section on cryptoperiods and details on the various risk factors, consequence factors and recommendations that allows one to decide the cryptoperiod for the various keys used in any secure application. The reader is referred to Section 5.3 of the SP 800-57 document [4] for more in-depth details.

4.1.1 Precomputing operations over the static public-private key pair

NIST recommends that a private signature key can have a cryptoperiod of about 1-3 years at the signer's side while the public signature key used for verification could be valid for several years depending on the key size [4]. Though these are mere recommendations from NIST and not strict guidelines, considering the complexity of repeatedly refreshing key-pairs from the perspective of a key-management system, one can expect most secure applications to work with static public-private key pairs with relatively long cryptoperiods. We observed a number of operations within Dilithium's signing procedure which operate over the static public-private key pair. But, in situations where public-private key pairs are static, these operations can simply be computed once and have its results reused to avoid unnecessary overheads from performing redundant computations.

To be specific, operations such as expanding a seed into the public parameter $\mathbf{A}$, unpacking the secret key sk into its individual components and NTT operations over the secret key components $\mathbf{s}_1$,$\mathbf{s}_2$,$\mathbf{t}_0$ are redundant if the public-private key pairs are static[2]. Thus, we consider the following two scenarios for evaluation based on the cryptoperiod of the public-private key pair. We denote:

– Scenario-1 : All operations are computed *online* assuming ephemeral public-private key pairs.
– Scenario-2 : Certain operations are *pre-computed offline* assuming static public-private key pairs with very long cryptoperiods.

4.1.2 Partitioning the Signing Procedure

Considering the randomized variant of Dilithium's signing procedure, we observe that some more operations within the signing procedure can be computed offline, independent of the message to be signed. In particular, operations such as sampling $\mathbf{y}$ using ExpandMask (Line 7 of Sign in Alg.1) and computation of $\mathbf{w}_0$ and $\mathbf{w}_1$ (Lines 8 & 9) can be computed offline. If we also assume static public-private key pair, it is possible to split the signing procedure into offline and online phases. Such partitioning techniques can significantly speed-up the signing procedure in real-time applications with main focus on low-latency times. In such scenarios, computations in lines 2,7,8 and 9 of Sign procedure in Alg.1 can be performed offline assuming that the device has a large enough buffer to store all the intermediates. The remaining operations can be computed online upon knowledge of the message to be signed. In fact, Aysu *et al.* in [1] utilized the same partitioning technique in their high-performance and low-latency HW-SW co-designed implementation of the GLP lattice-based signature scheme [5]. A similar idea of partitioning was also suggested by Pöppelmann *et al.* in [12] to improve the speed of the BLISS lattice-based signature scheme. We denote:

– Scenario-3 : Considering the randomized variant of Dilithium, we assume all message independent operations along with operations over the static public-private secret key to be computed offline. Thus, we only evaluate the performance of online phase of the signing procedure.

4.2 Results on the Intel Core i5-4460 CPU

We first present results of our optimized implementations of Dilithium's signing procedure on the Intel Core i5-4460 CPU 3.20GHz, compiled with gcc-4.2.1 without modifying the compiler flags set for the reference implementation. We use the average computational run-times of the signing procedure as our evaluation metric, which was obtained across 10^6 runs of the signing procedure. We tested two versions of Dilithium (i.e) (1) Dilithium-SHA that uses SHAKE from the SHA3 family as an XOF and (2) Dilithium-AES that uses AES-256 in counter mode as an XOF, across all parameter sets of Dilithium. Refer Tab.2-3 for a comparative performance evaluation of our optimized implementations (Opt-1 and Opt-2) against the reference implementation, in all the three identified scenarios (in terms of number of clock cycles). While we use the randomized variant of Dilithium for evaluation in Scenario-3 as stated earlier, we use the deterministic variant with the same secret key and message inputs for a direct comparative evaluation in Scenario-1 and Scenario-2.

[2] The authors of Dilithium also note that the above operations can be pre-computed and stored to "slightly" speed up the signing operation, but do not present any performance evaluation or the memory requirements due to the same (Refer Sec.3.1 of [8]).

Table 2: Comparative performance evaluation of the optimized Opt-1 implementation variant against the reference implementation of Dilithium's signing procedure on the Intel Core i5-4460 CPU. The results are reported in units of **million** (10^6) **clock cycles**.

Scheme	Cycles ($\times 10^6$)								
	Scenario-1			Scenario-2			Scenario-3		
	Ref	**Opt-1**	**Imp. (%)**	**Ref**	**Opt-1**	**Imp. (%)**	**Ref**	**Opt-1**	**Imp. (%)**
Dilithium1-SHA	0.904	0.833	**7.8**	0.778	0.715	**8.08**	0.365	0.303	**16.88**
Dilithium2-SHA	1.621	1.461	**9.88**	1.378	1.246	**9.57**	0.598	0.457	**23.5**
Dilithium3-SHA	2.359	2.153	**8.69**	2.042	1.838	**10.0**	0.812	0.598	**26.2**
Dilithium4-SHA	2.183	2.035	**6.77**	1.731	1.586	**8.38**	0.694	0.548	**20.95**
Dilithium1-AES	1.156	1.094	**5.33**	0.910	0.863	**5.21**	0.365	0.303	**17.01**
Dilithium2-AES	2.110	1.973	**6.919**	1.663	1.526	**8.23**	0.589	0.457	**22.4**
Dilithium3-AES	3.175	2.969	**6.498**	2.460	2.258	**8.17**	0.814	0.597	**26.5**
Dilithium4-AES	3.174	2.970	**6.414**	2.459	2.258	**8.18**	0.817	0.600	**26.5**

We first compare the runtimes of the reference implementations of the signing procedure in the three identified scenarios. Comparing Scenario-1 and Scenario-2, we observe a difference of about $13 - 14\%$ in runtime for Dilithium-SHA and $20 - 21\%$ for Dilithium-AES, which corresponds to the time spent on performing redundant operations over the static public-private key pair. When comparing Scenario-1 and Scenario-3, we observe a large difference of about 60% for Dilithium-SHA and $71 - 72\%$ for Dilithium-AES, which shows that a significant amount of time within each iteration is spent in sampling the ephemeral nonce $\mathbf{y}$ using XOF functions either through Keccak permutations in case of Dilithium-SHA and AES-256 in counter mode in case of Dilithium-AES. This difference also arises from computation of associated variables $\mathbf{w}_1$ and $\mathbf{w}_0$ in each iteration, but is very small when compared to the time taken from sampling $\mathbf{y}$.

We now perform a performance comparison of our optimized implementations against the reference implementations on the Intel i5-CPU, individually based on the different identified scenarios (Refer Tab.2-3). Considering Scenario-1, where all operations are done online, we observe a speed-up of about $6.7 - 9.8\%$ and $5.3 - 6.9\%$ for the Opt-1 implementation of Dilithium-SHA and Dilithium-AES respectively. But, our proposed Opt-2 variant which is additionally padded with *Impl-Level* optimizations yields a much higher speed-up of $17 - 21\%$ for Dilithium-SHA and $13.5 - 15.7\%$ for Dilithium-AES in Scenario-1. Considering Scenario-2, where the operations over the static public-private key pair are pre-computed, we observe improved speed-ups of about $8 - 10\%$ and $5.2 - 8.1\%$ for the Opt-1 implementation of Dilithium-SHA and Dilithium-AES respectively. But, our Opt-2 implementation shows an improved speed-up of about $20 - 23\%$ for Dilithium-SHA

and $16-20\%$ for Dilithium-AES in Scenario-2. The improved speed-up in Scenario-2 is mainly observed due to removal of the overheads due to operations over the static public-private key pair in all the compared implementations (Ref, Opt-1, Opt-2).

Table 3: Comparative performance evaluation of the optimized implementation Opt-2 against the reference implementation of Dilithium's signing procedure on the Intel(R) Core(TM) i5-4460 CPU. The results are reported in units of **million (10^6) clock cycles**.

Scheme	Cycles ($\times 10^6$)								
	Scenario-1			Scenario-2			Scenario-3		
	Ref	Opt-2	Imp. (%)	Ref	Opt-2	Imp. (%)	Ref	Opt-2	Imp. (%)
Dilithium1-SHA	0.904	0.742	**17.8**	0.778	0.617	**20.07**	0.365	0.280	**23.2**
Dilithium2-SHA	1.621	1.281	**20.9**	1.378	1.069	**22.4**	0.598	0.424	**29.1**
Dilithium3-SHA	2.359	1.86	**21.1**	2.042	1.545	**24.3**	0.812	0.557	**31.38**
Dilithium4-SHA	2.183	1.771	**18.85**	1.731	1.320	**23.7**	0.694	0.505	**27.2**
Dilithium1-AES	1.156	0.999	**13.55**	0.910	0.758	**16.6**	0.365	0.281	**23.04**
Dilithium2-AES	2.110	1.79	**15.15**	1.663	1.341	**19.3**	0.589	0.426	**27.59**
Dilithium3-AES	3.175	2.676	**15.72**	2.460	1.966	**20.0**	0.814	0.557	**31.53**
Dilithium4-AES	3.174	2.677	**15.65**	2.459	1.966	**20.0**	0.817	0.557	**31.7**

Considering Scenario-3, where we only evaluate the online phase of the signing procedure, we observe much higher speed-ups of about $16.9-23.5\%$ and $17.0-26.5\%$ for the Opt-1 implementation of Dilithium-SHA and Dilithium-AES respectively. But, the more optimized Opt-2 implementation yields significant speed-ups of about $23.2-31.4\%$ and $23.0-31.7\%$ for Dilithium-SHA and Dilithium-AES respectively in Scenario-3. The best speed-ups were observed in Scenario-3 because all the operations in the online phase of the signing procedure are enhanced by our optimizations. This is unlike Scenario-1 and Scenario-2, where the major computational time of the signing procedure was dominated by the XOF functions which are *unaffected* by either of our optimizations.

4.3 Results on the ARM Cortex-M4

In the following, we present results of our optimized implementations on the ARM Cortex-M4 MCU. We port our optimizations onto the publicly available implementation of Dilithium taken from the *pqm4* library [6], a benchmarking and testing framework for PQC schemes on the ARM Cortex-M4 family of microcontrollers. Our implementations were compiled with arm-none-eabi-gcc-7.2.1 with compiler flags `-O3 -mthumb -mcpu=cortex-m4 -mfloat-abi=hard -mfpu=fpv4-sp-d16` and run on

the STM32F4DISCOVERY board (DUT) housing the STM32F407, ARM Cortex-M4 microcontroller. Since we observe similar if not better speed-ups for both our Opt-1 and Opt-2 implementation variants on the ARM Cortex-M4 MCU when compared to the Intel CPU, we only provide detailed evaluation of our fastest Opt-2 implementation in Tab.4. These results were obtained across 10k runs of the signing procedure of Dilithium-SHA across all parameter sets. However, for the sake of completeness, we provide results for our Opt-1 variant on the recommended parameter set of Dilithium, Dilithium-3. Considering Scenario-1 for our Opt-2

Table 4: Performance evaluation of the reference, Opt-1 and Opt-2 implementation of Dilithium's signing procedure on the ARM Cortex-M4 MCU. The results are reported in units of **million (10^6) clock cycles**.

Scheme	**Cycles** ($\times 10^6$)								
	Scenario-1			**Scenario-2**			**Scenario-3**		
	Ref	**Opt-1**	**Imp. (%)**	**Ref**	**Opt-1**	**Imp. (%)**	**Ref**	**Opt-1**	**Imp. (%)**
Dilithium3-SHA	8.907	8.332	**6.45**	7.292	6.78	**7.01**	2.239	1.716	**23.35**
	Ref	**Opt-2**	**Imp. (%)**	**Ref**	**Opt-2**	**Imp. (%)**	**Ref**	**Opt-2**	**Imp. (%)**
Dilithium1-SHA	3.033	2.482	**18.16**	2.493	1.950	**21.76**	1.016	0.721	**29.08**
Dilithium2-SHA	5.761	4.632	**19.59**	4.752	3.640	**23.41**	1.630	1.085	**33.42**
Dilithium3-SHA	8.907	7.085	**20.45**	7.292	5.495	**24.64**	2.237	1.449	**35.21**
Dilithium4-SHA	8.648	7.061	**18.34**	6.283	4.733	**24.67**	1.916	1.274	**33.49**

variant, we observe speed-ups of about $18-20\%$ while for Scenario-2 we observe increased speed-ups in the range of $21-24\%$ across all parameter sets of Dilithium. As for Scenario-3, we observe a significant speed-up of about $29-35\%$, thus clearly demonstrating the portability and applicability of our optimization techniques across different implementation platforms. Please refer Tab.5 for the code-size of our optimized implementation variants. While there is negligible increase in code-size (0.5%) for our Opt-1 variant, we observe an increased overhead of about 17.6% for our Opt-2 variant, that can be mainly attributed due to the *unrolling* optimizations.

4.4 Memory Requirements for Scenario-2 and Scenario-3

Though we observe increased speed-ups for Scenario-2 and Scenario-3, it does come at the cost of requiring to precompute and store certain intermediate values, which consequently requires allocation of additional memory for storage. Hence, we analyze the memory requirements in both scenarios for all parameter sets of Dilithium. Considering Scenario-2, it is required to buffer the modules $\mathbf{A}$, $\mathsf{NTT}(\mathbf{s}_1)$, $\mathsf{NTT}(\mathbf{s}_2)$ and $\mathsf{NTT}(\mathbf{t}_0)$. All the coefficients of these modules occupy 23 bits and hence there are two possible ways to store them. We can either completely use 32

Table 5: Comparison of code-size of the different implementation variants of Dilithium. The size of actual code, constant data and the global variables are separately tabulated as **text**, **data** and **bss** respectively. All the numbers are reported in **bytes**.

Variant	text	data	bss	Total	Overhead (%)
Ref	29696	12	8	29716	**-**
Opt-1	29864	12	8	29884	**0.56**
Opt-2	34912	12	8	34932	**17.6**

bits (4 bytes) to store each coefficient (wasting 9 bits for each) or we can efficiently use a compact bit-packing strategy to efficiently store the same intermediates. Readers are referred to section 5.2 of [8] for the description of the bit-packing strategy used in Dilithium's reference implementation.

In case of Scenario-3, calculation of the memory requirement is a bit more involved, as it is required to additionally pre-compute and store the ephemeral nonce $\mathbf{y}, \mathbf{w}_0$ and $\mathbf{w}_1$ for every iteration. Since the number of iterations required to generate a signature is not known a priori, we perform an analysis of the number of repetitions observed over 10^7 runs of the signing procedure. Refer Fig. 3 for the cumulative distribution plot of the percentage of signatures passed against the minimum number of iterations to be pre-computed, for all parameter sets of Dilithium[3]. We empirically calculated the minimum number of iterations to be pre-computed so as to pass signatures according to three different success rates: 90%, 95% and 99%. Refer Tab.6 for these empirically calculated minimum iteration counts for the aforementioned success rates. Based on these numbers, we also calculated the additional memory requirements for storage of $\mathbf{y}, \mathbf{w}_0$ and $\mathbf{w}_1$ required for implementations in Scenario-3.

Refer Tab.6 for the total memory requirements for implementations in Scenario-2 and Scenario-3 for varying success rates across all parameter sets of Dilithium. We present the memory requirement results for both the packed and unpacked cases. As expected, memory requirements for the packed intermediates are much lesser compared to the unpacked intermediates. But, this comes at the expense of additional performance overhead of unpacking all the stored intermediates.

It is natural to see that the memory requirements increase with increasingly secure parameter sets (i.e) from Dilithium1 to Dilithium4 due to the increase in the module's dimensions. We can clearly see that the memory requirements for Scenario-2 are much lower (14-47 KB for the packed case and 10-34 KB for the unpacked case) compared to Scenario-3 with much higher memory requirements numbering in the hundreds of KBs. The main reason being that the memory requirements

[3] By precomputed iterations, we do not mean computation of the complete iterations, but only computation of $\mathbf{y}, \mathbf{w}_0$ and $\mathbf{w}_1$ corresponding to those iterations.

[4] The reported numbers remain the same irrespective of the utilized XOF function (AES or SHA-3).

Table 6: Memory requirements for implementations in Scenario-2 and Scenario-3 for all parameter sets of Dilithium. Both the packed and un-packed cases are considered. Memory requirements are reported in **Kilobytes**. Please note that Scenario-2 and Scenario-3 are abbreviated as Scen-2 and Scen-3 respectively.

Scheme[4]	**Minimum no. of iterations**			**No Packing (KB)**				**Packing (KB)**			
	90%	95%	99%	**Scen-2**	**Scen-3**			**Scen-2**	**Scen-3**		
					90%	95%	99%		90%	95%	99%
Dilithium1	9	12	18	14	86	110	158	10.1	35.9	44.6	61.8
Dilithium2	13	16	25	23	166	199	309	16.3	68.9	81.0	117.3
Dilithium3	15	19	29	34	244	300	440	24.4	102.3	123.0	174.9
Dilithium4	9	12	18	47	200	251	353	33.8	90.8	109.9	148.0

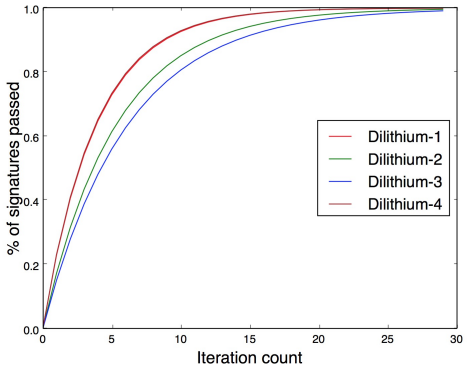

Fig. 3: Cumulative distribution plot of the percentage of signatures passed against the minimum number of iterations to be pre-computed. Please note that the curves for Dilithium1 and Dilithium4 are overlapping one-another.

for Scenario-2 only depend on the module dimensions, but memory requirements for Scenario-3 mainly depend on the repetition rate of the parameter set. This is also evident from the Tab.6 that Dilithium-4 with higher module dimensions $(k, \ell = 6, 5)$ but with a lower average repetition rate of 4.3 has reduced memory requirements in Scenario-3 compared to Dilithium-3 $(k, \ell = 5, 4)$ with a higher average repetition rate of 6.6.

5 Conclusion

In this paper, we have presented an algorithmic optimization on Dilithium's signing procedure which reduces the computations done in the rejected iterations through early-evaluation of the conditional checks. We also incorporate a couple of standard

optimization techniques such as inlining and unrolling to further improve upon the speed of the signing procedure. We also evaluate our optimizations in three different scenarios based on the possibility of performing pre-computations. We perform detailed evaluation of the performance of our optimizations and the memory requirements in the afore mentioned scenarios on the Intel Core i5-4460 CPU and the ARM Cortex-M4F MCU and reported speed-ups in the range of *6%* upto *35%*, thus demonstrating the effectiveness of our proposed optimizations.

Acknowledgment

The authors acknowledge the support from the Singapore National Research Foundation ("SOCure" grant NRF2018NCR-NCR002-0001 – `www.green-ic.org/socure`). This work is also partially supported by NRF TUM CREATE grant.

References

1. Aysu, A., Yuce, B., Schaumont, P.: The future of real-time security: Latency-optimized lattice-based digital signatures. ACM Transactions on Embedded Computing Systems (TECS) 14(3), 43 (2015)
2. Bai, S., Galbraith, S.D.: An Improved Compression Technique for Signatures Based on Learning with Errors. In: CT-RSA. vol. 8366, pp. 28–47 (2014)
3. Barends, R., Kelly, J., Megrant, A., Veitia, A., Sank, D., Jeffrey, E., White, T.C., Mutus, J., Fowler, A.G., Campbell, B., et al.: Superconducting quantum circuits at the surface code threshold for fault tolerance. Nature 508(7497), 500–503 (2014)
4. Barker, E., Barker, W., Burr, W., Polk, W., Smid, M.: Recommendation for key management part 1: General (revision 3). NIST special publication 800(57), 1–147 (2012)
5. Güneysu, T., Lyubashevsky, V., Pöppelmann, T.: Practical lattice-based cryptography: A signature scheme for embedded systems. In: International Conference on Cryptographic Hardware and Embedded Systems. pp. 530–547. Springer (2012)
6. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, `https://github.com/mupq/pqm4`
7. Lyubashevsky, V.: Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 598–616. Springer (2009)
8. Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehle, D.: CRYSTALS-Dilithium. Tech. rep., National Institute of Standards and Technology (2017), available at `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions`
9. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. J. ACM 60(6), 43 (2013)
10. Micciancio, D.: Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. computational complexity 16(4), 365–411 (2007)
11. NIST: Post-Quantum Crypto Project. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/` (2016)
12. Pöppelmann, T., Ducas, L., Güneysu, T.: Enhanced lattice-based signatures on reconfigurable hardware. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 353–370. Springer (2014)
13. Shor, P.W.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on. pp. 124–134. IEEE (1994)