Improving Speed of Dilithium's Signing Procedure

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

Abstract. In this short note, we propose an optimization to improve the signing speed of Dilithium's signing procedure. Our optimization works by reducing the number of computations in the rejected iterations through *Early-Evaluation* of the rejection condition thereby performing optimal number of computations to detect the rejection condition. We would like to note that this straightforward algorithmic optimization only reduces the computational overhead in every rejected iteration, without having any effect on the rejection rate. The *Early-Eval* optimization is also implementation agnostic and hence can be easily ported to all implementation platforms. We also identify possibilities of incorporating standard optimization techniques like inlining and unrolling to to further increase the speed of the signing procedure. We perform experimental validation of our optimization through software implementation on an Intel(R) Core(TM) i5-4460 CPU and observe a speed-up in the range of (5%-31%) across all the parameter-sets of Dilithium. We perform an improved evaluation of the performance of our optimized signing procedure in a range of scenarios based on factors such as the capability of the implementation-platform and the validity of the public-private key pair.

1 Introduction

NIST has called for proposals for standardization of post-quantum cryptographic schemes for public-key encryption, digital signatures, and key establishment protocols [10]. This initiative is partly driven by the onset of the era of practical and scalable quantum computers [11,5,3], which has motivated the cryptographic community to develop cryptographic schemes that are immune to cryptanalytic efforts using quantum algorithms. A total sum of 69 valid submissions (20 digital signature schemes and 49 Public key encryption/Key Establishment schemes) from various different types of post quantum cryptography were submitted for the first round of the standardization process. For the first round evaluation process, NIST identified three broad aspects of evaluation criteria for comparison of the

submitted candidates. They are Security, Cost & Performance and Algorithm and Implementation Characteristics.

After intense scrutiny by NIST and based on public feedback, NIST selected 26 algorithms for the second round of the standardization process. The Dilithium lattice-based signature scheme, part of the CRYSTALS (Cryptographic Suite for Algebraic Lattices) package based on the "Fiat-Shamir with Aborts" framework is also one of the second-round candidates. The security of Dilithium is based on the Module-Learning With Errors (MLWE) problem and offers good security and efficiency guarantees since most of the computations involves polynomials in a cyclotomic ring.

One of the main features of the signing procedure of Dilithium is the use of *rejection sampling* to generate secure signatures that do not leak the distribution of the secret key. The signing procedure loops over multiple iterations until it generates a signature that satisfies certain conditions. Let's say, for a given secret key and message input, the signing algorithm runs for L iterations $(0, \ldots, L-1)$, the computations performed in all except the last iterations are un-necessary overheads, since they are rejected by the signing procedure. While the computations involved in Dilithium are straightforward, the overhead due to computations in the rejected iterations hamper the performance of Dilithium's signing procedure.

In this small note, we would like to propose an optimization involving a straightforward early evaluation of the *rejection* condition so as to reduce the computational overhead in the rejected iterations. We would like to note that this straightforward algorithmic optimization only reduces the computational overhead in every rejected iteration, without having any effect on the rejection rate. We also identify possibilities to further improve the signing speed of Dilithium through incorporation of standard optimization techniques like unrolling and inlining. We also put up a case for refined evaluation of the signing procedure in different scenarios based on a range of factors such as capability of the implementation-platform and the validity of the public-private key pair. We evaluate the performance of our optimized implementation of the signing procedure in a couple of such identified scenarios.

2 Preliminaries

Notation: Let $q \in \mathbb{N}$ be a prime. Elements in ring $\mathbb{Z}$ or $\mathbb{Z}_q$ are denoted by regular font letters viz. $a, b \in \mathbb{Z}$ or $\mathbb{Z}_q$. For an integer r and an even positive integer α, we define centered reduction modulo q denoted as $r \pmod^{\pm} \alpha$, to be the unique integer r_0 such that, $r \equiv r_0 \pmod{\alpha}$ and $-\frac{\alpha}{2} < r_0 \leq \frac{\alpha}{2}$. The usual modulo reduction is denoted by $r \pmod{q}$. For a set X, we write $x \xleftarrow{\$} X$ to denote that x is chosen uniformly at random from X. We denote the polynomial ring $\mathbb{Z}_q[X]/\langle X^n + 1\rangle$ as R_q. Polynomials in ring R_q are also represented as equivalent vectors of length n such that $\mathbf{a} \equiv (\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{n-1})$ for $\mathbf{a}_i \in \mathbb{Z}_q$. For an element $\mathbf{a} \in R_q$, we define $\|\mathbf{a}\|_\infty = \max_{0 \leq i \leq n-1} \|a_i\|_\infty$, where $\|a_i\|_\infty = |a_i \pmod^{\pm} q|$. While

matrices and vectors with elements in $\mathbb{Z}_q$ are denoted by bold upper case letters ($\mathbf{A} \in \mathbb{Z}_q^n$), polynomials in R_q or matrices and vectors with elements in R_q are denoted using bold lower case letters ($\mathbf{a} \in R_q$, $\mathbf{b} \in R_q^\ell$). Multiplication of two polynomials $\mathbf{a}, \mathbf{b} \in R_q$ is denoted as $\mathbf{a} \cdot \mathbf{b}$ or $\mathbf{ab} \in R_q$. Due to the special structure (cyclotomic nature) of the factor polynomial used in R_q, multiplication can also be alternatively viewed as a matrix-vector multiplication such that $\mathbf{a} \cdot \mathbf{b} = \mathbf{a} \cdot \mathbf{B} = \mathbf{b} \cdot \mathbf{A}$ wherein the columns of the matrices $\mathbf{A}, \mathbf{B} \in \mathbb{Z}_q^{n \times n}$ are anti-cyclic rotations of $\mathbf{a}, \mathbf{b} \in R_q$ respectively. Point-wise multiplication (scalar product) is represented as $\mathbf{a} * \mathbf{b} \in R_q$. For a given $\eta \in \mathbb{N}$, define $S_\eta = \{\boldsymbol{a} \in R_q \mid \|\boldsymbol{a}\|_\infty \leq \eta\}$. Individual polynomials in a module $\mathbf{a} \in R_q^{k \times l}$ are denoted as $\mathbf{a}_{i,j}$ with $i \in \{0, k-1\}$ and $j \in \{0, \ell - 1\}$.

Lattice-based Cryptography: Most of the efficient lattice-based cryptographic schemes derive their hardness from two average-case hard problems, known as the Ring-Learning With Errors problem (RLWE) [8] and the Ring-Short Integer Solutions problem (RSIS) [9]. Both the problems reduce to worst-case hard problems over structured ideal lattices. Given a public key $(\mathbf{a}, \mathbf{t}) \in (R_q, R_q)$, an RLWE attacker is asked to find two small polynomials $\mathbf{s}_1, \mathbf{s}_2 \in R_q$ with $\mathbf{s}_1, \mathbf{s}_2 \in S_\eta$ such that $\mathbf{t} = \mathbf{a} \cdot \mathbf{s}_1 + \mathbf{s}_2$. Given m uniformly random elements $\mathbf{a}_i \in R_q$, an MSIS attacker is asked to find out a non-zero vector $\mathbf{z}$ with a small norm $\mathbf{z} \in S_\eta^m$ such that $\sum_i^m \mathbf{a}_i \cdot \mathbf{z}_i = 0 \in R_q$.

The more generalized versions of these problems known as Module-LWE (MLWE) and Module-SIS (MSIS) respectively deal with computations over the space $R_q^{k \times \ell} = \mathbb{Z}_q^{k \times \ell}[X]/(X^n + 1)$ for $k, l > 1$ (as opposed to R_q for their ring variants) and also provide better security guarantees compared to their corresponding ring variants. Any change in the security of a scheme (based on either MLWE or MSIS) only requires changes in the value of the module parameters (k, ℓ) while keeping the underlying structure of the ring fixed, thus warranting very minimal changes from a designer's perspective.

2.1 Dilithium

Dilithium is a lattice-based signature scheme whose security is based on MLWE and MSIS problems. There are two variants of the signature scheme, (i.e) deterministic or probabilistic. In particular, security against key-recovery attack under the classical random oracle model is based on the hardness assumption of the MLWE problem; and the security against existential signature forgery is based on the MSIS hardness assumption. The scheme's security against strong signature forgery attack, under the quantum random oracle model, is also discussed by the authors [7].

2.1.1 Description of Dilithium

In the following, we recall the details of the Dilithium signature scheme [7]. The underlying approach of the scheme is based on the "Fiat-Shamir with Aborts" framework [6] while the scheme in itself is an improved variant of 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 $(k \times \ell)$ for different security levels. The key generation, signing and verification algorithms for Dilithium are presented in Alg.1-2. For these individual procedures, please refer [7].

Key Generation: The key generation algorithm, $\mathsf{KeyGen}()$, generates the public constant $\mathbf{a} \in R_q^{k \times \ell}$ by expanding a given seed $\rho \leftarrow \{0,1\}^{256}$ such that $\mathbf{a} = \mathsf{ExpandA}(\rho)$. Next, the secret module $\mathbf{s}_1 \in S_\eta^\ell$ and the error module $\mathbf{s}_2 \in S_\eta^k$ are sampled after which the MLWE instance $\mathbf{t} \in R_q^k$ is computed as $\mathbf{t} = \mathbf{a} \cdot \mathbf{s}_1 + \mathbf{s}_2$. The LWE instance is not directly output as the public key but is decomposed into $\mathbf{t}_0, \mathbf{t}_1$ such that $\mathbf{t}_1 = \mathsf{HB}_q(\mathbf{t}, 2^d)$ and $\mathbf{t}_0 = \mathsf{LB}_q(\mathbf{t}, 2^d)$. Subsequently, $\mathbf{t}_1$ is published as part of the public key while $\mathbf{t}_0$ is kept secret. Subsequently, the published public key is $(\rho, \mathbf{t}_1)$ while the secret key sk is $(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$.

Signing: The signing procedure is iterative in nature with a number of conditional checks and it exits with a valid signature only when all the conditional checks are successfully passed. This is done to primarily ensure that the output signatures do not leak the distribution of the secret key. Moreover, these selective rejections in the signing procedure are also performed to ensure 100% correctness of the signature scheme.

Similar to the DSA and ECDSA signature schemes, the most important component of the signing procedure in case of Dilithium (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. There are two variants of Dilithium (i.e) deterministic and probabilistic. In deterministic Dilithium, $\mathbf{y} \in S_\eta^\ell$ is deterministically generated using the $\mathsf{ExpandMask}$ function which takes as input, the message μ to be signed, the secret key component K and the iteration count (Line 7 of Sign in Alg.1). But, in case of probabilistic Dilithium, $\mathbf{y}$ is generated using the same $\mathsf{ExpandMask}$ function which takes as inputs, ρ' and k where ρ' is selected at random from $\{0,1\}^{384}$ (Line 6 of Sign in Alg.1). Further, the product $\mathbf{w} = \mathbf{a} \cdot \mathbf{y} \in R_q^k$ is computed and decomposed into $\mathbf{w}_1$ and $\mathbf{w}_0$ such that $\mathbf{w} = \mathbf{w}_1 \cdot 2\gamma_2 + \mathbf{w}_0$. The signing procedure requires the verifier to recover the value of $\mathbf{w}_1$ for successful signature verification. To facilitate the same, a hint vector $\mathbf{h} \in R_q^k$ with coefficients in $\{0,1\}$ is also generated and output as part of the signature. Furthermore, a challenge polynomial $\mathbf{c}$ (sparse polynomial with only 60 non-zero coefficients in either ± 1) is also generated by hashing the ephemeral nonce along with the public key information and the message. The product $\mathbf{cs}_1$ is computed which is subsequently masked with the ephemeral nonce $\mathbf{y}$ through addition and the result is output as the primary signature component $\mathbf{z} \in R_q^\ell$ (Line 11 of Sign in Alg.1). The details of the verification procedure of Dilithium are provided for completeness in Alg.2.

Algorithm 1: Dilithium Signature scheme

Procedure KeyGen()
- $\rho, \rho' \leftarrow \{0,1\}^{256}$
- $K \leftarrow \{0,1\}^{256}$
- $N = 0$
- **for** i *from* 0 *to* $\ell - 1$ **do**
  - $\mathbf{s}_1[i] = \mathsf{Sample}(\mathsf{PRF}(\rho', N))$
  - $N := N + 1$
- **end**
- **for** i *from* 0 *to* $k - 1$ **do**
  - $\mathbf{s}_2[i] = \mathsf{Sample}(\mathsf{PRF}(\rho', N))$
  - $N := N + 1$
- **end**
- $\mathbf{a} \sim R_q^{k \times \ell} = \mathsf{ExpandA}(\rho)$
- $\mathbf{t} = \mathbf{a} \cdot \mathbf{s}_1 + \mathbf{s}_2$
- $\mathbf{t}_1 = \mathsf{Power2Round}_q(\mathbf{t}, d)$
- $tr \in \{0,1\}^{384} = \mathsf{CRH}(\rho || \mathbf{t}_1)$
- **return** $pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$

Procedure Sign(sk, M)
- $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
- $\mu = \mathsf{CRH}(\mathsf{tr} \| M)$
- $\kappa = 0, (\mathbf{z}, \mathbf{h}) = \perp$
- $\rho' \in \{0,1\}^{384} := \mathsf{CRH}(K \| \mu)$(or $\rho' \leftarrow \{0,1\}^{384}$ for randomized signing)
- **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  - $\mathbf{y} \in S_{\gamma_1 - 1}^{\ell} := \mathsf{ExpandMask}(\rho' \| \kappa)$
  - $\mathbf{w} = \mathbf{A} \cdot \mathbf{y}$
  - $\mathbf{w}_1 = \mathsf{HB}_q(\mathbf{w}, 2\gamma_2)$
  - $\mathbf{c} \in B_{60} = H(\mu \| \mathbf{w}_1)$
  - $\mathbf{z} = \mathbf{y} + \mathbf{c} \cdot \mathbf{s}_1$
  - $(\mathbf{r}_1, \mathbf{r}_0) := \mathsf{D}_q(\mathbf{w} - \mathbf{c} \cdot \mathbf{s}_2, 2\gamma_2)$
  - **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**
    - $(\mathbf{z}, \mathbf{h}) = \perp$
  - **else**
    - $\mathbf{h} = \mathsf{MH}_q(-\mathbf{c} \cdot \mathbf{t}_0, \mathbf{w} - \mathbf{c} \cdot \mathbf{s}_2 + \mathbf{c} \cdot \mathbf{t}_0, 2\gamma_2)$
    - **if** $\|\mathbf{c} \cdot \mathbf{t}_0\|_\infty \geq \gamma_2$ or $\mathsf{wt}(\mathbf{h}) > \omega$ **then**
      - $(\mathbf{z}, \mathbf{h}) = \perp$
  - **end**
  - $\kappa = \kappa + 1$
- **end**
- **return** $\sigma = (\mathbf{z}, \mathbf{h}, \mathbf{c})$

Algorithm 2: Dilithium Signature scheme

Procedure $\mathsf{Verify}(\mathsf{pk}, M, \sigma = (\mathbf{z}, \mathbf{h}, \mathbf{c}))$
  $\mathbf{a} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
  $\mu = \mathsf{CRH}\left(\mathsf{CRH}(\rho \| \mathbf{t}_1) \| M\right)$
  $\mathbf{w}_1 := \mathsf{UH}_q(\mathbf{h}, \mathbf{a} \cdot \mathbf{z} - \mathbf{c} \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2)$
  if $\mathbf{c} = H(\mu, \mathbf{w}_1)$ and $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$ and $\mathsf{wt}(\mathbf{h}) \leq \omega$ **then**
    return 1
  else
    return 0
  end

3 Early Evaluation Optimization

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

- $\|\mathbf{z}\|_\infty \leq \gamma_1 - \beta$: $\mathsf{Chk\_Norm}(\mathbf{z})$
- $\|\mathbf{r}_0\|_\infty \leq \gamma_2 - \beta$: $\mathsf{Chk\_Norm}(\mathbf{r}_0)$
- $\|\mathbf{ct}_0\|_\infty \leq \gamma_2$: $\mathsf{Chk\_Norm}(\mathbf{ct}_0)$
- $wt(\mathbf{h}) < w$: $\mathsf{Chk\_Norm}(\mathbf{h})$

Based on the specifications of Dilithium, we make the following observations over the afore mentioned rejection checks in the signing procedure:

- The first three checks ($\mathsf{Chk\_Norm}(\mathbf{z})$, $\mathsf{Chk\_Norm}(\mathbf{r}_0)$ and $\mathsf{Chk\_Norm}(\mathbf{ct}_0)$) contribute the maximum number of rejections in the signing procedure and they are all *infinity norm* checks ($\mathsf{Chk\_Norm}$) over modules consisting of multiple polynomials.
- Individual polynomials in the module can be computed in an independent manner. For eg. every polynomial $\mathbf{z}_i$ for $i \in \{0, L-1\}$ in the module $\mathbf{z}$ can be computed independently during the signing procedure and thus the same also applies to the corresponding computation of their $\mathsf{Chk\_Norm}$ conditions.

Infinity norm checks are necessary conditions (i.e) $\mathsf{Chk\_Norm}$ condition of all the individual polynomials $\mathbf{z}_i$ for $i \in \{0, L-1\}$ of $\mathbf{z}$ have to be satisfied to pass the rejection check. Hence, an iteration can be immediately rejected upon detecting a violation in any of the polynomials. In the reference implementation of Dilithium, the whole of $\mathbf{z}$ is evaluated before checking the validity of the $\mathsf{Chk\_Norm}(\mathbf{z})$ condition over the whole of $\mathbf{z}$. Lets assume a case where the first polynomial of $\mathbf{z}_1$ violates the condition $\mathsf{Chk\_Norm}(\mathbf{z})$. Though the violation can be detected only by computing the first polynomial $\mathbf{z}_1$ of $\mathbf{z}$, the reference implementation computes all the polynomials of $\mathbf{z}$ before starting the rejection check. The same also applies for other polynomials $\mathbf{r}_0$ and $\mathbf{ct}_0$ which are also required to pass the rejection checks. This results in performing un-necessary computations in every rejected iteration.

Hence, we alternately propose to compute $\mathbf{z}$ one polynomial at a time and immediately check its corresponding Chk_Norm before proceeding onto the computation of the other subsequent polynomials of $\mathbf{z}$. Considering the same example where $\mathbf{z}_1$ consists of a violation, we can immediately reject the iteration only upon calculating a single polynomial of $\mathbf{z}$ thus avoiding un-necessary computations related to the other polynomials of $\mathbf{z}$. As stated before, the same technique can also be applied to the computation of other polynomials $\mathbf{r}_0$, $\mathbf{ct}_0$ whose infinity norms are also checked for rejection.

Refer to Fig.1 and Fig.2 for the C-Code snippet corresponding to computation of $\mathbf{z}$ in the reference implementation and optimized implementation respectively. Please note that we have also incorporated another micro-optimization wherein we identify chains of multiple point-wise functions (i.e) functions which compute over one polynomial at a time and combine them into a single function. In particular, we identify polyvecl_add, polyvecl_freeze and polyvecl_chknorm (Line 6,7 and 8 of Listing 1) as the chain of point-wise functions and combine them into a single function into a single function poly_add_freeze_chk_norm (Line 5 in Listing 2). Similarly, we identify two other chains of point-wise functions in the reference implementation which we combine together as a single function. A chain of point-wise functions polyvecl_sub, polyvecl_freeze and polyvecl_chknorm, corresponding to computation of $\mathbf{r}_0$ was combined into poly_sub_freeze_chk_norm while another identified chain of point-wise functions polyvecl_csubq and polyvecl_chknorm, corresponding to the computation of $\mathbf{ct_0}$ were combined into a single function poly_csubq_chk_norm. By doing so, we extend the early-evaluation optimization to computations corresponding to individual coefficients.

We henceforth refer to these set of optimizations as the *Early-Eval* optimizations. Employment of the *Early-Eval* optimization enables to perform optimal number of computations to detect rejection, thus reducing the number of computations done in the rejected iterations.

```
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   }
6   polyvecl_add(&z, &y, &z);
7   polyvecl_freeze(&z);
8   if(polyvecl_chknorm(&z, GAMMA1 - BETA))
9   {
10    goto rej;
11  }
```

Fig. 1: C-Code snippet of computation of $\mathbf{z}$ according to the reference implementation

```
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     {
8         goto rej;
9     }
10  }
```

Fig. 2: C-Code snippet of computation of **z** according to our improved implementation with *Early-Eval* optimization

3.0.1 Implementation-level optimizations We could also observe that the reference implementation of Dilithium consisted of too many function calls used across all the three procedures (key-generation, signing and verification) of Dilithium, that could be considered un-necessary. These functions were implemented in separate files and were compiled into separate object files and hence the compiler couldn't *inline* those functions automatically. For example, a separate rounding function namely *freeze* was computed for every coefficient of certain intermediate polynomials during the signing procedure. This was also observed for multiple-other functions that computed over single-coefficients. Hence, we resorted to *inlining* those functions in order to avoid the un-necessary overhead resulting from branching to the functions and back for every coefficient. Though inlining doesn't call for very elegant code, we observed that we could observe speed-ups of about 5%.

We also resorted to another standard implementation level optimization of *unrolling* the loops. We observed that several functions were computed over all the coefficients of the polynomial. By unrolling the loop, we could potentially save some clock cycles by avoiding the overhead of branching to the first line of the iteration for every coefficient. Unrolling the loop, again calls for not very elegant code and also increases the code-size, but since we nevertheless employ the same as our goal is to speed-up the signing procedure. We henceforth refer to the aforementioned implementation-level optimizations together as *Impl-level* optimizations. It is important to note that both these *Impl-level* optimizations result in speed-up of all the three procedures (key-generation, signing and verification) of the Dilithium signature scheme. But, we only resort to evaluating the speed up in the signing procedure of Dilithium, while evaluation of the speed-up of key-generation and the verification procedures are out of the scope of this work.

4 Experimental Results

We implemented our improved signing procedure incorporating both our *Early-Eval* and *Impl-Level* optimization by modifying the updated reference implementation of Dilithium that was submitted as part of the *second round* of the NIST standardization process. The results were obtained for the two versions of Dilithium across all the four proposed parameter sets (i.e) Dilithium_SHAKE which uses SHAKE as an XOF and Dilithium_AES) which uses AES-256 in counter mode as an XOF. We performed about 10^6 runs of the signing procedure on an Intel(R) Core(TM) i5-4460 CPU 3.20GHz and compiled with gcc-4.2.1 without modifying the compiler flags set for the reference implementation. We perform an independent evaluation of both our *Early-Eval* and *Impl-Level* optimization by considering two different variants of implementation of the signing procedure.

- **Ref:** Without any optimization
- **Opt-1:** With only *Early-Eval* optimization
- **Opt-2:** With both *Early-Eval* and *Impl-Level* optimizations.

We evaluate the performance of all the three implementation variants in three different scenarios.

Scenario-1: We assume that all the operations in the signing procedure are done online (i.e) all operations in the signing procedure are performed only when the message to be signed is known. This is typically the case with the original implementation of the signing procedure. We only evaluate the performance speed-up of the deterministic variant of Dilithium as we do not expect similar results in case of the randomized variant as well.

Scenario-2: We observe that certain operations performed over the public parameters, public key and secret key need not be performed every time the signing procedure is called, since we can safely assume that the public-private key pair is not going to be refreshed for every new signature generated. The parameter sets of Dilithium allow to generate arbitrarily many number of signatures for the same secret key without leaking its distribution through the generated signatures. Thus, operations such as unpacking the secret key ($\mathbf{sk}$), expanding the public matrix (A) and NTT transforms over the public and secret key components ($\mathbf{s}_1$,$\mathbf{s}_2$,$\mathbf{t}_0$) can be done only once and their results being used for multiple calls of the signing procedure. In such a case, we neglect the time taken by the above mentioned operations in our evaluation. We evaluate the speed up due to our optimization in this scenario for both the deterministic and randomized variants.

Scenario-3: On analyzing the randomized variant of the signing procedure, we observed that operations such as sampling $\mathbf{y}$ (Line 7 of the Sign procedure in Alg.1) and computation of $\mathbf{w}_1$ (Line 8 and 9 of Sign procedure in Alg.1) can be performed independent of the message to be signed. In such a scenario, it is possible to split the signing procedure into offline and online phases wherein the message independent computations (Line 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 enough coefficients so as to generate a signature for any given message with very high probability. The remaining operations can be performed online upon the knowledge of the message to be signed and hence, in this scenario, we only evaluate the time taken by the online phase of the signing procedure. The same idea was used by Aysu *et al.* in [1], wherein they reported a high-performance implementation of the GLP signature scheme [4] by partitioning the signing procedure into offline and online phases. For evaluation in such a scenario, we only consider the time taken by the online operations. In general, both key-exchange and digital signature schemes spend a significant (40-50%) amount of time in just hashing (SHAKE functions from the SHA-3 family). Hence, having a hardware accelerated Keccak permutation could significantly accelerate operations in the offline phase, in which case, our *Early-Eval* optimizations in the online phase will yield a significant speedup.

Scheme	**Scenario-1 (Deterministic) ($\times 10^6$ clock cycles)**				
	Ref.	**Opt-1**	**Speed up (%)**	**Opt-2**	**Speed up (%)**
Dilithium1	0.904	0.833	**7.8**	0.742	**17.8**
Dilithium2	1.621	1.461	**9.88**	1.281	**20.9**
Dilithium3	2.359	2.153	**8.69**	1.86	**21.1**
Dilithium4	2.183	2.035	**6.77**	1.771	**18.85**
Dilithium1-AES	1.156	1.094	**5.33**	0.999	**13.55**
Dilithium2-AES	2.110	1.973	**6.919**	1.79	**15.15**
Dilithium3-AES	3.175	2.969	**6.498**	2.676	**15.72**
Dilithium4-AES	3.174	2.970	**6.414**	2.677	**15.65**

Table 1: Performance Evaluation of the reference and optimized implementation of the deterministic signing procedure for Scenario-1 across multiple parameter sets of Dilithium. The results are reported in units of **million (10^6) clock cycles**.

Please refer to Table 1-3 for a tabulation of the performance speedups (in terms of number of clock cycles) due to both *Early-Eval* and *Impl-Level* optimizations in each of the three scenarios across different parameter sets of Dilithium. In case of scenario-1 (Tab.1), we can see that our the optimized implementation variant **Opt-1** (with only *Early-Eval* optimization) has a speed improvement in the range of (5.33-9.88%) while the combination of *Early-Eval* and *Impl-Level* optimizations in the optimized variant **Opt-2** yields an increased-speed up in the range of (13.55-20.9%). In case of scenario-2 wherein we ignore the redundant

operations over the static public and private key, we observe an increased speed in the range of (5.21%-10%) for the **Opt-1** variant and a speed-up in the range of (16.66-24.3%) for the **Opt-2** variant.

In case of Scenario-3, we observe a much more considerable speed-up in the range of (16.88%-26.2%) for the **Opt-1** variant and (23.4-31.7%) speed-up for the **Opt-2** variant, considering an online-offline partition of the signing procedure. The increased speed up is observed in Scenario-3 since we only evaluate the time taken by the online phase, thus ignoring the constant overhead due to the offline phase.

Scheme	**Scenario-2 (Deterministic) ($\times 10^6$ clock cycles)**				
	Ref.	**Opt-1**	**Speed up (%)**	**Opt-2**	**Speed up (%)**
Dilithium1	0.778	0.715	**8.08**	0.617	**20.07**
Dilithium2	1.378	1.246	**9.57**	1.069	**22.4**
Dilithium3	2.042	1.838	**10.0**	1.545	**24.3**
Dilithium4	1.731	1.586	**8.38**	1.320	**23.7**
Dilithium1-AES	0.910	0.863	**5.21**	0.758	**16.6**
Dilithium2-AES	1.663	1.526	**8.23**	1.341	**19.3**
Dilithium3-AES	2.460	2.258	**8.17**	1.966	**20.0**
Dilithium4-AES	2.459	2.258	**8.18**	1.966	**20.0**

Table 2: Performance Evaluation of the reference and optimized implementation of the deterministic signing procedure for Scenario-2 across multiple parameters sets of Dilithium. The results are reported in units of **million (10^6) clock cycles**.

We would like to note that the *Early-Eval* optimization only reduces the computations performed in the rejected iterations and hence demonstrates improved speed without any change in the rejection rate. Since the *Early-Eval* optimization is performed at the algorithmic level, it is implementation agnostic and hence can be easily ported to multiple implementation platforms. But, the relative speed up observed over different platforms will be different depending on the architecture of the implementation of the signing procedure and also the implementation platform.

4.0.1 Note on Timing Attacks: Any given iteration of the signing procedure is *rejected immediately* as soon a rejection through the conditional check on the bound of any coefficient is detected. This is also true in the case of the original reference implementation. In both the cases, it enables an adversary with access

to the timing side-channel to derive information about the coefficient which resulted in rejection. But, since the probability of each coefficient violating the bound is independent of the secret key, knowledge of the position of the coefficient that resulted in rejection does not leak any information about the secret key. Thus, to the best of our knowledge, our *Early-Eval* optimization does not bring in any exploitable additional timing vulnerabilities.

Scheme	**Scenario-3 (Randomized)** ($\times 10^6$ **clock cycles**)				
	Ref.	**Opt-1**	**Speed up (%)**	**Opt-2**	**Speed up (%)**
Dilithium1	0.365	0.303	**16.88**	0.280	**23.2**
Dilithium2	0.598	0.457	**23.5**	0.424	**29.1**
Dilithium3	0.812	0.598	**26.2**	0.557	**31.38**
Dilithium4	0.694	0.548	**20.95**	0.505	**27.2**
Dilithium1-AES	0.365	0.303	**17.01**	0.281	**23.04**
Dilithium2-AES	0.589	0.457	**22.4**	0.426	**27.59**
Dilithium3-AES	0.814	0.597	**26.5**	0.557	**31.53**
Dilithium4-AES	0.817	0.600	**26.5**	0.557	**31.7**

Table 3: Performance Evaluation of the reference and optimized implementations of of the randomized signing procedure for Scenario-3 across multiple parameters sets of Dilithium. The results are reported in units of **million** (10^6) **clock cycles**.

5 Conclusion

In this short note, 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 rejection sampling condition. We also incorporate a couple of standard optimization techniques such as inlining and unrolling to further increase the speed of the signing procedure. We perform an improved evaluation of the performance of our optimized signing procedure in a range of scenarios based on factors such as the capability of the implementation-platform and the validity of the public-private key pair. Through experimental evaluation of the performance of our optimizations on an Intel(R) Core(TM) i5-4460 CPU, we observe speed-ups in the range of approximately (5%-31%) across multiple parameter sets of Dilithium.

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. 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)
5. Harty, T., Allcock, D., Ballance, C.J., Guidoni, L., Janacek, H., Linke, N., Stacey, D., Lucas, D.: High-fidelity preparation, gates, memory, and readout of a trapped-ion quantum bit. Physical review letters 113(22), 220501 (2014)
6. 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)
7. 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-1-submissions`
8. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. J. ACM 60(6), 43 (2013)
9. Micciancio, D.: Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. computational complexity 16(4), 365–411 (2007)
10. NIST: Post-Quantum Crypto Project. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/` (2016)
11. Preskill, J.: Reliable quantum computers. In: Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences. vol. 454, pp. 385–410. The Royal Society (1998)