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. 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 perform experimental validation of our optimization through software implementation on an Intel(R) Core(TM) i5-4460 CPU and observe observe a speed up of about 7-8% of Dilithium's signing procedure for recommended parameter sets of Dilithium. Moreover, this optimization is also implementation agnostic and hence can be ported to all implementation platforms.

1 Introduction

NIST has called for proposals for standardization of post-quantum cryptographic schemes for public-key encryption, digital signatures, and key establishment protocols [8]. This initiative is partly driven by the onset of the era of practical and scalable quantum computers [9,3,2], 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.

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) [6] and the Ring-Short Integer Solutions problem (RSIS) [7]. 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 deterministic lattice-based signature scheme whose security is based on MLWE and MSIS problems. 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 [5].

2.1.1 Description of Dilithium In the following, we recall the details of the Dilithium signature scheme [5]. The underlying approach of the scheme is based on the "Fiat-Shamir with Aborts" framework [4] while the scheme in itself is an improved variant of the lattice-based signature scheme proposed by Bai and Galbraith[1]. 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 Algorithm 1. For these individual procedures, please refer [5].

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. In Dilithium, $\mathbf{y} \in S_\eta^\ell$ is deterministically generated using the ExpandMask function which takes as input, the message μ to be signed, the secret key component K and the iteration count (Line 6 of Sign in Algorithm 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 10 of Sign in Figure 1). The details of the verification procedure of Dilithium are provided for completeness in Algorithm 1.

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.

- $\|\mathbf{z}\|_\infty \leq \gamma_1 - \beta$: Chk_Norm($\mathbf{z}$)
- $\|\mathbf{r}_0\|_\infty \leq \gamma_2 - \beta$: Chk_Norm($\mathbf{r}_0$)
- $\|\mathbf{ct}_0\|_\infty \leq \gamma_2$: Chk_Norm($\mathbf{ct}_0$)
- $wt(\mathbf{h}) < w$: 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 (Chk_Norm($\mathbf{z}$), Chk_Norm($\mathbf{r}_0$) and Chk_Norm($\mathbf{ct}_0$)) contribute the maximum number of rejections in the signing procedure and they are all *infinity norm* checks (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 Chk_Norm conditions.

Infinity norm checks are necessary conditions (i.e) 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 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 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 have combined the three consecutive point-wise functions polyvecl_add, polyvecl_freeze and polyvecl_chknorm (Line 6,7 and 8 of Listing 1) into a single function poly_add_freeze_chk_norm (Line 5 in Listing 2) so that we only compute until the coefficient which is rejected and thus immediately abort the iteration to avoid unnecessary computation of all the coefficients of the polynomial. We note the the same approach also applies for the computation of other intermediate components $\mathbf{r}$ and $\mathbf{ct}_0$. We henceforth refer to these set of optimizations as the *Early-Eval* optimizations.

4 Experimental Results

We implemented our improved signing procedure incorporating our *Early-Eval* optimization by modifying the reference implementation of Dilithium that was submitted to the *first round* of the NIST standardization process. We believe that our optimization would yield almost similar speedup if not exactly the same, even with the updated implementation of Dilithium submitted for the second round. The results were obtained for about 10^7 runs of the signing procedure

```
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 $\mathbf{z}$ according to our improved implementation with *Early-Eval* optimization

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.

Our early evaluation optimization yields a speed up of about 7.6% in the number of clock cycles for the recommended parameter settings of Dilithium's signing procedure. This optimization only reduces the computations performed in the rejected iterations and hence demonstrates improved speed without any change in the rejection rate. Since the optimization is performed at the algorithmic level and does not exploit any device specific features, it can be easily ported to multiple implementation platforms.

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. Our optimization

yields a speed up of about 7.6% (in clock cycles) for recommended parameter sets of Dilithium. Our optimization does not exploit any device-specific feature and hence can be readily included in Dilithium's implementations across multiple implementation platforms.

References

1. 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)
2. 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)
3. 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)
4. 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)
5. 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`
6. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. J. ACM 60(6), 43 (2013)
7. Micciancio, D.: Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. computational complexity 16(4), 365–411 (2007)
8. NIST: Post-Quantum Crypto Project. `http://csrc.nist.gov/groups/ST/post-quantum-crypto/` (2016)
9. 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)

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)
- $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
- $\mu = \mathsf{CRH}(\mathsf{tr} \| M)$
- $\kappa = 0, (\mathbf{z}, \mathbf{h}) = \perp$
- **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  - $\mathbf{y} \in S^{\ell}_{\gamma_1 - 1} := \mathsf{ExpandMask}(K \| \mu \| \kappa)$
  - $\mathbf{w} = \mathbf{A} \cdot \mathbf{y}$
  - $\mathbf{w}_1 = \mathsf{HB}_q(\mathbf{w}, 2\gamma_2)$
  - $\mathbf{c} \in B_{60} = H(\mu \| \mathbf{w}_1)$
  - $\mathbf{z} = \mathbf{y} + \mathbf{c} \cdot \mathbf{s}_1$
  - $(\mathbf{r}_1, \mathbf{r}_0) := \mathsf{D}_q(\mathbf{w} - \mathbf{c} \cdot \mathbf{s}_2, 2\gamma_2)$
  - **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** $\sigma = (\mathbf{z}, \mathbf{h}, \mathbf{c})$

Procedure 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**