Efficient Batch Post-quantum Signatures with Crystals Dilithium

Nazlı Deniz Türe[1,2](✉) and Murat Cenk[3]

[1] Middle East Technical University, Ankara, Turkey
denizzzture@gmail.com
[2] FAME CRYPT, Ankara, Turkey
[3] Ripple Labs Inc., Toronto, Canada

Abstract. Digital signatures ensure authenticity and secure communication. They are used to verify the integrity and authenticity of signed documents and are widely utilized in various fields such as information technologies, finance, education, and law. They are crucial in securing servers against cyberattacks and authenticating connections between clients and servers. Performing multiple signature generation simultaneously and efficiently is highlighted as a beneficial approach for many systems. This work focuses on efficient batch signature generation using Crystal Dilithium, NIST's post-quantum digital signature standard. One of the main operations of signature generation using Dilithium is the matrix-vector product with polynomial entries. So, the naive approach to generate m signatures where $m > 1$ is to perform m such multiplications. In this paper, we propose to use efficient matrix multiplications of sizes greater than four to generate m signatures. To this end, a batch algorithm that transforms the polynomial matrix-vector multiplication in Dilithium's structure into polynomial matrix-matrix multiplication is designed. The batch numbers and the sizes of the matrices to be multiplied based on the number of repetitions of Dilithium's signature algorithm are determined. Moreover, many efficient matrix-matrix multiplication algorithms, such as Strassen-like multiplications and commutative matrix multiplications, are analyzed to design the best algorithms that are compatible with the specified dimensions and yield improvements. Various multiplication formulas are derived for different security levels of Dilithium, and improvements up to 27.28%, 32.0%, and 30.31% in the arithmetic complexities are observed at three different security levels, respectively. The proposed batch Dilithium signature algorithm and the efficient multiplication algorithms are also implemented, and 34.22%, 17.40%, and 10.15% improvements on CPU cycle counts for three security levels are obtained.

Keywords: Batch Digital Signature Generation · Multiple Signing · Post-Quantum Cryptography · Commutative Matrix Multiplication · Crystals Dilithium · Digital Signature

This study is supported by TÜBİTAK 2244 - Industrial PhD Program.

1 Introduction

Digital signatures are essential to provide data integrity, authenticity, and security. Digital signatures are widely used in instant messaging applications, financial transactions, education, legal documents, and many other digital documents. Moreover, they are crucial for servers such as TLS (Transport Layer Security) servers. The servers often face cyber attacks. To prevent unauthorized access, using digital signatures for both the server and user to authenticate each other is essential. RSA [12] is used as a digital signing algorithm on many major servers, such as TLS servers. Servers use digital signatures whenever a secure connection between a client and the server is set. So, it can be said that digital signatures are used thousands or even millions of times daily on a busy server. For example, TLS servers can establish multiple connections per second. The processes need to be fast so that the system's flow is not disrupted. For this reason, performing multiple signing operations at once is faster and more advantageous for systems. Benjamin [4] has developed a system based on ECDSA and RSA that provides multiple signings for TLS. Techniques such as ElGamal [9] have implemented multiple signing in previous studies ([1,7,11]). Fiat [10] and Tanwar et al. [19] present a batch algorithm that depends on the RSA system. Pavlovski et al. [14] propose an efficient batch signature generation via binary tree structures.

High-capacity quantum computers can be used to break many of the cryptographic algorithms in use today, as demonstrated in 1994 by P. Shor [16]. Since then, a lot of work has been done, and many developments have been achieved. For this reason, the National Institute of Standards and Technology (NIST) has organized a contest to standardize algorithms that provide security against attacks via quantum computers. As a result of the third round of evaluations [2], CRYSTALS Dilithium [8] is selected as the digital signing standard. Following this process, Dilithium's integrability into TLS 1.3 is demonstrated in [17]. However, research has shown that using post-quantum in the TLS server will cause performance degradation [13].

The Dilithium structure is examined to increase performance, and the operations with the highest arithmetic complexity are identified. The security of Dilithium is based on the Module-Learning with Error (M-LWE) problem. In Dilithium's signing algorithm, if y is a column vector that is computed using the secret key, the column vector w is obtained as $w = A \cdot y$, where A is the matrix that the signer and the verifier can compute. The entries of the column vectors and the matrix are the elements of the selected commutative polynomial ring. The highlight is that in Dilithium, matrix-vector multiplication is the operation with the highest arithmetic complexity and forms the skeleton of the key generation, signing, and verification algorithms.

This work focuses on efficient batch digital signature generation using the post-quantum algorithm Dilithium. Instead of multiplying a matrix and a column vector, it is possible to multiply the matrix A by another matrix whose columns are column vectors formed for each message. In the article [5], the application of the Strassen method to FrodoKEM [3]'s system containing matrix-matrix multiplication is the starting point of the idea of transforming the primary approach of Dilithium from matrix-vector multiplication to matrix-matrix multiplication for multiple signing purposes. In this way, matrix-matrix multiplication algorithms using the least number of multiplications, such as [6,18,20], and [15], in multiple signing can increase efficiency. Note that since the entries are large-size polynomials, reducing the number of multiplications contributes significantly to complexity.

This study proposes a design of a batch signature generation algorithm for Dilithium's various security levels (i.e., Dilithium 2, Dilithium 3, and Dilithium 5). Matrix-vector multiplication in the Dilithium structure is converted to matrix-matrix multiplication using the batch technique. Batch numbers are determined separately for each security level, according to the number of repetitions in the signing and the probability that the signature produced is valid. Suitable matrix-matrix multiplication techniques are chosen for the selected batch numbers (4, 5, and 4 for Batch Dilithium 2, Batch Dilithium 3, and Batch Dilithium 5, respectively). Since the matrix and vector entries are polynomials from the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$, multiplication is much more expensive than addition. Reducing the number of multiplications is a priority to increase efficiency when selecting these algorithms. Therefore, matrix multiplication algorithms are chosen based on the cost metric minimizing multiplications. It is observed that using [15] for Dilithium 2, Dilithium 3, and Dilithium 5 is more functional. Since Batch Dilithium 2 contains the multiplication of two square matrices of size $(2^d \times 2^d)$, it is possible to apply [6,18,20] to it recursively. However, note that multiplication in [15] cannot be used recursively since it requires entries to be commutative, and matrix multiplications are non-commutative. The multiplication formulas are derived from the most efficient matrix-matrix multiplication techniques based on batch numbers, which determine the size of the matrices to be multiplied. Improvements of up to 28.1%, 33.3%, and 31.5% are observed in the arithmetic complexities of multiplication operations at three different security levels, respectively. The batch system is also implemented for Dilithium's three security levels. Speed comparisons are made between the batch and the reference implementations for 20 random messages, and improvements in CPU cycle counts up to 34.22%, 17.40%, and 10.15% are observed for Dilithium 2, Dilithium 3, and Dilithium 5, respectively.

The organization of the remainder of the paper is as follows: In Sect. 2, related notations and functions are described, and the Crystals Dilithium algorithm is introduced, followed by the presentation of the new algorithm compatible with batch processing in Sect. 3. Then, arithmetic complexity analysis and the improvements observed via implementations for Dilithium's different security levels are detailed in Sect. 4. Section 5 summarizes this research's accomplishments.

2 Preliminaries and Notations

Notations and functions that are used in the remaining sections are given in Table 1.

Table 1. Notations, Functions and their definitions

Notation	Definition
$\boldsymbol{A}$	Matrix (Bold Upper Case Letter)
$\boldsymbol{A}[i][j]$	The entry in the i^{th} row and j^{th} column of $\boldsymbol{A}$
$\boldsymbol{v}$	Column vector (Bold Lower Case Letter)
$\boldsymbol{v}[i]$	i^{th} entry of $\boldsymbol{v}$
R	Commutative ring $\mathbb{Z}[X]/(x^n+1)$
R_q	Commutative ring $\mathbb{Z}_q[x]/(x^n+1)$
$R_q^{k\times l}$	$k \times l$ matrices whose entries are from R_q
R_q^k	Column vectors of length k whose entries are from R_q
$\leftarrow S_\eta^l \times S_\eta^k$ and $\leftarrow S_\eta^l$	Uniform sampling
Function	Definition
NTT	Number Theoretic Transform described in [8]
NTT^{-1}	Inverse operation of Number Theoretic Transform
H	Hash Function (SHAKE-256)
$HighBits$ and $LowBits$	Decomposing operations defined in [8]
$\|z\|_\infty$	$\|z \mod {}^{\pm}q\|$ defined in [8]
$\|\|$	Concatenation Operation

2.1 Crystals Dilithium

Crystals Dilithium is an M-LWE (Module-Learning with Errors) based digital signature algorithm selected as a post-quantum signing standard in the NIST's standardization process. The key generation, signing, and verification algorithms of Dilithium are provided in Algorithms 1, 2 and 3, respectively.

Dilithium's key generation mechanism is explained in Algorithm 1. In the first stage, ρ, ρ' and K are generated using the 256-bit long ζ value. The parameter ρ is used to create the public polynomial matrix $\hat{\boldsymbol{A}}$, and ρ' is used to produce the hidden polynomial vectors $\boldsymbol{s_1}$ and $\boldsymbol{s_2}$. In step 5, $\boldsymbol{t}$ is calculated using the generated polynomial matrix and vectors. Finally, the key pair is created through sub and hash functions.

Algorithm 1. Dilithium Key Generation [8]

```
Output: Public key $pk$, Secret key $sk$
1: $\zeta \leftarrow \{0,1\}^{256}$
2: $(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} := H(\zeta)$
3: $\hat{\boldsymbol{A}} \in R_q^{k\times l} := ExpandA(\rho)$    ▷ $\boldsymbol{A}$ is generated in NTT representation as $\hat{\boldsymbol{A}}$
4: $(\boldsymbol{s_1}, \boldsymbol{s_2}) \in S_\eta^l \times S_\eta^k := ExpandS(\rho')$
5: $\boldsymbol{t} := NTT^{-1}(\hat{\boldsymbol{A}} \cdot NTT(\boldsymbol{s_1})) + \boldsymbol{s_2}$
6: $(\boldsymbol{t_1}, \boldsymbol{t_0}) := Power2Round_q(\boldsymbol{t}, d)$
7: $tr \in \{0,1\}^{256} := H(\rho \| \boldsymbol{t_1})$
8: return $pk = (\rho, \boldsymbol{t_1})$, $sk = (\rho, K, tr, \boldsymbol{s_1}, \boldsymbol{s_2}, \boldsymbol{t_0})$
```

Algorithm 2. Dilithium Signing [8]

```
Input: Secret key $sk = (\rho, K, tr, \boldsymbol{s_1}, \boldsymbol{s_2}, \boldsymbol{t_0})$, message $M$
Output: Signature $\sigma$
1: $\hat{\boldsymbol{A}} \in R_q^{k\times l} := ExpandA(\rho)$    ▷ $\boldsymbol{A}$ is generated in NTT representation as $\hat{\boldsymbol{A}}$
2: $\mu \in \{0,1\}^{512} := H(tr \| M)$
3: $\kappa := 0$, $(\boldsymbol{z}, \boldsymbol{h}) = \perp$
4: $\rho' \in \{0,1\}^{512} := H(K \| \mu)$
5: $\hat{\boldsymbol{s}}_1 := NTT(\boldsymbol{s}_1)$
6: $\hat{\boldsymbol{s}}_2 := NTT(\boldsymbol{s}_2)$
7: $\hat{\boldsymbol{t}}_0 := NTT(\boldsymbol{t}_0)$
8: while $(\boldsymbol{z}, \boldsymbol{h}) = \perp$ do
9:     $\boldsymbol{y} \in \tilde{S}_{\gamma_1}^{l} := ExpandMask(\rho', \kappa)$
10:    $\boldsymbol{w} := NTT^{-1}(\hat{\boldsymbol{A}} \cdot NTT(\boldsymbol{y}))$
11:    $\boldsymbol{w_1} := HighBits_q(\boldsymbol{w}, 2\gamma_w)$
12:    $\tilde{c} \in \{0,1\}^{256} := H(\mu \| \boldsymbol{w_1})$
13:    $c \in B_\tau := SampleInBall(\tilde{c})$
14:    $\boldsymbol{z} := \boldsymbol{y} + NTT^{-1}(\hat{c}\hat{\boldsymbol{s}}_1)$    ▷ $\hat{\boldsymbol{c}} = NTT(\boldsymbol{c})$
15:    $\boldsymbol{r_0} := LowBits_q(\boldsymbol{w} - NTT^{-1}(\hat{c} \cdot \hat{\boldsymbol{s}}_2), 2\gamma_2)$
16:    if $\|\boldsymbol{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\boldsymbol{r_0}\|_\infty \geq \gamma_2 - \beta$ then
17:        $(\boldsymbol{z}, \boldsymbol{h}) := \perp$
18:    else
19:        $\boldsymbol{h} := MakeHint_q(-NTT^{-1}(\hat{c} \cdot \hat{\boldsymbol{t}}_0), \boldsymbol{w} - c\boldsymbol{s_2} + NTT^{-1}(\hat{c} \cdot \hat{\boldsymbol{t}}_0), 2\gamma_2)$
20:        if $\|c\boldsymbol{t}_0\|_\infty \geq \gamma_2$ or the # of 1's in $\boldsymbol{h}$ is greater than $\omega$ then
21:            $(\boldsymbol{z}, \boldsymbol{h}) := \perp$
22:    $\kappa := \kappa + l$
23: return $\sigma = (\tilde{c}, \boldsymbol{z}, \boldsymbol{h})$
```

The Dilithium signature algorithm is described in Algorithm 2. The public polynomial matrix $\hat{\boldsymbol{A}}$ is obtained using the ρ generated by the signature process. The official document is signed using the private key and $\hat{\boldsymbol{A}}$ with the assistance of NTT. The signature phase's correctness is checked. This process needs to be repeated if the requirements are not satisfied. In Dilithium's official document [8], the probability that the signature is established correctly is described in detail, and the probability that the entire signature will be correct is calculated as $e^{-256\cdot\beta\cdot k/\gamma_2}$.

In Algorithm 3, the validity of the signature is checked using the public key.

Algorithm 3. Dilithium Verification [8]

```
Input: Public key $pk = (\rho, \boldsymbol{t_1})$, message $M$, signature $\sigma$
Output: Valid or Not
1: $\hat{\boldsymbol{A}} \in R_q^{k\times l} := ExpandA(\rho)$    ▷ $\boldsymbol{A}$ is generated in NTT representation as $\hat{\boldsymbol{A}}$
2: $\mu \in \{0,1\}^{512} := H(H(\rho \| \boldsymbol{t_1}) \| M)$
3: $c := SampleInBall(\tilde{c})$
4: $\boldsymbol{w'_1} := UseHint_q(\boldsymbol{h}, NTT^{-1}(\hat{\boldsymbol{A}} \cdot NTT(\boldsymbol{z}) - NTT(c) \cdot NTT(\boldsymbol{t_1} \cdot 2^d)))$
5: return [$\| \boldsymbol{z} \|_\infty < \gamma_1 - \beta$] and [$\tilde{c} = H(\mu \| \boldsymbol{w'_1})$] and [# of 1's in $\boldsymbol{h}$ is $\leq \omega$]
```

The parameters of Dilithium are presented in Table 2 where (k, l) is the size of the public matrix $\hat{\boldsymbol{A}}_{k\times l}$. The entries of the matrix $\hat{\boldsymbol{A}}$ are the polynomials from the ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$.

Table 2. Parameter Sets for Dilithium [8]

Security Level	Algorithm	n	(k, l)	q
2	Dilithium 2	256	(4,4)	8380417
3	Dilithium 3	256	(6,5)	8380417
5	Dilithium 5	256	(8,7)	8380417

The matrix-vector multiplications in the 5^{th} step of the Algorithm 1, the 10^{th} step of the Algorithm 2, and the 4^{th} step of the Algorithm 3 are the most costly operations for Dilithium since all the elements in the matrices and column vectors are polynomials from the commutative ring $\mathbb{Z}_q[X]/(x^{256}+1)$ where q is in Table 2. To illustrate this, consider an example where $\boldsymbol{A}$ belongs to $R_q^{6\times5}$ and $\boldsymbol{s'}$ belongs to R_q^5. To calculate $\boldsymbol{A}\cdot\boldsymbol{s'}$, 30 polynomial multiplications are needed, with all the polynomials coming from $\mathbb{Z}_q[X]/(x^{256}+1)$. Signing six different messages for six different users would require 180 polynomial multiplications. Signing multiple messages can be performed more efficiently by reducing the number of multiplications. The proposed algorithm, which ensures multiple messages can be signed at once and more efficiently, is explained in the next section.

3 Dilithium Signature Algorithm for Batch Operations

This section explains the proposed batch version of Dilithium's signing algorithm.

3.1 Batch Crystals Dilithium Signing

The batch Dilithium Signature Algorithm, that is based on the Dilithium Signature Algorithm specified in Algorithm 2 and allows signing more than one message at a time, is explained in Algorithm 4. As an example, batch numbers of the proposed algorithm are selected as 4, 5, and 4 for Dilithium 2, 3, and 5, respectively. How batch numbers can be selected is explained in detail in Sect. 3.3.

Algorithm 4. Batch Dilithium Signing for m Different Messages

```
Input: Secret key $sk = (\rho, K, tr, \boldsymbol{s_1}, \boldsymbol{s_2}, \boldsymbol{t_0})$, messages $M_i$, where $i = 0, 1, \ldots, m-1$
Output: Signatures $\sigma_i$, where $i = 0, 1, \ldots, m-1$
1: $\hat{\boldsymbol{A}} \in R_q^{k \times l} := ExpandA(\rho)$        ▷ $\boldsymbol{A}$ is generated in NTT representation as $\hat{\boldsymbol{A}}$
2: for $i = 0, 1, \ldots, m-1$ do
3:     $\mu_i \in \{0,1\}^{512} := H(tr \| M_i)$
4:     $\kappa_i := 0$, $(\boldsymbol{z_i'}, \boldsymbol{h_i'}) = \perp$
5:     $\rho_i' \in \{0,1\}^{512} := H(K \| \mu_i)$
6: $waitList_i = i \quad \forall i = 0, 1, 2, \ldots, m-1$
7: $\hat{\boldsymbol{s}}_1 := NTT(\boldsymbol{s}_1)$
8: $\hat{\boldsymbol{s}}_2 := NTT(\boldsymbol{s}_2)$
9: $\hat{\boldsymbol{t}}_0 := NTT(\boldsymbol{t}_0)$
10: while Length of $waitList \geq p$ do        ▷ Number of batch $p = 4, 5, 4$ for Dilithium 2, 3, 5.
11:     $status_i = 0 \quad \forall i = 0, 1, 2, \ldots, m-1$
12:     for $i = 0, 1, \ldots, p-1$ do
13:         $\boldsymbol{y}_i \in \tilde{S}_{\gamma_1}^l := ExpandMask(\rho'_{waitList_i}, \kappa_{waitList_i})$
14:     $\hat{\boldsymbol{Y}}$ := $(l \times p)$ matrix, whose columns are $NTT(\boldsymbol{y}_i)$'s, where $i = 0, 1, 2, \ldots, p-1$
15:     $\hat{\boldsymbol{W}} := \hat{\boldsymbol{A}} \cdot \hat{\boldsymbol{Y}}$,   $\hat{\boldsymbol{W}}$ : $(k \times p)$ matrix, whose columns are $\hat{\boldsymbol{w}}_i$'s, where $i = 0, 1, 2, \ldots, p-1$
16:     $\boldsymbol{W}$ := $(k \times p)$ matrix, whose columns are $NTT^{-1}(\hat{\boldsymbol{w}}_i)$'s, where $i = 0, 1, 2, \ldots, p-1$
17:     for $i = 0, 1, \ldots, p-1$ do
18:         $\boldsymbol{w_i'} := HighBits_q(\boldsymbol{i}, 2\gamma_2)$
19:         $\tilde{c}_{waitList_i} \in \{0,1\}^{256} := H(\mu_{waitList_i} \| \boldsymbol{w_i'})$
20:         $c_{waitList_i} \in B_\tau := SampleInBall(\tilde{c}_{waitList_i})$
21:         $\hat{c}_{waitList_i} = NTT(c_{waitList_i})$
22:         $\boldsymbol{z_i'} := \boldsymbol{y}_i + NTT^{-1}(\hat{c}_{waitList_i} \cdot \hat{\boldsymbol{s}}_1)$
23:         $\boldsymbol{r}_i := LowBits_q(\boldsymbol{w}_i - NTT^{-1}(\hat{c}_{waitList_i} \cdot \hat{\boldsymbol{s}}_2), 2\gamma_2)$
24:         if $\|\boldsymbol{z_i'}\|_\infty \geq \gamma_1 - \beta$ then
25:             $status_i = status_i + 1$
26:         if $\|\boldsymbol{r}_i\|_\infty \geq \gamma_2 - \beta$ then
27:             $status_i = status_i + 1$
28:         $\boldsymbol{h_i'} := MakeHint_q(-NTT^{-1}(\hat{c}_{waitList_i} \cdot \hat{\boldsymbol{t}}_0), \boldsymbol{w}_i - c_{waitList_i} \cdot \boldsymbol{s_2} + NTT^{-1}(\hat{c}_{waitList_i} \cdot \hat{\boldsymbol{t}}_0), 2\gamma_2)$
29:         if $\|c\boldsymbol{t}_0\|_\infty \geq \gamma_2$ then
30:             $status_i = status_i + 1$
31:         if The # of 1's in $\boldsymbol{h_i'}$ is greater than $\omega$ then
32:             $status_i = status_i + 1$
33:         $\kappa_{waitList_i} = \kappa_{waitList_i} + l$
34:     for $i = 0, 1, \ldots, p-1$ do
35:         if $!status_i$ then
36:             $\boldsymbol{z}_{waitList_i} = \boldsymbol{z_i'}$
37:             $\boldsymbol{h}_{waitList_i} = \boldsymbol{h_i'}$
38:             Delete $waitList_i$ from the list
39: return $\sigma_i = (\tilde{c}_i, \boldsymbol{z}_i, \boldsymbol{h}_i)$ where $i = 0, 1, \ldots, m-1$
```

The inputs of the batch signing algorithm are the user's secret key $sk = (\rho, K, tr, s_1, s_2, t_0)$ and m different messages. These messages are indexed as $M_0, M_1, \ldots, M_{m-1}$. First, the matrix $\hat{\boldsymbol{A}}$ is produced using ρ as in Algorithm 2. Then, μ_i, κ_i, and ρ_i' values corresponding to each message M_i are computed. κ_i values start with 0, and $(\boldsymbol{z_i'}, \boldsymbol{h_i'})$ are initialized to $\perp$. A $waitList = 0, 1, 2, \ldots, m-1$ list representing indices of messages waiting to be signed is defined. The first p indices in the list represent which messages are in the signing phase at the same time. At the end of the while loop, the indices of messages with appropriate signatures are deleted from this list. Just before the signature phase, $\boldsymbol{s_1}$, $\boldsymbol{s_2}$, and $\boldsymbol{t_0}$, which are the secret key elements, are converted to their NTT formats: $\hat{\boldsymbol{s_2}}$, $\hat{\boldsymbol{s_2}}$, and $\hat{\boldsymbol{t_0}}$. Once the preliminary preparations are completed, the signing loop begins. The candidate signatures produced must satisfy certain conditions. The status of these conditions is controlled by $status_i$.

Note that y_i's are computed using the ρ_i''s and κ_i's of the relevant messages and brought to their NTT format. The produced $\hat{\boldsymbol{y}}_i$'s form the columns of the $\hat{\boldsymbol{Y}}$ matrix. The batch operation of the algorithm is carried out in the 15^{th} step. This step, which is matrix-vector multiplication in the original version, is turned into matrix-matrix multiplication. Each column of the matrix $\hat{\boldsymbol{W}}$ obtained by multiplication is converted to its normal form with NTT^{-1}. These are called w_i's and are used for p messages in the signing phase. In Steps 17–33, the operations performed for a single message in the original algorithm are performed for p messages. The candidate signatures generated for each message are checked with "if" statements, and as a result, the $status_i$'s are updated. Between stages 34 and 38, candidate signatures that meet all conditions are determined to be real signatures, and the indices of properly signed messages are deleted from the $waitList$. p messages waiting in the queue in the $waitList$ enter the loop with updated κ_i values (κ_i values are updated in step 33). The loop continues until less than p elements are left in the $waitList$. Messages that cannot be signed with Algorithm 4 are signed individually with the Original Dilithium Signature Algorithm (Algorithm 2). Thus, all messages are signed properly.

Example: Let M_0, M_1, M_2, M_3, M_4, M_5, M_6, M_7 ($m = 8$) messages to be signed with Dilithium 2. The number of batch p is four and $\kappa = [\kappa_0, \kappa_1, \kappa_2, \kappa_3, \kappa_4, \kappa_5, \kappa_6, \kappa_7]$ is initialized to [0,0,0,0,0,0,0,0] as in Algorithm 4, line 4. The signatures are generated in the while loop and checked to determine whether they comply with the required conditions. Let the generated signatures $(\tilde{c}_3, \boldsymbol{z}_3, \boldsymbol{h}_3)$, $(\tilde{c}_1, \boldsymbol{z}_1, \boldsymbol{h}_1)$, $(\tilde{c}_5, \boldsymbol{z}_5, \boldsymbol{h}_5)$, $(\tilde{c}_6, \boldsymbol{z}_6, \boldsymbol{h}_6)$, and $(\tilde{c}_0, \boldsymbol{z}_0, \boldsymbol{h}_0)$ are valid in order. For example, the change of κ and $waitList$ according to the number of loops and valid signatures generated is shown in Table 3.

Table 3. Change of κ and $waitList$ according to the number of loops and valid signatures generated.

# of Loop	Valid	$\kappa = [\kappa_0, \kappa_1, \kappa_2, \kappa_3, \kappa_4, \kappa_5, \kappa_6, \kappa_7]$	$waitList$
0 (start)	–	[0,0,0,0,0,0,0,0]	[0,1,2,3,4,5,6,7]
1	$(\tilde{c}_3, \boldsymbol{z}_3, \boldsymbol{h}_3)$	[4,4,4,4,0,0,0,0]	[0,1,2,4,5,6,7]
2	$(\tilde{c}_1, \boldsymbol{z}_1, \boldsymbol{h}_1)$	[8,8,8,4,4,0,0,0]	[0,2,4,5,6,7]
3	$(\tilde{c}_5, \boldsymbol{z}_5, \boldsymbol{h}_5)$	[12,8,12,4,8,4,0,0]	[0,2,4,6,7]
4	$(\tilde{c}_6, \boldsymbol{z}_6, \boldsymbol{h}_6)$	[16,8,16,4,12,4,4,0]	[0,2,4,7]
5	$(\tilde{c}_0, \boldsymbol{z}_0, \boldsymbol{h}_0)$	[20,8,20,4,16,4,4,4]	[2,4,7]

First, the matrix $\hat{\boldsymbol{A}}$ is generated via Algorithm 4 and μ_i, ρ_i' are computed for $i = 0, 1, 2, \ldots, 7$. Four messages are handled simultaneously since $p = 4$. The four messages that are processed are determined by the first four components of $waitList$. At the beginning, $waitList = [0, 1, 2, 3, 4, 5, 6, 7]$. Since the first four components are $[0, 1, 2, 3]$, the first while loop tries to generate signatures for the messages M_0, M_1, M_2, M_3. According to Table 3, the signature candidates produced at the end of the first loop are $(\tilde{c}_0, \boldsymbol{z}_0, \boldsymbol{h}_0)$, $(\tilde{c}_1, \boldsymbol{z}_1, \boldsymbol{h}_1)$, $(\tilde{c}_2, \boldsymbol{z}_2, \boldsymbol{h}_2)$, $(\tilde{c}_3, \boldsymbol{z}_3, \boldsymbol{h}_3)$. Only one of these four signature candidates satisfies the conditions and is valid, which is $\sigma_3 = (\tilde{c}_3, \boldsymbol{z}_3, \boldsymbol{h}_3)$. The corresponding κ values are updated to [4,4,4,4,0,0,0,0]. Since the signature generated for M_3 is valid, three is removed from the *waitList* and *waitList* $= [0, 1, 2, 4, 5, 6, 7]$. In the second round, signature candidates are produced for M_0, M_1, M_2, M_4. Assume that only $(\tilde{c}_1, \boldsymbol{z}_1, \boldsymbol{h}_1)$ is correct among the four signatures generated at the end of the second round. Then, $\kappa = [8, 8, 8, 4, 4, 0, 0, 0]$ and 1 is deleted from *waitList* and the list becomes *waitList* $= [0, 2, 4, 5, 6, 7]$. This structure continues until the number of elements of *waitList* is less than p. Thus, at the end of the Batch Dilithium 2 Signing algorithm, σ_3, σ_1, σ_5, σ_7, and σ_0 are produced, respectively.

The messages M_2, M_4, and M_7 that cannot be signed with the batch algorithm are signed individually using the original Dilithium 2 Signing Algorithm (Algorithm 2). One step further, it is possible to initialize κ values of the original Dilithium 2 signing algorithm to the values obtained at the end of the batch algorithm since some κ values are tried for these messages, and no correct results are obtained. Finally, all messages are signed by combining Algorithms 4 and 2.

3.2 Making the Batch Algorithm More Efficient Compared to the Naive Approach

The process in step 15 of Algorithm 4 is emphasized to ensure that the batch algorithm is more efficient than the naive one. The matrix-vector multiplication, which is done separately for each message in the original algorithm, has now turned into matrix-matrix multiplication. Due to the structure of Dilithium, the elements of these matrices are not integers but polynomials of degree 255. For this reason, although the matrix sizes are very small, one of the most expensive phases of the algorithm is the multiplication of these matrices. If these multiplications are done using the schoolbook method, the batch version does not seem to have an advantage over the original version. However, many matrix-matrix multiplication algorithms are available in the literature that can be used depending on the dimensions and properties of the matrices. If appropriate algorithms are chosen, the number of polynomial multiplications required by this operation can be reduced, and generating multiple signatures at once is more efficient than generating them one by one.

The Importance of Commutativity Property and Choosing The Proper Efficient Matrix-Matrix Multiplication Algorithms. The dimensions of the matrices to be multiplied in the 15^{th} step of Algorithm 4 according to the determined batch numbers and different security levels of Dilithium signing are shown in Table 4.

Table 4. Batch numbers and matrix sizes according to different security levels.

Security Level	Algorithm	Batch Number	Size of $\hat{A}$	Size of $\hat{Y}$	Size of $\hat{W}$
2	Dilithium 2	4	(4×4)	(4×4)	(4×4)
3	Dilithium 3	5	(6×5)	(5×5)	(6×5)
5	Dilithium 5	4	(8×7)	(7×4)	(8×4)

Considering the matrix dimensions and structure of Dilithium, many matrix-matrix multiplication algorithms can be used. Dilithium's ring, $R = \mathbb{Z}_{8380417}[X]/(X^{256}+1)$, is commutative. For this reason, the multiplication of polynomials, which are the entries of matrices, is commutative. Thanks to this feature, the product of any two entries, a_{ij} and y_{kl} has the equality $a_{ij} \cdot y_{kl} = y_{kl} \cdot a_{ij}$. By taking advantage of this feature, a fast commutative matrix-matrix multiplication algorithm detailed in [15] can be used for all security levels. This algorithm works more efficiently than the Strassen [18] method or the non-commutative methods in the literature. Non-commutative methods can also be preferred when the matrix multiplication is performed recursively for larger-size matrices.

Strassen-like multiplications such as Strassen method, Cenk&Hassan's method [6], and Winograd's Multiplication [20] are known to be the best recursive matrix-matrix multiplications.

In this work, we derive the Batch Dilithium 2 algorithm, $(4\times4)\cdot(4\times4)$ matrix-matrix multiplication formulas using Strassen's Method and the fast commutative method. Matrix multiplication formulas for $(6\times5)\cdot(5\times5)$ and $(8\times7)\cdot(7\times4)$ are obtained for Batch Dilithium 3 and Dilithium 5 using the fast commutative method.

3.3 Probability Computations and Choosing the Batch Sizes

Dilithium's signing algorithm requires specific requirements to be satisfied by signature candidates. The signature algorithm uses if statements to verify these conditions. Lines 24, 26, 29, and 31 in the batch algorithm (Algorithm 4) and 13, 17 in the classical algorithm (Algorithm 2) have these criteria. Step 13 is more dominant in the condition checks in Algorithm 2. Because of this, the following formula-which is defined in the [8]-is used to determine the probability that the signature created is valid (for step 13):

$$\approx e^{-256\cdot\beta(l/\gamma_1+k/\gamma_2)}. \tag{1}$$

The values of the variables included in the formula and probabilities that a generated signature candidate will satisfy the requirements based on the various Dilithium security levels are given in Table 5. Signatures that do not satisfy the conditions are reproduced with the while loop.

Table 5. Variables required to compute the probability

Variable	Dilithium 2	Dilithium 3	Dilithium 5
γ_1	2^{17}	2^{19}	2^{19}
γ_2	95232	261888	261888
(k, l)	(4,4)	(6,5)	(8,7)
β	78	196	120
Probability	≈ 0.24	≈ 0.196	≈ 0.26

Depending on the batch number, the probability of at least one of the p signatures produced by Dilithium 2, Dilithium 3, and Dilithium 5 being valid are calculated as $1-(1-0.24)^p$, $1-(1-0.196)^p$, and $1-(1-0.26)^p$, respectively.

The probability that at least one of the p signatures is valid increases with the batch number. When calculating the batch number p, two scenarios need to be taken into account. The first is the probability that when p number of messages are signed, at least one of these signatures is correct. The second is the rate of improvement that the chosen p will provide. Based on these two scenarios, the user may determine the priorities and select an appropriate batch number. Moreover, while choosing the batch number, the formulas to be derived based on the matrix sizes and the possible matrix-matrix multiplication algorithms should also be considered.

In Table 6, the probabilities that at least one of the p signatures is valid based on some eligible batch numbers (p) are given for Dilithium 2, 3, and 5.

Table 6. Probabilities of at least one of p signatures being correct according to batch number (p) for Dilithium 2, 3, and 5.

	Probability		
p (#of batch)	Dilithium 2	Dilithium 3	Dilithium 5
3	0.561	0.480	0.595
4	0.666	0.582	0.700
5	0.746	0.664	0.778
6	0.807	0.730	0.836
7	0.854	0.783	0.878
8	0.889	0.825	0.910
9	0.915	0.860	0.933
10	0.936	0.887	0.951
11	0.951	0.909	0.964
12	0.963	0.927	0.973
13	0.972	0.941	0.980
14	0.979	0.953	0.985
15	0.984	0.962	0.989
16	0.988	0.970	0.992
17	0.991	0.975	0.994
18	0.993	0.980	0.996
19	0.995	0.984	0.997
20	0.996	0.987	0.998

Figures 1, 2, and 3 show how the probability of at least one of p signatures being correct according to the batch number p for Dilithium 2, 3, and 5, respectively. Table 6 and Figs. 1, 2, and 3 show that the probability converges to 1 as the number of batches increases.

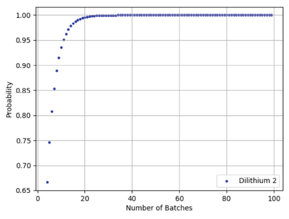

Fig. 1. Change in probability of at least one of p signatures generated with Dilithium 2 being correct according to batch number (p).

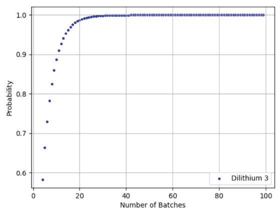

Fig. 2. Change in probability of at least one of p signatures generated with Dilithium 3 being correct according to batch number (p).

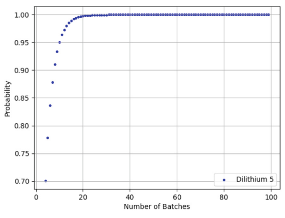

Fig. 3. Change in probability of at least one of p signatures generated with Dilithium 5 being correct according to batch number (p).

The number of polynomial multiplication operations that need to be performed according to the selected batch number is given in Table 7. For the batch method, [15] is used as the efficient matrix-matrix multiplication method.

In Dilithium 2, via [15] the number of multiplication operations required to multiply matrices $\hat{\boldsymbol{A}}^{k\times l}$ and $\hat{\boldsymbol{Y}}^{l\times p}$ can be computed as $l(kp+k+p-1)/2$ where $k=4$ and $l=4$. Similarly, in Dilithium 3 and 5, the number of multiplication operations required can be computed as $(l(kp+k+p-1)+k-1)/2$ for p is even and $l(kp+k+p-1)/2$ for p is odd, where $k=6$, $l=5$ for Dilithium 3, and $k=8$, $l=7$ for Dilithium 5.

Table 7. Required number of multiplications for a single signature assuming one of the p signatures is correct

	Dilithium 2		Dilithium 3		Dilithium 5	
p (# of Batch)	Batch	Classical	Batch	Classical	Batch	Classical
3	36	48	65	90	119	168
4	46	64	85	120	154	224
5	56	80	100	150	182	280
6	66	96	120	180	217	336
7	76	112	135	210	245	392
8	86	128	155	240	280	448
9	96	144	170	270	308	504
10	106	160	190	300	343	560
11	116	176	205	330	371	616
12	126	192	225	360	406	672
13	136	208	240	390	434	728
14	146	224	260	420	469	784
15	156	240	275	450	497	840
16	166	256	295	480	532	896
17	176	272	310	510	560	952
18	186	288	330	540	595	1008
19	196	304	345	570	623	1064
20	206	320	365	600	658	1120

Table 8. Improvement Rates (%) for 20 Signatures

	Dilithium 2	Dilithium 3	Dilithium 5
p (#of Batch)	Impr (%)	Impr (%)	Impr (%)
3	22.50	25.00	26.25
4	23.91	24.79	26.56
5	24.00	26.67	28.00
6	23.44	25.00	26.56
7	22.50	25.00	26.25
8	21.33	23.02	24.38
9	20.00	22.22	23.33
10	18.56	20.17	21.31
11	17.05	18.94	19.89
12	15.47	16.88	17.81
13	13.85	15.38	16.15
14	12.19	13.33	14.06
15	10.50	11.67	12.25
16	8.79	9.64	10.16
17	7.06	7.84	8.24
18	5.31	5.83	6.15
19	3.55	3.95	4.14
20	1.78	1.96	2.06

For example, let 20 messages be signed with Batch Dilithium. The improvement rates that will be obtained by reducing the number of multiplications are given in Table 8. As can be observed from the table, the best improvement rate is obtained by selecting p as 5. According to improvement percentages and probability calculations, p can be selected by taking into account the available efficient matrix-matrix multiplication algorithms. Let C be the number of multiplications required for the classical method, B be the number of multiplications required for the efficient matrix-matrix multiplication algorithm used in the batch method, m be the number of messages to be signed, and p be the number of batches. The improvement rate is calculated with the following formula:

$$(Cm-(B(m-(p-1))+C(p-1)))100/(Cm) \tag{2}$$

As observed in Table 5, the probabilities of signatures being valid for Dilithium 2, 3, and 5 are approximately 1/4, 1/5, and 1/4, respectively. So, one of the four signatures produced by Dilithium 2, one of the five for Dilithium 3, and one of the four for Dilithium 5 are expected to be valid with a high probability. Therefore, the probability that at least one of the four signatures produced by Dilithium 2 is valid is $1 - (1 - 0.24)^4 \approx 0.67 = 67\%$. Similarly, the probability that at least one of the five signatures produced by Dilithium 3 is valid is 66%, and the probability that at least one of the four signatures produced by Dilithium 5 is valid is 70%. As an example, batch numbers are selected as 4, 5, and 4 for Dilithium2, 3, and 5, respectively.

4 Results

The results of the batch application are examined under two main headings. These are improvements in the arithmetic complexity and the improvements observed via the implementations.

4.1 Matrix Multiplications and Arithmetic Complexity Analysis

The batch method must be used together with efficient matrix-matrix multiplication algorithms in order to be effective. For this reason, the batch numbers to be used in the Batch Dilithium Algorithm for each security level are determined by the method explained in Sect. 3.3. The dimensions of the matrices in the 15^{th} step of the Algorithm 4 according to each security level of Dilithium are given in Table 4. Since the entries of the matrices are polynomials and the multiplications of those entries are highly costly compared to their additions, the aim is to reduce the number of multiplications. The matrix-matrix multiplication algorithms in the literature with the minimum number of multiplications are generally obtained by using the Strassen-like multiplications recursively. However, since the sizes of the matrices in our work are small, they do not require recursions. So, the commutative matrix multiplication algorithms that have better multiplicative complexity than the non-commutative multiplication algorithms can also be used for our purposes since the entries are polynomials from the commutative ring R_q. Therefore, it is advantageous to use the commutative matrix-matrix multiplication algorithm described in [15].

Let (n_1, n_2, n_3) matrix multiplication be the product of a matrix of dimension $(n_1 \times n_2)$ and a matrix of dimension $(n_2 \times n_3)$. We need an efficient $(4, 4, 4)$ matrix multiplication for Dilithium 2, $(6, 5, 5)$ matrix multiplication for Dilithium 3, and $(8, 7, 4)$ for Dilithium 5. For the $(4, 4, 4)$ multiplications, the best choice seems to be Strassen-like recursive multiplications [6,18,20] with 49 multiplications and the commutative matrix multiplication method by [15] with 46 multiplications. For $(6, 5, 5)$ matrix multiplication in batch Dilithium 3 can be performed with 100 multiplications, and $(8, 7, 4)$ matrix multiplication in batch Dilithium 5 can be performed with 154 multiplications using [15]. We derive all explicit formulas and use them in our implementation.

Batch Dilithium 2. Assume that m messages are signed independently with Dilithium 2 using the classical approach. Each message needs 16 multiplications. However, due to the failure rate in the algorithm, $16 \cdot 4 = 64$ multiplications are expected to sign the message on average. Therefore, a total of $64m$ multiplications are needed for m messages.

In the batch method in Algorithm 4, step 15 is performed for four signatures. Since the probability that a signature candidate is valid for Dilithium 2 is approximately 0.24, one of the four signatures can be assumed to be valid. This process ends when $m - (p-1)$ messages are signed. Thus, the batch operation requires a total of $46 \cdot (m - (p-1)) = 46 \cdot (m-3)$ multiplication operations. The remaining $p - 1 = 3$ messages from the batch process are signed individually with classical Dilithium 2. Step 7 in the Algorithm 2 requires 16 multiplications to sign each message. If we consider that the loop is repeated four times to generate a valid signature, the number of multiplications needed for a message is calculated as $16 \cdot 4 = 64$. Thus, the number of multiplications required for three messages is $3 \cdot 64 = 192$, and the number of multiplications required for batch Dilithium 2 is approximately $46 \cdot (m-3) + 192 = 46\,m + 54$. The number of multiplications required for the classical and batch methods, together with the improvement rates obtained with batch Dilithium 2, is presented in Table 9 for the selected number of messages up to 100. The improvement rate is calculated as $(64\,m - (46\,m + 54)) \cdot 100/64\,m$, and as the number of messages increases, this rate converges to 28.1%.

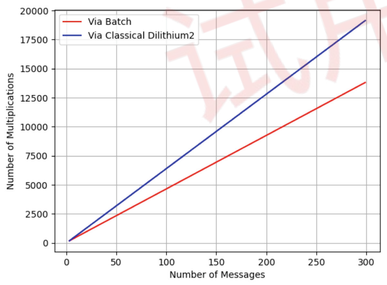

Fig. 4. Variation of the number of multiplications required according to the number of messages for Batch Dilithium 2 and Classical Dilithium 2

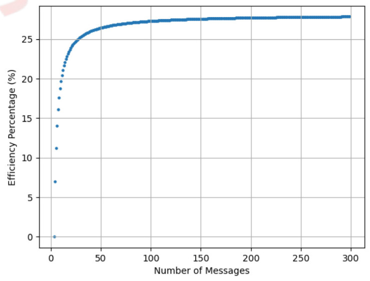

Fig. 5. Variation of the improvement rate (%) provided by the batch algorithm compared to the classical version depending on the number of messages for Dilithium 2

Variation of the number of multiplications required according to the number of messages in Batch Dilithium 2 and Classical Dilithium 2 is shown in Fig. 4. Variation of the improvement rate (%) provided by the batch algorithm compared to the classical version depending on the number of messages can be observed in Fig. 5.

Table 9. Variation of the number of multiplications required for batch and classical use of Dilithium 2, and the improvement rates provided by batch method with [15] according to the number of messages

# of Messages	Batch ($p = 4$)	Classic	Improvement (%)
m	$46\,m + 54$	$64\,m$	
4	238	256	7.0
5	284	320	11.25
6	330	384	14.1
7	376	448	16.1
8	422	512	17.58
9	468	576	18.85
10	514	640	19.69
20	974	1280	23.91
40	1894	2560	26.02
80	3734	5120	27.07
100	4654	6400	27.28

Batch Dilithium 3. If the classical method is used to sign m messages using Dilithium 3, it takes 30 multiplications for a message. But, because of the failure rate, $30 \cdot 5 = 150$ multiplications are done on average. For m messages, a total of $150m$ multiplications are expected.

The batch signature generation is performed for five signatures in Algorithm 4 at step 15. It is presumed that at least one of the five signatures is valid since the probability that a signature candidate is valid for Dilithium 3 is approximately 0.196. As a result of calculations similar to those in batch Dilithium 2, the batch operation requires a total of $100 \cdot (m - 4)$ multiplication operations. When the remaining four messages are signed using the classical method, the total number of multiplications required for batch Dilithium 3 is $100m + 200$. According to the number of messages, the number of multiplications required for the classical or batch method, and the improvement rates obtained with batch Dilithium 3 are provided in Table 10. The improvement rate is computed using $(150m - (100m + 200)) \cdot 100/150\,m$. As the number of messages increases, this rate converges to 33.3%. Variation of the number of multiplications required according to the number of messages in Batch Dilithium 3 and Classical Dilithium 3 is shown in Fig. 6. Variation of the improvement rate (%) provided by the batch algorithm compared to the classical version depending on the number of messages can be observed in Fig. 7.

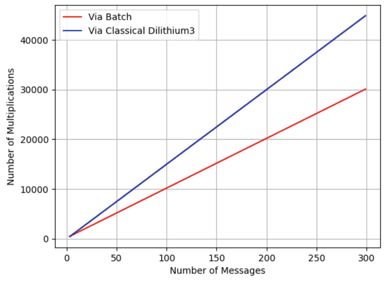

Fig. 6. Variation of the number of multiplications required according to the number of messages for Batch Dilithium 3 and Classical Dilithium 3

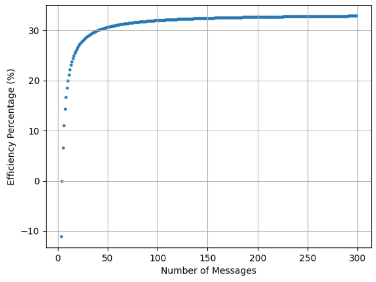

Fig. 7. Variation of the improvement rate (%) provided by the batch algorithm compared to the classical version depending on the number of messages for Dilithium 3

Table 10. Variation of the number of multiplications required for batch and classical use of Dilithium 3, and the improvement rates provided by batch method with [15] according to the number of messages

# of Messages	Batch ($p = 5$)	Classic	Improvement (%)
m	$100\,m + 200$	$150\,m$	
5	700	750	6.7
6	800	900	11.11
7	900	1050	14.29
8	1000	1200	16.67
9	1100	1350	18.52
10	1200	1500	20.0
20	2200	3000	26.67
40	4200	6000	30.0
80	8200	12000	31.67
100	10200	15000	32.0

Batch Dilithium 5. Assume the classical method is used to sign m messages separately using Dilithium 5. It takes 56 multiplications for each message. Performing $56 \cdot 4 = 224$ multiplications is done under the assumption that a valid signature requires four repetitions. For m messages, a total of $224m$ multiplications must be done.

Similar to Batch Dilithium 2, 15^{th} step of the Algorithm 4 is performed for four signatures. Since the probability that a signature candidate is valid for Dilithium 5 is approximately 0.26, it is assumed that at least one of the four signatures will be valid. As a result of calculations similar to those in batch Dilithium 2 and 3, the batch operation requires a total of $154 \cdot (m - 3)$ multiplication operations. When the remaining three messages are signed using the classical method, the total number of multiplications required for batch Dilithium

5 is $154m + 210$. According to the number of messages, the number of multiplications required for the classical or batch method, and the improvement rates obtained with batch Dilithium 5 are presented in Table 11.

Table 11. Variation of the number of multiplications required for batch and classical use of Dilithium 5, and the improvement rates provided by batch method with [15] according to the number of messages

# of Messages	Batch ($p = 4$)	Classic	Improvement (%)
m	$154\,m + 210$	$224\,m$	
4	826	896	7.81
5	980	1120	12.5
6	1134	1344	15.63
7	1288	1568	17.86
8	1442	1792	19.53
9	1596	2016	20.83
10	1750	2240	21.86
20	3290	4480	26.56
40	6370	8960	28.91
80	12530	17920	30.08
100	15610	22400	30.31

The improvement rate is computed using $(224m - (154m + 210)) \cdot 100/224m$. As the number of messages increases, this rate converges to 31.5%. Variation of the number of multiplications required according to the number of messages in Batch Dilithium 5 and Classical Dilithium 5 is shown in Fig. 8. Variation of the improvement rate (%) provided by the batch algorithm compared to the classical version depending on the number of messages can be observed in Fig. 9.

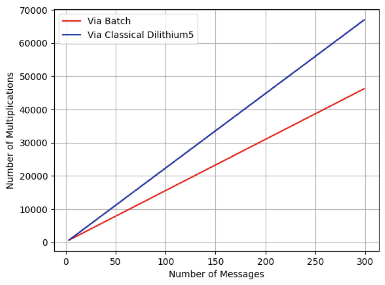

Fig. 8. Variation of the number of multiplications required according to the number of messages for Batch Dilithium 5 and Classical Dilithium 5

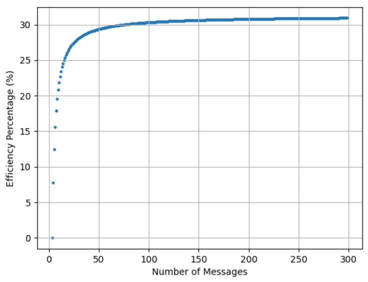

Fig. 9. Variation of the improvement rate (%) provided by the batch algorithm compared to the classical version depending on the number of messages for Dilithium 5

4.2 The Improvements Observed via the Implementations

Twenty random messages are generated to compare the batch implementations and the reference. Each message is signed individually using the reference implementation for CPU cycle counting. After that, the same messages are signed via the batch technique, and similar speed tests are performed. In this implementation, cycle counts are obtained utilizing a single core of the Intel Core i7-8700 processor.

Table 12 gives the cycle counts of only the multiplication process of signing 20 random messages with Classical Dilithium and Batch Dilithium. Using Batch Dilithium, 17 messages are signed, and CPU cycles are counted using the batch process. For the remaining three messages, classical Dilithium is used. The same 20 messages were also signed with Classical Dilithium. In order to make an accurate comparison, the cycle counts of signing 17 messages for Classical Dilithium are measured.

Table 12. CPU Cycle counts of the multiplication stage obtained by signing 17 random messages ($m = 20 - 3$) with reference and batch implementations of Dilithium 2, Dilithium 3, and Dilithium 5. Cycle counts are obtained on one core of Intel Core i7-8700.

Dilithium Signature Generation (Only the Multiplication Stage)				
Algorithm	Batch Size	Reference	Batch	Improvement (%)
Dilithium 2	4	2555202	1532297 (via [15])	40.03
Dilithium 2	4	2555202	1809040 (via [18])	29.20
Dilithium 3	5	4723348	3331311 (via [15])	29.47
Dilithium 5	4	5715468	4914677 (via [15])	14.01

Table 13 shows the tests' results for Dilithium's three security levels. Dilithium 2 is implemented using Strassen's algorithm and Rosowski's commutative algorithm. Since the latter has slightly better multiplicative complexity, it yields faster results. Dilithium 3 and Dilithium 5 are implemented only using Rosowski's method.

Table 13. CPU Cycle counts obtained by signing 20 random messages ($m = 20$) with reference and batch implementations of Dilithium 2, Dilithium 3, and Dilithium 5. Cycle counts are obtained on one core of Intel Core i7-8700.

Dilithium Signature Generation				
Algorithm	Batch Size	Reference	Batch	Improvement (%)
Dilithium 2	4	31382991	20645253 (via [15])	34.22
Dilithium 2	4	31382991	20991482 (via [18])	33.11
Dilithium 3	5	50128969	41407391 (via [15])	17.40
Dilithium 5	4	46078440	41833217 (via [15])	10.15

The fact that the method we have presented uses matrix-matrix multiplication instead of matrix-vector multiplication means that the entire matrix is represented instead of a single vector. Calculating and keeping the linear combinations contained in the structure of multiplication methods and obtaining entries of the matrix $\hat{\boldsymbol{W}}$ by addition and subtraction operations increases stack memory consumption that can be reduced with appropriate implementation techniques (using loops to calculate linear combinations or parallelization techniques, etc.). Using matrices for the multiplication operations instead of vectors requires dynamic memory allocations for the operations to be appropriately calculated in the algorithm represented by the entire function. Thus, these create a time-memory trade-off, as expected. However, our computations show that it is not significant since the matrix is constructed by adding a few more vectors. It should also be noted that the proposed method can be implemented using SIMD instructions such as AVX2. This approach can significantly enhance performance and could be considered for future work.

5 Conclusion

This study proposes efficient batch signature generation with the Dilithium algorithm. The batch Dilithium signing algorithm is designed to enable many messages to be signed simultaneously. According to the repetition numbers of Dilithium 2, Dilithium 3, and Dilithium 5 signing algorithms, the column sizes of the first matrix (i.e., batch numbers) are determined as 4, 5, and 4, respectively. Dilithium's matrix-vector multiplication has been converted to matrix-matrix multiplication. This transformation allows us to employ efficient matrix-matrix multiplications for many signature generations. The matrix dimensions that provide improvements are determined, and efficient multiplication methods, such as commutative matrix multiplication by Rosowski and Strassen's multiplication algorithms, are integrated into Dilithium. As a result of those multiplications, the arithmetic complexities of generating many signatures are enhanced up to 28.1% for Dilithium 2, 33.3% for Dilithium 3, and 31.5% for Dilithium 5. Moreover, we implement the proposed batch signature generation by signing 20 messages using the efficient matrix-matrix multiplication algorithms for three security levels and obtain improvements in terms of CPU cycle counts, which are 34.22% for Dilithium 2, 17.40% for Dilithium 3, and 10.15% for Dilithium 5.

Acknowledgments. We would like to extend our heartfelt gratitude to Sevim Seda Odacıoğlu for her invaluable assistance in the implementation phase of this paper.

Availability of the software. All source code is available at https://github.com/denizzzture/Efficient-Batch-Dilithium.

References

1. Aguilar-Melchor, C., et al.: Batch signatures, revisited. Cryptology ePrint Archive (2023)
2. Alagic, G., et al.: Status report on the third round of the NIST post-quantum cryptography standardization process. US Department of Commerce, NIST (2022)
3. Alkim, E., et al.: Frodokem learning with errors key encapsulation. NIST PQC Stand.: Round **3** (2020)
4. Benjamin, D.: Batch signing for TLS. Internet Engineering Task Force, Internet-Draft Draft-Davidben-TLS-Batch-Signing-02 (2019)
5. Bos, J.W., Ofner, M., Renes, J., Schneider, T., van Vredendaal, C.: The matrix reloaded: Multiplication strategies in frodokem. In: Cryptology and Network Security: 20th International Conference, CANS 2021, Vienna, Austria, 13–15 December 2021, Proceedings 20, pp. 72–91. Springer (2021)
6. Cenk, M., Hasan, M.A.: On the arithmetic complexity of Strassen-like matrix multiplications. J. Symb. Comput. **80**, 484–501 (2017)
7. Chang, Y.S., Wu, T.C., Huang, S.C.: ElGamal-like digital signature and multisignature schemes using self-certified public keys. J. Syst. Softw. **50**(2), 99–105 (2000)
8. Ducas, L., et al.: Crystals-dilithium algorithm specifications and supporting documentation (version 3.1) (2021)
9. Elgamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Trans. Inf. Theory **31**(4), 469–472 (1985). https://doi.org/10.1109/TIT.1985.1057074
10. Fiat, A.: Batch RSA. In: Advances in Cryptology-CRYPTO 1989 Proceedings 9, pp. 175–185. Springer (1990)
11. Hwang, S.J., Lee, Y.H.: Repairing ElGamal-like multi-signature schemes using self-certified public keys. Appl. Math. Comput. **156**(1), 73–83 (2004)
12. Moriarty, K., Kaliski, B., Jonsson, J., Rusch, A.: PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017 (2016). https://doi.org/10.17487/RFC8017
13. Paquin, C., Stebila, D., Tamvada, G.: Benchmarking post-quantum cryptography in TLS. In: Ding, J., Tillich, J.-P. (eds.) PQCrypto 2020. LNCS, vol. 12100, pp. 72–91. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44223-1_5
14. Pavlovski, C., Boyd, C.: Efficient batch signature generation using tree structures. In: International workshop on cryptographic techniques and E-commerce, CrypTEC. vol. 99, pp. 70–77. Citeseer (1999)
15. Rosowski, A.: Fast commutative matrix algorithms. J. Symbolic Comput. **114**, 302–321 (2023). https://doi.org/10.1016/j.jsc.2022.05.002, https://www.sciencedirect.com/science/article/pii/S0747717122000499
16. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Rev. **41**(2), 303–332 (1999)
17. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: a performance study. Cryptology ePrint Archive (2020)
18. Strassen, V., et al.: Gaussian elimination is not optimal. Numer. Math. **13**(4), 354–356 (1969)
19. Tanwar, S., Kumar, A.: An efficient and secure identity based multiple signatures scheme based on rsa. Journal of Discrete Mathematical Sciences and Cryptography **22**(6), 953–971 (2019)
20. Winograd, S.: On multiplication of 2$\times$ 2 matrices. Linear Algebra Appl. **4**(4), 381–388 (1971)