



# Compact Memory Implementations of the ML-DSA Post-Quantum Digital Signature Algorithm

**Rodrigo Duarte de Meneses, Caio Teixeira, Marco Aurélio Amaral Henriques**

[1]Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas (Unicamp) – Campinas, SP – Brasil

r197962@dac.unicamp.br, caio@dca.fee.unicamp.br, maah@unicamp.br

***Abstract.*** *This paper explores memory optimization techniques in the implementation of the Module-Lattice-Based Digital Signature Algorithm (ML-DSA) in the context of post-quantum cryptography. It shows how to achieve significant reductions in memory usage, and evaluates the trade-offs in computational speed. Moreover, it demonstrates how the secret (private) key can be managed to reduce significantly its storage requirements, thereby enhancing ML-DSA's applicability in some resource-constrained environments.*

## 1. Introduction

With the recent advent of quantum computing, the cryptographic research community faces significant challenges due to the potential capabilities of quantum computers. One of the most critical issues is the threat quantum computers pose to current cryptographic systems, particularly public key algorithms like RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography) [Paar and Pelzl 2010]. The primary concern arises from Shor's algorithm [Shor 1994], a quantum algorithm that can efficiently solve the integer factorization and discrete logarithm problems, which are foundational to RSA and ECC, respectively. This capability threatens the security of vast amounts of data currently protected by these cryptographic schemes, potentially rendering them obsolete.

In response to these emerging challenges, post-quantum cryptography (PQC) has gained prominence. PQC aims to develop cryptographic algorithms secure against both classical and quantum attacks. Recognizing the importance of establishing robust post-quantum standards, the National Institute of Standards and Technology (NIST) initiated an international competition in 2016 to evaluate and select quantum-resistant cryptographic standards [NIST 2016]. This competition has identified several promising candidates, including lattice-based, hash-based, code-based, and isogeny-based cryptosystems. Each category has unique strengths and weaknesses but all share the common goal of resisting quantum computational attacks.

Among the various candidates, the Module-Lattice-Based Digital Signature Algorithm (ML-DSA) was defined as one of the standards for post-quantum digital signatures [NIST 2024]. ML-DSA is based on the CRYSTALS-Dilithium algorithm, a leading lattice-based digital signature scheme that has shown good performance in terms of security and efficiency [Lyubashevsky et al. 2021]. CRYSTALS-Dilithium leverages the hardness of lattice problems, specifically structured lattices, which are believed to be secure against both classical and quantum attacks. Lattice-based cryptography, in general, offers several advantages, including worst-case hardness guarantees and versatility in constructing various cryptographic primitives [Regev 2005].

Despite its potential, ML-DSA, like many post-quantum algorithms, encounters challenges related to efficiency and resource consumption, particularly in terms of memory usage. The large key sizes and complex mathematical operations inherent to lattice-based schemes can lead to significant memory demands, making their practical deployment difficult. Therefore, an efficient implementation of ML-DSA requires careful optimization to balance security and performance while minimizing resource consumption.

**Related work.** Previous work on ML-DSA has primarily focused on its earlier version, CRYSTALS-Dilithium [Lyubashevsky et al. 2021], with an emphasis on reducing latency by targeting specific processor architectures [Ji et al. 2024]. The work by [Bos et al. 2022] proposes some architecture-independent memory optimizations, using the PQClean implementations [Kannwischer et al. 2022] as a foundation. However, benchmarking data for the recently standardized version ML-DSA [NIST 2024] has not yet been provided.

**Paper contribution.** This paper investigates memory optimization techniques for ML-DSA to enhance its practical applicability while preserving its robust security properties. One of the key optimizations explored is the reduction of the memory footprint through in-place matrix-vector multiplications. By performing these operations in-place, the algorithm can significantly reduce the amount of memory required, making it more feasible to deploy ML-DSA in environments with constrained resources, such as embedded systems, IoT devices, and mobile platforms. Another important optimization is the reduction of the secret key size, which was obtained by delaying the calculation of several parameters that form the secret key until the moment they are needed for a signature. Thus, the key memory consumption of ML-DSA could be decreased, enhancing its suitability for various practical use cases where the storage of large secret keys is a problem.

The paper is organized as follows: Section 2 provides an overview of lattice-based cryptography, outlining the mathematical problems that ensure the security of ML-DSA. Section 3 delves into the operational details of the core functions of ML-DSA. The subsequent sections explore various strategies to optimize ML-DSA's memory usage and analyze the associated trade-offs in processing time and code size.

## 2. Lattice-Based Cryptography

Lattice-based cryptography is a promising area of cryptographic research that leverages the mathematical structure of lattices to develop secure cryptographic primitives. Lattice-based cryptography relies on hard mathematical problems over algebraic structures called lattices. These problems are believed to be difficult to solve even for quantum computers, making lattice-based schemes resistant to quantum attacks [Regev 2005]. This resilience against quantum computing threats positions lattice-based cryptography as a crucial element in the development of post-quantum cryptographic standards.

A lattice is a discrete subset of point in a $n$-dimensional vector space, denoted by $S$. Each point in $S$ can be represented as an integer linear combination of $n$ linearly independent vectors, which form a basis for the lattice. Formally, if $(\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n)$ are basis vectors, any point $\mathbf{p}$ in the lattice can be written as $\mathbf{p} = \lambda_1\mathbf{b}_1 + \lambda_2\mathbf{b}_2 + \cdots + \lambda_n\mathbf{b}_n$, where $\lambda_i \in \mathbb{Z}$. Different choices of basis vectors can describe the same lattice, and the quality of a basis can be characterized based on how straightforwardly points in $S$ can be expressed as linear combinations of the basis vectors (Figure 1).

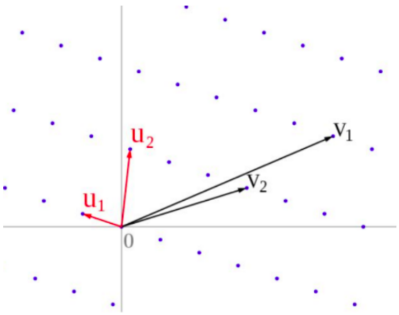


**Figure 1. Two possible basis for a given lattice.**

From these properties, various algebraic problems of particular interest to cryptography can be defined, such as the Learning With Errors (LWE) problem [Rivest et al. 1978]. In this problem, the goal is to distinguish a random vector from a vector that is perturbed by the addition of a small noise. Formally, given a ring $\mathbb{Z}_q$ of integers modulo $q$, the challenge is to discriminate a random vector $\mathbf{t} \in \mathbb{Z}_q^n$ from a vector $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$. Here, the $\mathbf{A}$ is a matrix with entries in $\mathbb{Z}_q$, and the secret vectors $\mathbf{s_1} \in \mathbb{Z}_q^\ell$ and $\mathbf{s_2} \in \mathbb{Z}_q^k$ have small coefficients within the range $[-\eta, \eta]$, introducing "noise" to the equation. Therefore, the difficulty lies in distinguishing $\mathbf{t}$ from a uniformly random vector in $\mathbb{Z}_q^n$ based only on the knowledge of $\mathbf{t}$, $\mathbf{A}$ and $q$.

The Module Learning With Errors (MLWE) is a generalization of LWE in which we use a polynomial $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ instead of $\mathbb{Z}_q$ [Lyubashevsky et al. 2010]. The security of ML-DSA is based on the MLWE problem, where the parameters of the polynomial ring $R_q$ are fixed for all security levels, with $q = 2^{23} - 2^{13} - 1$ and $n = 256$. This structure allows for more compact key sizes while retaining the strong security properties of standard LWE.

## 3. ML-DSA

In August 2024, NIST released the official version for the FIPS 204 standard, after review from the scientific community [NIST 2024]. The standard defines ML-DSA, a digital signature scheme based on CRYSTALS-Dilithium, and believed to be secure against an adversary using a large-scale quantum computer. The algorithm has its architecture based on the Fiat-Shamir with aborts applied to algebraic lattices, as proposed by [Lyubashevsky 2009].

The Algorithm 1 describes the generation of keys for ML-DSA. The function is responsible for generating a pair of secret and public keys $(sk, pk)$. KeyGen uses an extendable-output hash function (XOF) such as SHAKE-256 to expand a random seed $\xi$ to produce the seeds $\rho$, $\rho'$ and $K$ that will later be used for the expansion of random parameters. The basic operation consists on the "noisy" linear system $\mathbf{t} = \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2$, where $(\mathbf{A}, \mathbf{t})$ can be understood as an expanded public key. In practice, the public key is composed by a compressed version $\mathbf{t}_1$ (obtained by dropping the $d$ least significant bits from each coeficcient of $\mathbf{t}$), and the seed $\rho$ used for expanding the public matrix $\mathbf{A}$. The secret key consists in a byte encoding of the secret vectors $\mathbf{s_1}$ and $\mathbf{s_2}$, a vector $\mathbf{t}_0$ which encodes the dropped bits from $\mathbf{t}$, and the seeds $\rho$, $K$ and $tr$.

The Algorithm 3 outlines the signing procedure for ML-DSA, which takes a secret

key $sk$ and a message $M$ as inputs, and outputs a signature $\sigma$. The most fundamental mechanism of Sign is a rejection sampling loop. In each iteration of the loop, either a valid or an invalid signature is produced, which could potentially reveal information about the secret key. During the process, the signer generates a commitment $\mathbf{w}_1$ and derives a challenge $c$ from the the concatenation of $\mathbf{w}_1$ and the message representative $\mu$. Some variables are used to store intermediate products to avoid recomputation and are denoted by double brackets (e.g., $\langle\langle c\mathbf{t}_0 \rangle\rangle$).

The response $\mathbf{z}$ is then generated by using a pseudorandomly generated polynomial masking vector $\mathbf{y}$ with coefficients in the range $[1 - \gamma_1, \gamma_1]$ and the product $\langle\langle c\mathbf{s}_1 \rangle\rangle$. The response $\mathbf{z}$ is subjected to various tests that attests the signature validity, and iterates the rejection sampling loop until a valid response is produced. After sucessfully passing the validity tests, the signer computes a hint $\mathbf{h}$ that allows the verifiers to obtain $\mathbf{w}_1$ using $pk$. The final output signature is generated as a byte enconding of the hint $\mathbf{h}$, the valid response $\mathbf{z}$ and the commitment hash $\tilde{c}$.

The Algorithm 2 describes the verification process for ML-DSA, in which the verifier takes as inputs the public key $pk$, the message $M$ and the signature $\sigma$ and outputs 0 or -1 depending if the signature is valid or invalid, respectively. If the hint $\mathbf{h}$ is not properly encoded (denoted by "$\perp$"), the verifier will consider the signature invalid. The verifier attempts to reconstruct $\mathbf{w}_1$ from $pk$ and $\sigma$ by producing $\mathbf{w}'_1 = \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d$. Assuming that the secret vector $\mathbf{s}_2$ and the challenge $c$ have small coefficients:

$$\mathbf{w} = \mathbf{Ay} = \mathbf{Az} - c\mathbf{t} + c\mathbf{s}_2 \approx \mathbf{Az} - c\mathbf{t}_1 \cdot 2^d = \mathbf{w}'_1$$

And the verifier can reconstruct $\mathbf{w}_1$ from $\mathbf{w}'_1$ and the hint $\mathbf{h}$. Lastly, the signer's response $\mathbf{z}$ and hint $\mathbf{h}$ are subjected to a series of validity checks. If the checks are successful, the signature is considered valid.

## 4. Reducing Memory Footprint

The optimization of memory usage is a critical aspect in the development of efficient cryptographic algorithms, particularly in the context of post-quantum digital signatures like ML-DSA. This section explores the impact of memory optimizations on the performance of ML-DSA, focusing on enhancing space efficiency without compromising security. By reducing memory requirements, these optimizations aim to broaden the applicability of ML-DSA across resource-constrained platforms, thereby facilitating its adoption in various real-world scenarios.

### 4.1. In-Place Matrix Multiplication

The two main operations performed by ML-DSA are the expansion of random parameters from a seed and multiplications within the polynomial ring $R_q$ in the domain of the Number Theoretic Transform (NTT). Thus, varying the security level of ML-DSA involves changing the number of operations performed on the polynomial ring and the number of expansions, providing the algorithm with significant flexibility.

The most memory-intensive operations are those associated with the multiplications between the public matrix $\mathbf{A}$ and a polynomial vector in $R_q^\ell$ (e.g., Figure 2). Normally, to perform these operations, it is necessary to have both operands fully available

in memory. However, since the matrix $\mathbf{A}$ is generated pseudo-randomly and used exclusively for multiplications, it is possible to generate its elements sequentially without storing the entire matrix in memory.

ML-DSA uses three types of structures to perform these operations: **poly**, **polyvecl**, and **polyveck**, which store values in $R_q$, $R_q^\ell$ and $R_q^k$, respectively. By taking the first **row** of matrix $\mathbf{A}$ and the **input** vector, the product will be the first **entry** or the **output** vector. By performing this operation for each of the $k$ rows of matrix $\mathbf{A}$, we can completely construct the **output** vector. Thus, the multiplication can be done in-place by sequentially generating each **row** of $\mathbf{A}$ within the same **polyvecl** structure and operating with the **input** vector. This method is referred as vector-by-vector generation.

To further optimize memory usage, we can generate each **entry** of matrix $\mathbf{A}$ within the same **poly** structure and multiply it by the corresponding **entry** of the **input** vector. The results are accumulated in the **output** vector. After generating and multiplying each of the $k \times \ell$ **entries** of matrix $\mathbf{A}$, we fully construct the **output** vector. This approach is referred as polynomial-by-polynomial generation.

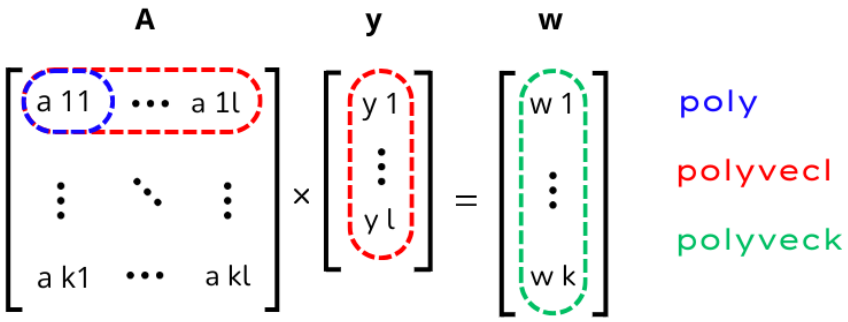


**Figure 2. Multiplication of the matrix $\mathbf{A}$ and vector $\mathbf{y}$ in ML-DSA.**

Each **poly** structure is stored in a buffer of 256 `(u)int32_t`s, corresponding to 1 KiB per polynomial. Therefore, the public matrix $\mathbf{A}$ requires up to $\{16, 30, 56\}$ KiB for ML-DSA-$\{44, 65, 87\}$. Using vector-by-vector generation, we can reduce the memory usage to a single **polyvecl**, which comprises $\ell$ **poly** structures, totaling $\{4, 5, 7\}$ KiB for ML-DSA-$\{44, 65, 87\}$. Furthermore, for polynomial-by-polynomial generation, we use a single **poly** structure to generate matrix $\mathbf{A}$, thereby needing only 1 KiB, regardless of the chosen security level.

### 4.2. Secret Key Compression

In many use cases as, for example, in those based on TPM (Trusted Platform Module), it is desirable to minimize the size of the secret key as much as possible. To achieve this, the authors of CRYSTALS-Dilithium suggest storing the seed $\xi$ and generating all private parameters as needed. Note that the secret key $sk$ is composed by $(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$, where $\rho$, $K$ and $tr$ are SHAKE-256 digests. However, storing polynomial vectors $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{t}_0$ requires significant space. Using the bit-packing techniques outlined in ML-DSA official documentation [NIST 2024], we achieve secret key sizes of $\{2528, 4000, 4864\}$ bytes for ML-DSA-$\{44, 65, 87\}$, respectively.

Following this idea of minimizing $sk$ involves delaying the generation of private parameters until the moment of signing, so that the KeyGen routine only needs to generate

parameters related to the public key. This modification can significantly reduce the size of the secret key but can also result in a notable increase in execution time, especially for the Sign routine. Moreover, restructuring the Sign routine to incorporate the generation of secret key parameters also leads to an increase in code size. All these factors must be considered to evaluate the feasibility of the proposed optimization.

## 5. Materials & Methods

The CRYSTALS-Dilithium PQClean implementation [Kannwischer et al. 2022] was used as the basis for the optimizations discussed in Section 4.1 and 4.2. This implementation does not rely on any architecture-specific instructions, and can therefore be implemented on any platform with sufficient resources.

We conducted measurements of CPU cycles and RAM usage peaks for the three primary functions of the ML-DSA: KeyGen, Sign, and Verify. We refer to the optimized versions, which implement vector-by-vector and polynomial-by-polynomial generation of the public matrix $\mathbf{A}$, as ML-DSA-v and ML-DSA-p, respectively. Additionally, we performed simulations using the KAT (Known Answer Test) vectors provided by NIST, comparing the outputs and intermediate values of our implementation with the reference PQClean version, thereby confirming the correctness and integrity of our approach.

For the CPU usage tests, the cicle counts were averaged over 15.000 iterations of KeyGen, Sign, and Verify for a randomly selected 59-byte message in each iteration. For the RAM usage evaluation, the analyses were performed for a single execution of the algorithm. ML-DSA is available in three versions: ML-DSA-$\{44, 65, 87\}$, which correspond to the NIST security levels $\{2, 3, 5\}$. These versions consist of distinct sets of parameters defined in the official documentation [NIST 2024], arranged in increasing order of security. We conducted analyses for all available security levels, assessing the impact of the optimizations for each case.

For the measurements, we used a laptop equipped with an Intel Core i7-1185G7 processor, 3.00 GHz CPU, OS Ubuntu 22.04.4 LTS, and 16 GiB RAM. We utilized the massif tool, available within the Valgrind 3.18.1 [1] suite, to measure and monitor RAM usage during each execution. For CPU cycle measurements, we employed the callgrind tool to generate the program's call tree, indicating how each function contributes to overall CPU usage. Additionally, we used the kcachegrind tool to visualize the obtained data.

## 6. Results & Discussion

The results regarding the techniques discussed in Section 4.1 are presented in Tables 1 and 2. Table 1 shows the peak RAM values for each ML-DSA function (KeyGen, Sign and Verify) across the different algorithm implementations, alongside the relative variation compared to the reference implementation ML-DSA. Table 2 presents the average number of CPU cycles and the percentual variation compared to the reference implementation.

As highlighted in the tables, ML-DSA-p brings the most significant reduction in RAM usage, as expected. However, there is a cost in the form of an increase in the number of CPU cycles, which is negligible in key generation and signature verification, but is very expressive in signature generation.

[1] https://valgrind.org/

**Table 1. RAM usage for ML-DSA implementations: reference (ML-DSA), vector-optimized (ML-DSA-v) and polynomial-optimized (ML-DSA-p).**

| | | NIST Security Level | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 | | 3 | | 5 | |
| | | KiB | Δ | KiB | Δ | KiB | Δ |
| ML-DSA | KeyGen | 43.9 | - | 68.9 | - | 107.6 | - |
| | Sign | 57.3 | - | 87.2 | - | 132.0 | - |
| | Verify | 41.9 | - | 65.8 | - | 102.8 | - |
| ML-DSA-v | KeyGen | 32.0 | -27.1% | 43.9 | -36.3% | 58.6 | -45.5% |
| | Sign | 45.3 | -20.9% | 62.2 | -28.7% | 83.0 | -37.1% |
| | Verify | 29.9 | -28.6% | 40.9 | -37.8% | 53.9 | -47.6% |
| ML-DSA-p | KeyGen | **30.0** | **-31.7%** | **40.9** | **-40.6%** | **53.7** | **-50.1%** |
| | Sign | **43.4** | **-24.3%** | **59.3** | **-32.0%** | **78.0** | **-41.0%** |
| | Verify | **28.0** | **-33.2%** | **36.7** | **-44.2%** | **48.9** | **-52.4%** |

As shown in Table 1, there is a clear and significant impact on RAM usage. Notably, there is a substantial reduction in memory consumption during the KeyGen and Verify routines. This outcome is expected because multiplications with the public matrix $\mathbf{A}$ accounts for the majority of memory usage in these functions. Additionally, the reduction in memory usage is more pronounced for higher security levels, as the dimensions of the public matrix $\mathbf{A}$ are larger in these cases. The changes in code size that occupies the system ROM were negligible in these implementations.

In Table 2, it is observed that the increase in CPU cycles is primarily noticeable in the Sign routine, whereas KeyGen and Verify show a much less significant impact. This phenomenon arises because the public matrix $\mathbf{A}$ is generated multiple times during signing due to the rejection sampling mechanism. Since the entire matrix is never stored in memory, it must be recalculated when needed, increasing processing time.

**Table 2. Number of CPU cycles for ML-DSA implementations: reference (ML-DSA), vector-optimized (ML-DSA-v) and polynomial-optimized (ML-DSA-p).**

| | | NIST Security Level | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 | | 3 | | 5 | |
| | | kcc | Δ | kcc | Δ | kcc | Δ |
| ML-DSA | KeyGen | 190 | - | 350 | - | 533 | - |
| | Sign | 838 | - | 1416 | - | 1717 | - |
| | Verify | 209 | - | 334 | - | 551 | - |
| ML-DSA-v | KeyGen | 191 | +0.5% | 352 | +0.6% | 537 | +0.8% |
| | Sign | 1141 | +36.2% | 2057 | +45.3% | 2538 | +47.8% |
| | Verify | 211 | +1.0% | 335 | +0.3% | 554 | +0.5% |
| ML-DSA-p | KeyGen | **191** | **+0.5%** | **352** | **+0.6%** | **537** | **+0.8%** |
| | Sign | **1155** | **+37.8%** | **2081** | **+47.0%** | **2549** | **+48.5%** |
| | Verify | **211** | **+1.0%** | **335** | **+0.3%** | **554** | **+0.5%** |

The data indicates a trade-off between memory reduction and increased processing for the algorithm. The most significant trade-off is observed in the Sign routine, while the increase in processing for KeyGen and Verify is practically negligible. Furthermore, it can be seen that the increase in processing is approximately the same for KeyGen and Verify

in both ML-DSA-v and ML-DSA-p implementations. This suggests that generating the public matrix $\mathbf{A}$ polynomial-by-polynomial does not significantly affect the algorithm's execution speed compared to vector-by-vector generation.

With respect to the secret key compression (Section 4.2), we compared the average number of CPU cycles in the reference implementation and the version with secret key compression, as shown in Table 3. It is worth noting that the secret key compression has no impact on the Verify function, and therefore we opted to only present the results for KeyGen and Sign. Since the seed $\xi$ is only 256-bit, we achieve a reduction of $\{316, 500, 608\}$-fold in the size of the secret key for ML-DSA-$\{44, 65, 87\}$, respectively. It should be mentioned that there was no impact on memory usage since the RAM peak values are primarily associated with the matrix-vector polynomial multiplications, and they remain the same for all security levels.

**Table 3. Comparison of the average number of CPU cycles for ML-DSA with secret key compression (ssk) and reference implementation.**

| | | NIST Security Level | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 | | 3 | | 5 | |
| | | kcc | $\Delta$ | kcc | $\Delta$ | kcc | $\Delta$ |
| ML-DSA | KeyGen | 190 | - | 350 | - | 533 | - |
| | Sign | 838 | - | 1416 | - | 1717 | - |
| ML-DSA (ssk) | KeyGen | **180** | **-5.3%** | **329** | **-6.0%** | **500** | **-6.6%** |
| | Sign | **1350** | **+61.1%** | **2459** | **+73.7%** | **4138** | **+141.0%** |

However, there was a change in the code size after this compression. The compiled code that will be stored in the system ROM has changed from 7.3 KiB to 9.6 KiB, a variation of 24%. This change in code size is another parameter that must be considered when deciding which kind of optimization is the best for a given application. The optimized versions of the ML-DSA code are available in the stable GitHub repository: `https://github.com/regras/ml-dsa_in_place`.

## 7. Conclusion

The memory optimizations implemented for the post-quantum digital signature algorithm ML-DSA have yielded significant reduction in memory usage while maintaining its security and integrity. This achievement is crucial as it facilitates the algorithm's implementation on computationally constrained platforms, thereby enabling its adoption across diverse use cases. There is a price to pay for such reduction: an increase in processing time, which has to be evaluated as a trade-off with respect to the memory usage.

Future research could be done on optimizing the resource-intensive computation of the Number Theoretic Transform (NTT), exploring various transform architectures that explore trade-offs between memory usage and processing efficiency. Additionally, further research could involve incorporating threat modeling to identify potential vulnerabilities, as well as conducting statistical analysis to assess the robustness of ML-DSA under various attack scenarios. By refining computational techniques and exploring new architectural optimizations, researchers can enhance the efficiency and applicability of post-quantum cryptographic algorithms in diverse computing environments.

## References

Bos, J., Renes, J., and Sprenkels, A. (2022). Dilithium for memory constrained devices. Cryptology ePrint Archive, Paper 2022/323. https://eprint.iacr.org/2022/323.

Ji, X., Dong, J., Huang, J., Yuan, Z., Dai, W., Xiao, F., and Lin, J. (2024). ECO-CRYSTALS: Efficient cryptography CRYSTALS on standard RISC-v ISA. Cryptology ePrint Archive, Paper 2024/1198. https://eprint.iacr.org/2024/1198.

Kannwischer, M. J., Schwabe, P., Stebila, D., and Wiggers, T. (2022). Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&P 2022*, pages 19–30, Los Alamitos, CA, USA. IEEE Computer Society. https://eprint.iacr.org/2022/337.

Lyubashevsky, V. (2009). Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*. https://api.semanticscholar.org/CorpusID:11853511.

Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., and Stehlé, D. (2021). Crystals-dilithium: Algorithm specification and supporting documentation. https://pq-crystals.org/dilithium/.

Lyubashevsky, V., Peikert, C., and Regev, O. (2010). On ideal lattices and learning with errors over rings. Lecture Notes in Computer Science, vol 6110. https://doi.org/10.1007/978-3-642-13190-5_1.

NIST (2016). Request for comments on post-quantum cryptography requirements and evaluation criteria. https://csrc.nist.gov/projects/post-quantum-cryptography.

NIST (2024). Fips 204: Module-lattice-based digital signature standard. https://doi.org/10.6028/NIST.FIPS.204.

Paar, C. and Pelzl, J. (2010). *Understanding Cryptography*. Springer. https://doi.org/10.1007/978-3-642-04101-3.

Regev, O. (2005). On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing,*, Baltimore, MD, USA. https://dx.doi.org/10.1145/1568318.1568324.

Rivest, R., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the Association for Computing Machinery. https://doi.org/10.1145/359340.359342.

Shor, P. W. (1994). Algorithms for quantum computation: discrete logarithm and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124 – 134. https://api.semanticscholar.org/CorpusID:15291489.

## Appendixes

**Algorithm 1** ML-DSA.KeyGen_internal($\xi$) **Source:** [NIST 2024]

**Input:** Seed $\xi \in \mathbb{B}^{32}$
**Output:** Public key $pk \in \mathbb{B}^{32+32k(\text{bitlen}(q-1)-d)}$
and private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot\text{bitlen}(2\eta)+dk)}$

$(\rho, \rho', K) \in \mathbb{B}^{32} \times \mathbb{B}^{64} \times \mathbb{B}^{32} \leftarrow \text{H}(\xi \parallel \text{IntegerToBytes}(k, 1) \parallel \text{IntegerToBytes}(\ell, 1), 128)$
▷ expand seed
$\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow \text{ExpandS}(\rho')$
$\mathbf{t} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{s}_1)) + \mathbf{s}_2$ ▷ compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
$(\mathbf{t}_1, \mathbf{t}_0) \leftarrow \text{Power2Round}(\mathbf{t})$ ▷ compress $\mathbf{t}$
▷ PowerTwoRound is applied componentwise
$pk \leftarrow \text{pkEncode}(\rho, \mathbf{t}_1)$
$tr \leftarrow \text{H}(pk, 64)$
$sk \leftarrow \text{skEncode}(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0)$ ▷ $K$ and $tr$ are for use in signing
**return** $(pk, sk)$

**Algorithm 2** ML-DSA.Verify_internal($pk, M', \sigma$) **Source:** [NIST 2024]

**Input:** Public key $pk \in \mathbf{B}^{32+32k(\text{bitlen}(q-1)-d)}$ and message $M' \in \{0,1\}^*$
**Input:** Signature $\sigma \in \mathbf{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}(\gamma_1-1))+\omega+k}$
**Output:** Boolean

$(\rho, \mathbf{t}_1) \leftarrow \text{pkDecode}(pk)$
$(\tilde{c}, \mathbf{z}, \mathbf{h}) \leftarrow \text{sigDecode}(\sigma)$ ▷ signer's commitment hash $\tilde{c}$, response $\mathbf{z}$, and hint $\mathbf{h}$
**if** $\mathbf{h} = \perp$ **then return** false ▷ hint was not properly encoded
**end if**
$\hat{\mathbf{A}} \leftarrow \text{ExpandA}(\rho)$ ▷ $\mathbf{A}$ is generated and stored in NTT representation as $\hat{\mathbf{A}}$
$tr \leftarrow \text{H}(pk, 64)$
$\mu \leftarrow \text{H}(\text{BytesToBits}(tr) \parallel M', 64)$ ▷ message representative that may optionally be computed in a different cryptographic module
$c \in R_q \leftarrow \text{SampleInBall}(\tilde{c})$ ▷ compute verifier's challenge from $\tilde{c}$
$\mathbf{w}'_{\text{Approx}} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{z}) - \text{NTT}(c) \circ \text{NTT}(\mathbf{t}_1 \cdot 2^d))$ ▷ $\mathbf{w}'_{\text{Approx}} = \mathbf{A}\mathbf{z} - c\mathbf{t}_1 \cdot 2^d$
$\mathbf{w}'_1 \leftarrow \text{UseHint}(\mathbf{h}, \mathbf{w}'_{\text{Approx}})$ ▷ reconstruction of signer's commitment
▷ UseHint is applied componentwise
$\tilde{c}' \leftarrow \text{H}(\mu \parallel \text{w1Encode}(\mathbf{w}'_1), \lambda/4)$ ▷ hash it; this should match $\tilde{c}$
**return** $[[\|\mathbf{z}\|_\infty < \gamma_1 - \beta]]$ **and** $[[\tilde{c} = \tilde{c}']]$

**Algorithm 3** ML-DSA.Sign_internal$(sk, M', rnd)$ **Source:** [NIST 2024]

**Input:** Private key $sk \in \mathbb{B}^{32+32+64+32\cdot((\ell+k)\cdot \text{bitlen}(2\eta)+dk)}$, formatted message $M' \in \{0,1\}^*$, and per message randomness or dummy variable $rnd \in \mathbb{B}^{32}$
**Output:** Signature $\sigma \in \mathbb{B}^{\lambda/4+\ell\cdot 32\cdot(1+\text{bitlen}(\gamma_1-1))+\omega+k}$

$(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) \leftarrow$ skDecode$(sk)$
$\hat{\mathbf{s}}_1 \leftarrow$ NTT$(\mathbf{s}_1)$
$\hat{\mathbf{s}}_2 \leftarrow$ NTT$(\mathbf{s}_2)$
$\hat{\mathbf{t}}_0 \leftarrow$ NTT$(\mathbf{t}_0)$
$\hat{\mathbf{A}} \leftarrow$ ExpandA$(\rho)$ ▷ $\hat{\mathbf{A}}$ is generated and stored in NTT representation as $\mathbf{A}$
$\mu \leftarrow$ H(BytesToBits$(tr) \parallel M', 64)$ ▷ message representative that may optionally be computed in a different cryptographic module
$\rho'' \leftarrow$ H$(K \parallel rnd \parallel \mu, 64)$ ▷ compute private random seed
$\kappa \leftarrow 0$ ▷ initialize counter $\kappa$
$(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
**while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do** ▷ rejection sampling loop
    $\mathbf{y} \in R_q^\ell \leftarrow$ ExpandMask$(\rho'', \kappa)$
    $\mathbf{w} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}} \circ \text{NTT}(\mathbf{y}))$
    $\mathbf{w}_1 \leftarrow$ HighBits$(\mathbf{w})$ ▷ signer's commitment
    ▷ HighBits is applied componentwise
    $\tilde{c} \leftarrow$ H$(\mu \parallel$ w1Encode$(\mathbf{w}_1), \lambda/4)$ ▷ commitment hash
    $c \in R_q \leftarrow$ SampleInBall$(\tilde{c})$ ▷ verifier's challenge
    $\hat{c} \leftarrow$ NTT$(c)$
    $\langle\langle c\mathbf{s}_1 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_1)$
    $\langle\langle c\mathbf{s}_2 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2)$
    $\mathbf{z} \leftarrow \mathbf{y} + \langle\langle c\mathbf{s}_1 \rangle\rangle$ ▷ signer's response
    $\mathbf{r}_0 \leftarrow$ LowBits$(\mathbf{w} - \langle\langle c\mathbf{s}_2 \rangle\rangle)$
    ▷ LowBits is applied componentwise
    **if** $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ **or** $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ **then** $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$ ▷ validity checks
    **else**
        $\langle\langle c\mathbf{t}_0 \rangle\rangle \leftarrow \text{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)$
        $\mathbf{h} \leftarrow$ MakeHint$(-\langle\langle c\mathbf{t}_0 \rangle\rangle, \mathbf{w} - \langle\langle c\mathbf{s}_2 \rangle\rangle + \langle\langle c\mathbf{t}_0 \rangle\rangle)$ ▷ Signer's hint
        ▷ MakeHint is applied componentwise
        **if** $||\langle\langle c\mathbf{t}_0 \rangle\rangle||_\infty \geq \gamma_2$ **or** the number of 1's in $\mathbf{h}$ is greater than $\omega$ **then** $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
        **end if**
    **end if**
    $\kappa \leftarrow \kappa + \ell$ ▷ increment counter
**end while**
$\sigma \leftarrow$ sigEncode$(\tilde{c}, \mathbf{z} \bmod^{\pm} q, \mathbf{h})$
**return** $\sigma$