İSTİNYE ÜNİVERSİTESİ İSTANBUL

Capstone Project Final Report

Submitted to

Faculty of Engineering and Natural Sciences

Department of *Computer Engineering*

In partial fulfillment of the requirements for the degree

BACHELOR of SCIENCE in *COMPUTER ENGINEERING*

"*Implementation of Post − Quantum Secure Digital Signature Schemes*"

Student: İREM SUDE SALĞAR

ID Number: 200722070

Supervisor:

SEDAT AKLEYLEK

Istanbul

May 2025

Abstract

This study investigates the integration of alternative polynomial multiplication algorithms Karatsuba and Toom Cook 4 into the CRYSTALS-Dilithium2 digital signature scheme, a NIST recommended post-quantum cryptographic standard. The reference implementation of Dilithium2 uses the Number Theoretic Transform (NTT) to accelerate polynomial multiplications. However, NTT can be memory-intensive and difficult to optimize for certain hardware platforms. To address this, the original NTT based routines were removed and replaced with classical recursive multiplication algorithms. Karatsuba was selected for its divide and conquer efficiency ($O(n^{1.58})$), while Toom Cook 4 allowed higher order evaluations at seven distinct points, fitting Dilithium2's 256 coefficient structure. Custom C implementations were integrated into LibOQS's x86_64-ref variant, and both correctness and performance were validated using unit tests, gprof based profiling, and Valgrind memory analysis. Experimental results revealed that Toom Cook 4 outperforms Karatsuba in terms of latency under high load, while both significantly alter function level performance profiles. This work demonstrates that classical multiplication techniques can be viable drop-in replacements for NTT in resource constrained or NTT incompatible environments, laying the groundwork for future hybrid and adaptive implementations.

Keywords: Post-Quantum Cryptography, Dilithium2, Polynomial Multiplication, Karatsuba, Toom-Cook-4

Table of Contents

List of Tables

List of Figures

List of Abbreviations

PQC	Post-Quantum Cryptography
NTT	Number Theoretic Transform
SHAKE256	Secure Hash Algorithm with Keccak Extension
OQS	Open Quantum Safe
AVX2	Advanced Vector Extensions 2
VM	Virtual Machine
API	Application Programming Interface
LLVM	Low Level Virtual Machine (Compiler Suite)

Section 1:

Introduction

1. Introduction

1.1 Background and Motivation

The rapid advancements in quantum computing are poised to revolutionize computational paradigms while simultaneously threatening the cryptographic primitives that underlie modern digital communication. Shor's groundbreaking algorithm (Shor, 1994) demonstrated that quantum systems can efficiently solve the integer factorization and discrete logarithm problems—the mathematical backbones of RSA and Elliptic Curve Cryptography (ECC). This directly compromises the security guarantees provided by these widely deployed cryptographic protocols.

In response to these emerging threats, the National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography (PQC) Standardization Project to identify cryptographic algorithms resilient against quantum adversaries (NIST, 2025). Among the algorithms selected for standardization, CRYSTALS-Dilithium, a lattice-based digital signature scheme, gained prominence due to its reliance on the Module Learning With Errors (MLWE) problem—a hardness assumption believed to be secure against both classical and quantum attacks (PQCRYSTALS, 2025; NIST, 2024).

The formal standardization of Dilithium was announced with the release of FIPS 204, which includes specifications for Dilithium2, Dilithium3, and Dilithium5, each corresponding to NIST's security levels 2, 3, and 5 respectively (NIST, 2024). A key feature of Dilithium's design is the use of the Fiat-Shamir with Aborts transformation, which significantly improves efficiency and reduces signature sizes (Lyubashevsky, 2009).

Beyond theoretical security, practical implementation and performance benchmarking are crucial for real-world deployment. Performance evaluations conducted by Open Quantum Safe (2025) show that Dilithium2 achieves approximately 2,099 signatures per second and 8,752 verifications per second on x86_64-ref platforms. It also maintains moderate memory usage, making it suitable for constrained systems such as IoT devices (Open Quantum Safe, 2025a, 2025b).

A critical component of Dilithium's efficiency is its use of the Number Theoretic Transform (NTT) for polynomial multiplication. However, alternative multiplication strategies—most notably Karatsuba and Toom-Cook—offer compelling trade-offs in terms of computational complexity, memory efficiency, and cycle latency (Kannwischer, 2022; Aikata et al., 2022; Bermudo Mera et al., 2020; Pratap et al., 2022; Zanoni, 2009). These alternatives are particularly valuable in environments with limited hardware resources, such as embedded systems and cryptographic accelerators.

The significance of developing quantum-resistant cryptographic schemes extends beyond academia. Critical sectors including finance, defense, healthcare, and national infrastructure must adapt to this new cryptographic landscape to preserve long-term data authenticity and integrity. Therefore, developing practical, high-performance post-quantum digital signatures is not only a technical challenge but also a strategic necessity.

1.2 Problem Statement

Although the Dilithium2 algorithm has been standardized and is founded on a strong security framework, its performance in resource-constrained or latency-sensitive environments is significantly affected by the efficiency of its underlying polynomial multiplication method. While the Number Theoretic Transform (NTT) offers asymptotic speed advantages, it also introduces notable implementation complexity and memory overhead. In contrast, Karatsuba and Toom-Cook multiplication algorithms present structurally simpler alternatives with potentially more favorable constant-time behavior—though with differing computational trade-offs (Kannwischer, 2022; Bermudo Mera et al., 2020; Aikata et al., 2022; Zanoni, 2009).

This discrepancy motivates the central research question of this thesis:

> In the context of the Dilithium2 digital signature scheme, how do alternative polynomial multiplication algorithms—such as Karatsuba and Toom-Cook—compare with NTT in terms of performance, memory usage, and correctness under real-world constraints?

1.3 Research Objectives or Scope

The primary aim of this project is to explore and evaluate the integration of Karatsuba and Toom-Cook multiplication methods as alternatives to NTT within the Dilithium2 digital signature scheme. To achieve this:

- Multiple variants of Dilithium2 were implemented using the Open Quantum Safe (LibOQS) library (Open Quantum Safe, 2025).
- The correctness of key generation, signature creation, and signature verification was validated through functional testing.
- Performance profiling was conducted using the gprof tool, measuring execution cycles and runtime over 1,000 to 100,000 iterations for each multiplication method.
- Each algorithm was evaluated based on critical performance metrics such as signature throughput, verification speed, memory footprint, and round-trip correctness.

1.4 Contributions

This thesis makes the following contributions to the field of post-quantum cryptography:

- Benchmark-Based Comparison: Experimental measurement and comparison of Dilithium2 variants implemented with NTT, Karatsuba, and Toom-Cook multiplication on real systems.
- Integration with LibOQS: A modular implementation developed under the Open Quantum Safe framework, fully compliant with NIST-defined parameter sets (Open Quantum Safe, 2025).
- Correctness Validation: Round-trip signature verification was performed on all variants to ensure mathematical integrity and conformance with Dilithium specifications (PQCRYSTALS, 2025; NIST, 2024).
- Algorithmic Insights: In-depth analytical discussion on the computational complexity and implementation trade-offs of alternative polynomial multipliers in lattice-based digital signature schemes.

Section 2:

Literature Review

2. Literature Review

2.1 Rationale for Choosing Dilithium2 and Baseline Performance Analysis

The Post-Quantum Cryptography Standardization Project initiated by the National Institute of Standards and Technology (NIST) has systematically evaluated numerous candidate algorithms across various security levels (NIST, 2025). Within this initiative, module-lattice-based digital signature schemes have emerged as robust options due to their solid mathematical foundations and resistance to quantum attacks. Among these, CRYSTALS-Dilithium was formally selected and standardized through FIPS 204, confirming its suitability for post-quantum digital signatures (NIST, 2024).

CRYSTALS-Dilithium offers multiple parameter sets—Dilithium2, Dilithium3, and Dilithium5—corresponding to different security levels. Among these, Dilithium2 stands out for software-based implementations, primarily due to its reduced stack memory usage and favorable runtime characteristics (PQCRYSTALS, 2025). These features make it highly compatible with resource-constrained computing environments.

For this project, the x86_64-ref variant was chosen as the reference platform for all benchmarking and comparative analysis. This variant represents the unoptimized baseline implementation provided by the Open Quantum Safe (LibOQS) framework and is ideal for evaluating the isolated impact of algorithmic modifications without hardware-specific optimizations (Open Quantum Safe, 2025).

According to performance data published by Open Quantum Safe, Dilithium2's reference implementation achieves approximately 9,486 key generations, 2,099 signatures, and 8,752 verifications per second on x86_64-ref systems (Open Quantum Safe, 2025c). Its memory usage during key generation is as low as 12,576 bytes (Open Quantum Safe, 2025b), demonstrating its efficiency in low-resource environments.

To maintain consistency and ensure isolated analysis, all algorithmic optimizations in this study were applied exclusively to this reference implementation. This approach enabled precise measurement of performance, memory footprint, and correctness variations resulting from alternative polynomial multiplication strategies.

2.2 Use of NTT in Literature and Alternative Polynomial Multiplication Techniques

Polynomial operations are fundamental to post-quantum digital signature algorithms, especially within lattice-based cryptographic constructions. The Number Theoretic Transform (NTT)—a modular field analogue of the classical Fast Fourier Transform (FFT)—is widely utilized for fast polynomial multiplication. CRYSTALS-Dilithium leverages the NTT to strike a balance between computational efficiency and mathematical accuracy (PQCRYSTALS, 2025; NIST, 2024; Kannwischer, 2022).

In his doctoral dissertation, Kannwischer (2022) provided a detailed analysis of NTT's role in lattice-based cryptosystems including Dilithium, Kyber, and Saber, examining its performance across different processor architectures such as AVX2, Cortex-M4, and ARMv8. While the NTT provides asymptotic performance advantages for high-degree polynomials, its reliance on precomputed parameters, memory-intensive operations, and tight parameter constraints poses practical challenges in certain deployments.

These challenges have motivated the exploration of alternative polynomial multiplication strategies. Two prominent alternatives are the Karatsuba and Toom-Cook multiplication algorithms. Due to their relatively lower computational and memory requirements, these algorithms are particularly appealing in embedded systems and resource-constrained cryptographic hardware (Aikata et al., 2022; Pratap et al., 2022; Bermudo Mera et al., 2020; Zanoni, 2009).

In a comparative hardware-level study, Khan, Ullah, and Ahmad (2022) examined the performance of Karatsuba, Toom-Cook, and NTT-based multipliers within the NTRU cryptographic framework. Their analysis revealed that although NTT excels in raw computational speed, Karatsuba and Toom-Cook exhibit significant advantages in hardware area efficiency and energy consumption—critical metrics for FPGA-based and constrained environments. This underscores the relevance of these alternatives in Dilithium2’s context, where trade-offs between latency and hardware efficiency are essential.

Wardhani, Putra, and Nurdin (2024) further extended this discussion to quantum circuit design by evaluating the efficiency of high-degree and half-degree multipliers in post-quantum settings.

Their results emphasized the importance of multiplication algorithms that reduce quantum resource costs, reinforcing the value of exploring non-NTT techniques in quantum-resistant digital signature schemes.

The Karatsuba algorithm, originally introduced in 1962, reduces the multiplication complexity from $O(n^2)$ to approximately $O(n^{1.58})$. This divide-and-conquer strategy is especially suitable for recursively splitting and processing polynomials in LWE-based systems. Pratap et al. (2022) demonstrated Karatsuba's effectiveness in accelerating signature generation on FPGA and ARM Cortex platforms.

Toom-Cook, as a generalization of Karatsuba, is optimized for higher-degree polynomials. Variants such as Toom-3, Toom-4, and Toom-8 split polynomials into more segments before evaluating and recombining. Bermudo Mera, Karmakar, and Verbauwhede (2020) analyzed the trade-offs between memory access and computation time in Toom-Cook multiplication, concluding that a well-optimized Toom-Cook implementation can yield improvements in both execution speed and energy consumption.

Finally, Zanoni (2009) introduced the Toom-Cook 8-way technique for large-integer polynomial multiplication, optimized through improved interpolation routines. Though resource-intensive, this method has been shown to provide substantial performance benefits in modular arithmetic-heavy cryptographic algorithms.

2.3 Research Gaps and This Project's Contribution

While the Number Theoretic Transform (NTT) remains the de facto standard for fast polynomial multiplication in lattice-based cryptographic schemes, a critical gap exists in the comparative evaluation of alternative multiplication strategies—particularly Karatsuba and Toom-Cook—within standardized, open-source implementations like CRYSTALS-Dilithium2. Prior work, such as that by Kannwischer (2022), has focused on optimizing NTT for specific hardware architectures (e.g., AVX2, ARMv8), while studies by Khan et al. (2022) and Bermudo Mera et al. (2020) explored alternatives mostly in FPGA or theoretical contexts. However, these studies lack direct side-by-side comparisons under uniform baseline conditions, and they often

omit software-level concerns such as round-trip correctness, modular integration, and function-specific performance bottlenecks.

This thesis addresses these limitations through a comprehensive, implementation-level analysis based on the x86_64-ref variant of LibOQS. Specifically, it contributes:

- Algorithmic Integration: A modular implementation of Dilithium2 using three multiplication methods (NTT, Karatsuba, Toom-Cook) within a consistent software environment.
- Quantitative Benchmarking: Extensive performance testing across varying iteration scales (1,000–100,000) to evaluate runtime, memory footprint, and signature throughput.
- Profiling and Bottleneck Analysis: Fine-grained profiling via gprof and memory analysis with valgrind to identify computational hot spots and guide optimization efforts.
- Correctness and Portability Validation: Round-trip signature verification and compliance with Dilithium specifications, ensuring viability in embedded and low-resource platforms.

By bridging theoretical alternatives with practical implementation, this study establishes a reproducible framework for evaluating non-NTT polynomial multiplications within post-quantum digital signature schemes and contributes new empirical insights for future standardization and deployment.

2.4 Tools, Platforms, and the LibOQS Library

This research goes beyond theoretical exploration and is rooted in practical, implementation-level experimentation on an open-source cryptographic framework. The choice of compilation tools, runtime environment, and profiling instrumentation plays a crucial role in ensuring the accuracy, reproducibility, and generalizability of performance results. By isolating hardware dependencies and ensuring modularity in implementation, this project creates a controlled environment ideal for benchmarking algorithmic variants under uniform conditions.

2.4.1 LibOQS Library

The Open Quantum Safe (LibOQS) library (Open Quantum Safe, 2025) serves as the cryptographic foundation of this work. Developed to support experimentation with post-quantum cryptographic algorithms, LibOQS implements a range of NIST PQC finalists as well as other lattice- and code-based schemes, unified under a consistent API interface. This modular architecture is ideal for comparative algorithm testing, as it enables quick swapping of cryptographic primitives while preserving the same software infrastructure.

For this project, LibOQS was manually configured to support only the Dilithium2 signature scheme. All unused algorithms and KEM (Key Encapsulation Mechanism) primitives were excluded at build time. This minimal configuration provided several benefits:

- Faster compilation and reduced binary size,
- Elimination of library-level noise in profiling results,
- Isolation of performance metrics specific to Dilithium2.

Moreover, the modularity of the library facilitated seamless integration of custom polynomial multiplication routines (Karatsuba and Toom-Cook), allowing all variants to be benchmarked using a consistent framework.

2.4.2 Development and Test Environment

All experiments were conducted in a virtualized Ubuntu 22.04 environment, selected to ensure platform independence, fine-grained resource control, and a reproducible software stack. This decision also aligns with common cryptographic development practices where virtual machines serve as neutral baselines for performance testing.

The environment included a modern C development toolchain and the following build systems:

- CMake: Employed for scalable and modular build configuration, CMake allowed flexible integration of custom source files, conditional compilation flags, and platform-specific optimizations.
- Ninja: Used in conjunction with CMake, Ninja drastically reduced build times by efficiently managing parallel compilation tasks—particularly beneficial during iterative optimization and debugging cycles.

This setup ensured consistent and repeatable builds, reducing external variability and improving confidence in the results obtained from performance and correctness tests.

2.4.3 Profiling Tool: gprof

To capture fine-grained performance data, the GNU Profiler (gprof) was employed. Unlike high-level benchmarking tools that measure only total execution time, gprof offers a hierarchical breakdown of CPU time spent in each function, enabling the identification of performance bottlenecks within the codebase.

Profiling focused on three critical routines in the Dilithium2 signature pipeline:

- Key Pair Generation
- Signature Generation
- Signature Verification

By instrumenting all algorithmic variants (NTT, Karatsuba, and Toom-Cook), gprof provided precise statistics for each multiplication technique's impact on the signing workflow. These results were instrumental in assessing:

- CPU-bound function hotspots,
- The cost of recursive and evaluation/interpolation routines,
- The effectiveness of replacing NTT with alternative algorithms in practice.

The use of gprof not only added depth to the quantitative analysis but also improved the interpretability of results, supporting claims about efficiency, correctness, and scalability.

2.5 Profiling Analysis and Evaluation Based on Common Function Set

In this study, the reference implementation of Dilithium2 (x86_64-ref) was profiled using gprof for the crypto_sign_keypair, crypto_sign_signature, and crypto_sign_verify functions over 1,000 and 10,000 iterations. The goal was to identify the subfunctions contributing the most to overall execution time and to guide targeted optimizations.

2.5.1 Shared Function Usage

The profiling results revealed the following functions consistently consumed the highest execution time across all three stages:

The gprof output results shown in Appendix F were obtained using the standard testing code provided with Dilithium2, which sequentially executes the crypto_sign_keypair, crypto_sign, crypto_sign_verify, and crypto_sign_free functions over 10,000 iterations.

Rank	Function Name	Avg. Time (%)	Affected Functions
1	KeccakP1600_Permute_24rounds	~30%	Keygen, Sign, Verify
2	pqcrystals_dilithium2_ref_invntt_tomont	~18%	Keygen, Sign, Verify
3	pqcrystals_dilithium2_ref_ntt	~15%	Keygen, Sign, Verify
4	pqcrystals_dilithium2_ref_montgomery_reduce	~14%	Keygen, Sign, Verify
5	poly_pointwise_montgomery and reduce calls	<6%	Keygen, Sign, Verify

Table 1. analysis of common functions

Six additional functions were confirmed to be shared across all three routines:

- shake256
- polyvec_matrix_expand
- polyvecl_ntt
- polyvec_matrix_pointwise_montgomery
- polyveck_reduce
- polyveck_invntt_tomont

Additional shared functions such as poly_challenge, poly_ntt, and shake256_inc_* were also common to the signing and verification phases

2.5.2 Interpretation and Impact

This profiling not only visualizes time distribution at a micro level but also helps identify high-impact targets for optimization. For example

- KeccakP1600_Permute_24rounds dominates SHAKE256-related operations.
- invntt_tomont, ntt, and montgomery_reduce drive the cost of NTT-based multiplication.

These insights are pivotal when evaluating alternatives such as Karatsuba and Toom-Cook, which offer lighter computational paths. As a result, gprof profiling provided a high-resolution roadmap for guiding algorithmic improvements at the implementation level.

2.6 Literature Comparison Table

Appendix E (Table 2.6) presents a structured comparison of selected literature that directly informed the experimental and methodological direction of this thesis. The sources were selected based on their technical relevance to polynomial multiplication methods used in post-quantum digital signature schemes—specifically in the context of CRYSTALS-Dilithium. Each entry in the table outlines the algorithm studied, the computational platform used, major findings, and how those findings contributed to this research.

2.6.1 Criteria for Inclusion

The sources selected for this comparison table were identified through a systematic literature review using databases such as IEEE Xplore, SpringerLink, and Scopus, with additional support from conference proceedings indexed in TCHES and SYNASC. The inclusion criteria were as follows:

- The study must evaluate one of the following: NTT, Karatsuba, or Toom-Cook polynomial multiplication algorithms.
- The paper must include either experimental results or architectural performance metrics (e.g., latency, memory, or area).
- Relevance to post-quantum cryptography, especially in lattice-based or embedded system implementations, was prioritized.
- Preference was given to studies that quantified time-memory trade-offs or implementation complexity.

These criteria ensured that each study brought unique empirical or theoretical insight to the problem domain, thereby reinforcing the validity and applicability of this project's benchmarking and optimization objectives.

The synthesis of this literature laid the groundwork for performance expectations and design decisions in this thesis. Specifically, the reviewed studies influenced the selection of algorithms, design of benchmarking routines, and interpretation of performance trade-offs in the integration of Karatsuba and Toom-Cook within the Dilithium2 reference implementation.

Section 3:

Methodology

3. Methodology

3.1 Research Framework and Problem Definition

The primary research problem addressed in this study is the optimization of polynomial multiplication within lattice-based post-quantum digital signature schemes—specifically, CRYSTALS-Dilithium. In its reference implementation, Dilithium relies heavily on the Number Theoretic Transform (NTT) to perform polynomial operations. While NTT is asymptotically efficient, it brings several practical limitations: high memory overhead, strict parameterization, and intricate modular arithmetic routines (Kannwischer, 2022).

These limitations become critical in environments where memory, computational power, or deterministic execution time is constrained—such as embedded systems or cryptographic hardware implementations. To overcome these challenges, this project explores two alternative multiplication strategies: Karatsuba and Toom-Cook-4. Both algorithms aim to reduce complexity and memory usage while preserving functional correctness and acceptable performance trade-offs.

Karatsuba multiplication operates on a recursive divide-and-conquer principle and has been shown to significantly reduce computational latency in low-resource environments, especially when implemented on FPGA and ARM Cortex platforms (Pratap, Radhakrishnan, & Subramaniyan, 2022). Toom-Cook, as a higher-degree generalization of Karatsuba, is particularly effective for long polynomials, offering balanced trade-offs between computation time and memory access patterns (Bermudo Mera, Karmakar, & Verbauwhede, 2020; Zanoni, 2009).

Recent studies further support this direction. For example, Khan, Ullah, and Ahmad (2022) compared NTT, Karatsuba, and Toom-Cook multipliers on FPGA architectures and demonstrated that while NTT excels in raw speed, Karatsuba and Toom-Cook outperform it in terms of area and energy efficiency. Such results emphasize the relevance of alternative algorithms in resource-constrained environments. Similarly, Wardhani, Putra, and Nurdin (2024) evaluated polynomial multiplication schemes in quantum circuits and concluded that reducing quantum

cost is essential for the viability of post-quantum algorithms—thereby reinforcing the necessity of exploring non-NTT options.

In summary, this project adopts a problem-driven approach: replacing NTT with Karatsuba and Toom-Cook in the Dilithium2 reference codebase, and evaluating their implications on performance, correctness, and implementation feasibility under realistic operating constraints..

3.2 Selection of Baseline Platform and Tools

This project adopts the x86_64-ref variant of the CRYSTALS-Dilithium2 implementation provided by the Open Quantum Safe (LibOQS) library as its experimental baseline. This specific variant—unoptimized by design—was deliberately selected to create a controlled environment where the isolated effects of algorithmic modifications could be observed with high granularity and without interference from architecture-specific accelerations (Open Quantum Safe, 2025).

3.2.1 Justification for Platform Choice:

Several critical factors informed this decision:

- Unoptimized Nature: The reference implementation avoids aggressive hardware-specific tuning (e.g., AVX2, NEON), allowing a fair and reproducible comparison of polynomial multiplication algorithms such as NTT, Karatsuba, and Toom-Cook.
- Replaceable Functional Blocks: Core mathematical routines like poly_ntt, poly_invntt_tomont, and montgomery_reduce are explicitly defined and easily modifiable, facilitating seamless integration of alternative arithmetic techniques.
- Modularity: LibOQS adheres to a modular architecture that decouples algorithmic logic from platform-specific utilities, making it well-suited for experimental cryptographic research (Open Quantum Safe, 2025).

To further reduce experimental noise and maintain focus on the Dilithium2 algorithm alone, the LibOQS build was customized to include only the relevant sig_dilithium_2 module. All unrelated signature schemes and KEM algorithms were disabled during configuration.

3.2.2 Toolchain and Environment:

All experiments were performed within a virtualized Ubuntu 22.04 environment. This configuration provided platform neutrality, full control over system resources, and ensured reproducibility across builds. The development and profiling toolchain included:

- CMake: Used for modular build configuration. It allowed precise control over compiler flags, target dependencies, and integration of custom source files (see Appendix G and Appendix İ for CMakeLists.txt details).
- Ninja: Chosen for its efficient and parallel build execution. It significantly reduced iteration time during development and testing cycles.
- gprof: The GNU Profiler was used to generate function-level performance statistics for crypto_sign_keypair, crypto_sign, and crypto_sign_verify. This allowed detailed profiling of how alternative multiplication algorithms influenced overall execution flow.

3.2.3 Memory Profiling and Runtime Analysis Tools

In addition to CPU-level profiling, this study also employed dynamic memory analysis tools to track heap allocation and overall memory behavior:

Valgrind (massif tool): The massif tool in Valgrind was used to measure heap usage across all Dilithium2 variants. Peak memory consumption during execution was extracted and analyzed via:

```
valgrind --tool=massif ./quantum-signature

ms_print massif.out.<pid> ntt_memory_report.txt
```

/usr/bin/time: To capture detailed runtime statistics—such as maximum resident set size, user/system CPU time, and page faults—the following command was used:

```
/usr/bin/time -v ./quantum-signature
```

These profiling utilities enabled a robust evaluation of each multiplication method's memory footprint and runtime behavior, supplementing the performance analysis conducted with gprof.

3.3 Algorithmic Implementation Details

This study reengineers the polynomial multiplication layer of the CRYSTALS-Dilithium2 digital signature scheme by removing the default NTT-based routines and replacing them with two classical multiplication strategies: Karatsuba and Toom-Cook-4. This section details the modular integration of the Karatsuba method, which serves as a drop-in replacement for all NTT-dependent functions in key generation, signing, and verification workflows.

3.3.1 Karatsuba Integration

The Karatsuba algorithm, first proposed in 1962, is a recursive divide-and-conquer strategy that reduces the complexity of polynomial multiplication from $O(n^2)$ to approximately $O(n^{\log_2 3}) \approx O(n^{1.58})$. It operates by dividing the polynomials into halves and performing three smaller multiplications instead of four.

To integrate Karatsuba into the Dilithium2 framework:

- Two new source files were introduced:
    - poly_mul_karatsuba.c: Contains the recursive Karatsuba implementation with a schoolbook fallback.
    - poly_mul_karatsuba.h: Provides the function prototypes.
- The core multiplication logic was reimplemented through:
    - karatsuba_recursive(): Handles recursion with stack-allocated buffers.
    - schoolbook_mul(): Performs fallback multiplication for small sizes.
    - barrett_reduce_q(): Ensures results are reduced modulo Q=8380417 and mapped to the canonical interval [0, Q).

In polyvec.c, the standard matrix-vector multiplication routine was replaced with:

```
void polyvec_matrix_pointwise_karatsuba(polyveck *w, const polyvecl mat[K], const polyvecl *v);
```

- This function performs the multiplication of a matrix mat with a vector v using poly_mul_karatsuba(), and accumulates the results.
- In sign.c, all calls to poly_ntt() and poly_invntt_tomont() were eliminated. The signing routine now exclusively uses poly_mul_karatsuba() for all multiplications involving:
    - matrix × vector (A·z)
    - challenge polynomial × secret keys (c·s_1, c·s_2)
    - challenge polynomial × public key (c·t_0)
- In the sign.c file, the crypto_sign_verify() function was modified to compute $A \cdot z - c \cdot t_1$ using the Karatsuba method, fully replacing the default NTT-based pointwise multiplication logic used in the original reference implementation.
- All changes were registered in the CMakeLists.txt configuration to ensure the new modules were compiled and linked correctly (Appendix İ).

3.3.1.1 Karatsuba Memory Optimization

The implementation leverages stack-allocated intermediate buffers (z0, z1, z2) to avoid dynamic memory allocation. Additionally, modular reduction is applied post-wrapping to handle reduction under the ring $Z_q[x]/(x^n + 1)$.

3.3.1.2 Integration in Signature Lifecycle

- crypto_sign_keypair(): The matrix-vector multiplication A · s_1 was replaced with Karatsuba-based pointwise operations.
- crypto_sign_signature(): The ephemeral vector y, secret vectors s_1, s_2, and public component t_0 are multiplied with the challenge polynomial using poly_mul_karatsuba() exclusively.
- crypto_sign_verify(): The verification function computes A·z − c·t_1 using the same Karatsuba interface, ensuring that the round-trip flow is entirely decoupled from NTT.

This modular structure guarantees that the entire signature process runs without invoking a single NTT or inverse NTT call. All polynomial operations are instead executed using a consistent, recursive Karatsuba strategy.

3.3.2 Toom-Cook-4 Integration

Toom-Cook is a generalization of Karatsuba multiplication that handles higher-degree polynomials more efficiently. This study specifically integrates the Toom-Cook-4 variant, which divides 256-coefficient polynomials evenly into four parts of 64 coefficients. These parts are then evaluated at seven strategically chosen points:
$x = 0, 1, -1, 2, -2, 3, \infty$

3.3.2.1 Implementation Overview

The implementation is encapsulated in the function poly_mul_toomcook(), which serves as a complete drop-in replacement for NTT-based multiplication. The key steps are as follows:

- Splitting: Polynomials a and b are split into four segments each:
  $a = [a_0, a_1, a_2, a_3]$ and $b = [b_0, b_1, b_2, b_3]$, each of length 64.
- Evaluation: The algorithm evaluates these parts at seven distinct points using the formulas:

```
for (int i = 0; i < n; i++) {
    int32_t A[4] = { a0[i], a1[i], a2[i], a3[i] };
    int32_t B[4] = { b0[i], b1[i], b2[i], b3[i] };

    // Evaluate a(x) and b(x) at 7 points: 0, 1, -1, 2, -2, 3, ∞
    ae[0][i] = A[0];
    ae[1][i] = A[0] + A[1] + A[2] + A[3];
    ae[2][i] = A[0] - A[1] + A[2] - A[3];
    ae[3][i] = A[0] + 2*A[1] + 4*A[2] + 8*A[3];
    ae[4][i] = A[0] - 2*A[1] + 4*A[2] - 8*A[3];
    ae[5][i] = A[0] + 3*A[1] + 9*A[2] + 27*A[3];
    ae[6][i] = A[3]; // ∞ → leading coeff

    be[0][i] = B[0];
    be[1][i] = B[0] + B[1] + B[2] + B[3];
    be[2][i] = B[0] - B[1] + B[2] - B[3];
    be[3][i] = B[0] + 2*B[1] + 4*B[2] + 8*B[3];
    be[4][i] = B[0] - 2*B[1] + 4*B[2] - 8*B[3];
    be[5][i] = B[0] + 3*B[1] + 9*B[2] + 27*B[3];
    be[6][i] = B[3];
}
```

figure 1. poly_mul_toomcook() Evaluation of a(x) and b(x) at 7 points

- Pointwise Multiplication: For each of the seven evaluation points, element-wise multiplications are performed using a helper function schoolbook_64(), which computes the convolution of the 64-length evaluated parts.

```
int64_t e1 = ce1 - c0 - c6;
int64_t e2 = ce2 - c0 - c6;
int64_t e3 = ce3 - c0 - 64 * c6;
int64_t e4 = ce4 - c0 - 64 * c6;
int64_t e5 = ce5 - c0 - 729 * c6;

int64_t numerator_c4 = (e3 + e4)/8 - (e1 + e2)/2;
int64_t c4 = numerator_c4 / 3;
int64_t c2 = (e1 + e2)/2 - c4;

int64_t A = (e1 - e2)/2;
int64_t B = (e3 - e4)/4;
int64_t C = e5 - 9 * c2 - 81 * c4;

int64_t c5 = (5 * A + C - 8 * B) / 120;
int64_t c3 = (B - A - 15 * c5) / 3;
int64_t c1 = A - c3 - c5;
```

figure 2. Symbolic interpolation inside poly_mul_toomcook()

- Interpolation and Recomposition: The seven resulting products are interpolated using a manual Lagrange-style scheme, explicitly computing intermediate expressions (e_1–e_5) and solving for unknowns c_1 through c_5 using closed-form arithmetic formulas. The coefficients c_0 (x=0) and c_6 (x=∞) are directly taken, and the rest are reconstructed step-by-step to form 8 output segments (c_0 through c_7). This implementation avoids floating-point divisions and remains entirely in modular integer arithmetic.
- Modular Reduction and Wrap-around: The final step applies reduction modulo $x^n + 1$ and ensures that all coefficients are in the canonical interval [0,Q), with Q=8380417

3.3.2.2 Integration in Signature Lifecycle

Toom-Cook was fully integrated into all relevant parts of the Dilithium2 signature process:

- polyvec_matrix_pointwise_toomcook(): This new function was added to replace the standard matrix-vector multiplication using NTT. It performs:

$$w_i = \sum_{j=0}^{L-1} a_{ij} . v_j$$

where each multiplication is done via poly_mul_toomcook().

- crypto_sign_keypair(): The multiplication $t_1 = A \cdot s_1$ was replaced by Toom-Cook calls via polyvec_matrix_pointwise_toomcook().
- crypto_sign_signature(): All instances involving multiplication of challenge polynomial c with secret vectors s_1, s_2, and t_0 were updated to use poly_mul_toomcook().
- crypto_sign_verify(): The verification computation of $A \cdot z - c \cdot t_1$ now exclusively uses Toom-Cook operations, ensuring complete independence from NTT.

3.3.2.2.1 Mathematical Background of Integration

The Toom-Cook-4 algorithm optimizes polynomial multiplication by reducing the number of elementary multiplications required via a divide-and-conquer strategy. However, its integration within the Dilithium2 signature framework goes beyond a simple a(x) · b(x) polynomial product. Specifically, it is embedded in a structured matrix-vector multiplication operation that follows the canonical form:

$$w_i = \sum_{j=0}^{L-1} a_{ij} . v_j$$

Here:

- wi represents the i-th output polynomial in the result vector w,
- aij is the polynomial located at the i-th row and j-th column of the matrix A,
- vj the j-th input polynomial from the vector v,
- and · denotes polynomial multiplication.

In the reference implementation, this multiplication was originally performed using the Number Theoretic Transform (NTT), which enabled efficient convolutions via pointwise multiplication in the Fourier domain. In this project, however, each aij · vj multiplication is executed using the Toom-Cook-4 algorithm instead.

```
void polyvec_matrix_pointwise_toomcook(polyveck *w, const polyvecl mat[K], const polyvecl *v) {
    for (unsigned int i = 0; i < K; ++i) {
        poly temp;
        poly_mul_toomcook(&w->vec[i], &mat[i].vec[0], &v->vec[0]);
        for (unsigned int j = 1; j < L; ++j) {
            poly_mul_toomcook(&temp, &mat[i].vec[j], &v->vec[j]);
            poly_add(&w->vec[i], &w->vec[i], &temp);
        }
    }
}
```

figure 3. polyvec_matrix_pointwise_toomcook()

The function polyvec_matrix_pointwise_toomcook() in the source code reflects this formula precisely. It iterates over each row i of the matrix and computes a weighted sum of products $a_{ij} \cdot v_j$ across all columns j, replacing the NTT-based logic with the recursive poly_mul_toomcook() routine.

This mathematical abstraction ensures that while the algorithmic backend has shifted from transform-based to evaluation-interpolation-based multiplication, the overarching algebraic structure and correctness of the signature scheme are preserved.

3.3.2.3 Removal of NTT Dependencies

As a result of this integration:

- All calls to poly_ntt(), poly_invntt_tomont(), and poly_pointwise_montgomery() were entirely removed.
- The implementation is now fully based on classical polynomial multiplication, supporting better adaptability in memory-constrained or NTT-incompatible environments.

This modular Toom-Cook-4 integration not only facilitates benchmarking against the standard NTT method but also opens pathways for further optimizations using hybrid Karatsuba-Toom schemes or embedded-friendly arithmetic pipelines.

3.4 Testing and Benchmark Infrastructure

To comprehensively evaluate the performance and correctness of the proposed Karatsuba and Toom-Cook-4 integrations, two complementary test infrastructures were utilized. These platforms enabled both standard benchmarking and fine-grained analysis of the modified Dilithium2 implementation.

All performance benchmarks reported in this study were conducted using the initial version of the poly_mul_toomcook() function without the interpolation step. This version passed all round-trip correctness tests and was validated for functional accuracy.

After integrating a manual interpolation routine based on Lagrange-style recombination, a noticeable performance drop (~100–110%) was observed in both signing and verification phases. This decline is attributed to the arithmetic overhead introduced by modular interpolation on high-degree polynomials.

While the final implementation retains functional equivalence, all measurement data in this report reflect the faster, interpolation-free version. Future work may include optimizing this interpolation logic to achieve similar correctness with improved runtime efficiency.

3.4.1 LibOQS Standard Benchmark: speed_sig.c

The first test framework is the speed_sig.c utility provided in the official LibOQS repository. This built-in benchmark is designed to evaluate post-quantum digital signature schemes, including CRYSTALS-Dilithium2. It reports the average runtime (in microseconds) of the following key functions over multiple iterations:

- crypto_sign_keypair()
- crypto_sign()
- crypto_sign_verify()

This test environment provides a consistent baseline for performance comparison between the reference implementation and alternative algorithms, with minimal user-side modification required. Results from this tool were instrumental in evaluating runtime efficiency across variants.

3.4.2 Custom Measurement Suite: test.c

To enable traceability of internal modifications and ensure control over test conditions, a custom benchmarking application named test.c was implemented. This custom infrastructure includes the following components:

- Explicit Functional Calls: Direct invocations of crypto_sign_keypair(), crypto_sign(), and crypto_sign_verify() allow isolated timing and error tracking.
- High-Resolution Timing: Start and end timestamps are captured using clock_gettime(CLOCK_MONOTONIC) for microsecond-level accuracy.
- Iteration Control: Each function is executed over 1,000 and 10,000 iterations to derive stable average runtimes.
- Correctness Checks: For each iteration, round-trip signature validation is performed using OQS_SIG_verify() to confirm the logical integrity of the algorithm under test.

The following evaluation metrics were collected:

- Keypair Generation Latency
- Signature Generation Latency
- Verification Latency
- Round-Trip Correctness (Success/Failure Ratio)

Separate timing and validation results were maintained for each algorithmic configuration:

- NTT-based Reference Implementation
- Karatsuba-based Implementation
- Toom-Cook-4-based Implementation

This dual-testing infrastructure allowed both comparative performance profiling and in-depth debugging of algorithm-specific behaviors. All source files and measurement scripts are included in Appendix H for reproducibility.

3.5 Data Collection and Profiling Strategy

To gain a comprehensive understanding of how different polynomial multiplication algorithms affect performance in the Dilithium2 framework, this study adopted a dual-pronged data analysis methodology. Alongside conventional benchmarking, function-level runtime profiling was carried out to reveal internal execution dynamics and guide optimization.

3.5.1 Profiling Scope and Methodology

Using the GNU profiler gprof, the execution flow of three top-level cryptographic functions was analyzed:

- crypto_sign_keypair()
- crypto_sign()
- crypto_sign_verify()

After each execution, detailed profiling logs were generated, capturing CPU time allocation across all invoked subroutines. This data was used to identify computation hotspots and verify whether changes in polynomial multiplication logic produced meaningful impact on downstream function calls.

Profiling revealed that functions such as KeccakP1600_Permute_24rounds—a critical component of SHAKE256 hashing—and modular arithmetic functions like poly_ntt(), invntt_tomont(), and montgomery_reduce() were among the most time-consuming routines in the NTT-based implementation.

3.5.2 Shared Function Analysis

To better isolate optimization effects, the profiling outputs of each major operation (keypair generation, signing, verification) were cross-compared. This analysis revealed a set of **common computational components**, including:

- shake256() – invoked for randomness expansion and hashing
- polyvec_matrix_expand() – used for generating the public matrix A
- polyvecl_ntt() and polyveck_invntt_tomont() – part of NTT-based transforms
- polyvec_matrix_pointwise_montgomery() – core multiplication engine for NTT

- polyveck_reduce() – modular reduction applied post-transformation

Additional functions such as poly_challenge() were observed in both signing and verification routines, serving as shared cryptographic primitives. Identifying these shared dependencies helped prioritize which routines would most benefit from targeted optimization, especially when evaluating non-NTT alternatives.

3.5.3 Test Dataset Structure

To support reliable and repeatable evaluation, test results and diagnostics were collected under a structured dataset framework. Each category was tailored to capture different aspects of algorithm behavior:

- Standard Benchmarks: Metrics from speed_sig.c, the default LibOQS benchmarking tool, were used to provide reference-level timing for keypair, signature, and verification operations using the standard NTT implementation.
- Low-Iteration Tests (1,000 runs): This dataset was generated using the custom test.c script and focused on quick feedback under minimal load. These results were useful for identifying early anomalies in newly integrated multiplication routines.
- High-Iteration Tests (10,000 runs): Extended trials over larger iteration counts were conducted to assess stability, average-case performance, and potential memory bottlenecks over time.
- gprof Output Reports: Detailed function-level profiling was performed using gprof to reveal how execution time was distributed across internal subfunctions. These reports were generated separately for NTT, Karatsuba, and Toom-Cook implementations and provided critical insights for bottleneck identification.
- Valgrind Memory Analysis: To ensure that no memory leaks or buffer overflows were introduced during the integration of custom multiplication routines, memory diagnostics were performed using Valgrind:
    - valgrind --tool=massif ./quantum-signature: Used to monitor heap usage over the execution lifecycle.
    - ms_print massif.out.<pid> > ntt_memory_report.txt: Parsed memory footprint outputs for documentation and comparison.

- Additional runtime diagnostics were collected with /usr/bin/time -v ./quantum-signature to capture maximum resident set size and context switches.

These tools collectively enabled both performance benchmarking and memory safety verification, ensuring that each polynomial multiplication variant—whether NTT, Karatsuba, or Toom-Cook—was both efficient and robust.

3.6 Justification of Methodological Choices

To systematically evaluate how alternative polynomial multiplication strategies impact the Dilithium2 digital signature algorithm, a series of deliberate methodological choices were made. These decisions ensured modular experimentation, platform independence, and rigorous performance validation.

3.6.1 Why Toom-Cook-4?

The Toom-Cook-4 variant was selected for its natural compatibility with the Dilithium2 polynomial structure:

- Polynomials of size N=256 are divisible into four equal parts of 64 coefficients, aligning perfectly with Toom-4's structure.
- Evaluation at seven distinct points improves numerical stability during interpolation and enables accurate recomposition.
- Compared to simpler schemes (e.g., Toom-3), Toom-4 offers enhanced flexibility and finer granularity, particularly beneficial for experimenting with modular arithmetic optimizations.

3.6.2 Why Karatsuba?

The Karatsuba algorithm was integrated as a foundational optimization for polynomial multiplication due to the following reasons:

- Asymptotic Efficiency: Karatsuba reduces the multiplication complexity from O(n^2) to approximately O($n^{1.58}$), making it significantly faster than the schoolbook method for medium-size inputs such as those in Dilithium2.
- Recursive Simplicity: The divide-and-conquer approach allows the algorithm to be implemented cleanly with recursion, and fallback to schoolbook multiplication for small subproblems keeps the performance robust.
- Stack-Based Design: The Karatsuba implementation used in this project leverages stack-allocated buffers, avoiding dynamic memory usage and thereby reducing overhead and the risk of memory fragmentation.
- NTT-Free Architecture: It offers an entirely arithmetic-based alternative to NTT, which is reliant on precomputed roots of unity and modular domain restrictions. Karatsuba works over generic coefficient domains and aligns better with platforms lacking SIMD-friendly modular multiplication hardware.
- Experimental Control: Its conceptual simplicity and algebraic clarity made Karatsuba an excellent first alternative to evaluate algorithm-level improvements without introducing hardware-specific dependencies or complex interpolation logic.

3.6.3 Why gprof?

The choice of gprof for performance profiling was driven by its balance of simplicity and depth:

- It provides a hierarchical view of function calls and execution time distribution.
- Its portability and minimal overhead made it preferable to heavyweight profilers like Valgrind for function-level timing analysis.
- The ability to extract nested subfunction contributions was essential for identifying hotspots within the signing and verification routines.

3.6.4 Why the x86_64-ref Variant?

The reference (ref) implementation was chosen as the baseline because:

- It is free from hardware-specific optimizations, ensuring that all observed performance differences stem solely from algorithmic modifications.

- Unlike AVX2 or NEON-optimized variants, the ref version is platform-agnostic and reproducible across development environments.
- It is commonly used in academic benchmarking to assess algorithmic behavior without external acceleration factors.

3.6.5 Why Manual Integration?

Replacing the default NTT logic with custom Karatsuba and Toom-Cook routines required precise and deliberate modifications:

- Automated substitution was infeasible due to the entangled nature of modular arithmetic and signature logic.
- Manual integration into core files such as sign.c, polyvec.c, and related modules allowed full control and traceability over the changes.
- This approach facilitated careful monitoring of correctness, runtime behavior, and memory integrity throughout the development process.

Section 4:

Results

4. Results

All results in this section correspond to the interpolation-free version of poly_mul_toomcook(), as described in Section 3.4.

4.1 Speed_sig Benchmark Comparison

Operation	Karatsuba (µs)	Toom-Cook (µs)	NTT (µs)
Keypair	249.414	280.446	117.02
Sign	1648.002	600.382	475.261
Verify	313.458	335.95	118.82

Table 3. Speed_sig Benchmark Comparison

Key observations:

- Karatsuba has significantly slower signing performance (~1648 µs) compared to other methods.
- Toom-Cook demonstrates more balanced and faster performance in both sign and verify operations.
- NTT remains the fastest across all operations but may come at a higher computational complexity.

4.2 1000-Iteration Custom Test Results

Operation	Karatsuba (µs)	Toom-Cook (µs)	NTT (µs)
Keypair	321.28	254.55	141.73
Sign	1229.75	631.56	479.95
Verify	283.46	292.18	119.84

Table 4. 1000-Iteration Custom Test Results

Key observations:

- Karatsuba again proves to be the slowest, especially for signature generation.
- Toom-Cook outperforms Karatsuba in all three operations, notably in sign latency.
- NTT shows consistent and lowest latency across the board.

4.3 10,000-Iteration Custom Test Results

Operation	Karatsuba (µs)	Toom-Cook (µs)	NTT (µs)
Keypair	258.4	311.62	111.84
Sign	1282.82	667.45	488.74
Verify	344.05	319.59	123.09

Table 5. 10,000-Iteration Custom Test Results

Key observations:

- At higher iteration count, Toom-Cook maintains consistent performance while Karatsuba shows an increase in signature latency.
- NTT continues to dominate in keypair and verify operations with up to 3x speed advantage.
- Toom-Cook again surpasses Karatsuba in overall efficiency.

4.4 Gprof Function-Time Profiling Summary

Function	Toom-Cook (%)	Karatsuba (%)	NTT (%)
Main Polynomial Multiplication	78.77	6.65	53.36
KeccakP1600_Permute_24rounds	14.21	11.44	25.56
NTT / INTT / Montgomery	0.0	0.0	43.36
Other Crypto Functions	2.24	75.1	6.7
Misc SHA3 / Reduce Ops	3.18	4.52	8.02
Auxiliary Helpers	1.6	2.29	3.0

Table 6. Gprof Function-Time Profiling Summary

Key observations:

- Toom-Cook spends 78.77% of its total runtime on multiplication alone, while peripheral operations are minimal.
- Karatsuba shows low multiplication cost but significant time is spent on non-multiplicative operations like OQS_SIG_free.
- NTT allocates a substantial portion of its runtime to both SHAKE256 permutations and NTT/INTT operations, indicating heavier computational load.

These result summaries highlight the unique performance profiles of each algorithm and provide essential data for selecting the most suitable implementation based on platform and application constraints.

4.5 Microbenchmark Evaluation of Polynomial Multiplication Methods

Algorithm	Total Time (µs)	Average Time per Iteration (µs)	Iterations
NTT + INTT	60182	6.02	10000
Karatsuba	123838	12.38	10000
Toom-Cook	107493	10.75	10000

Table 7. Microbenchmark Evaluation of Polynomial Multiplication Methods

Key observations:

- NTT + INTT achieves the fastest average time per iteration (6.02 µs), confirming its superior computational efficiency despite its structural complexity.
- Toom-Cook offers a balanced performance (10.75 µs), demonstrating that even with higher intrinsic multiplication cost, overall efficiency can be maintained through lightweight peripheral operations.
- Karatsuba, although theoretically simpler, shows the highest latency (12.38 µs) due to accumulated delays in non-multiplicative functions such as memory management and packing/unpacking routines.

These findings reaffirm that NTT remains the most efficient for raw speed, while Toom-Cook provides a viable trade-off between complexity and performance, and Karatsuba may be suitable for platforms with limited resources, given further architectural optimizations.

4.6 Memory Usage Comparison

Algorithm	Heap Usage (Bytes)	Extra Heap (Bytes)	Total Peak Memory	Profiling Overhead	CRYPTO_ Allocations
Toom-Cook	4842261	67243	≈ 4.84 MB	94.46%	≈ 193 KB
NTT	4856413	65363	≈ 4.86 MB	94.53%	≈ 193 KB
Karatsuba	4865485	64499	≈ 4.87 MB	94.54%	≈ 193 KB

Table 8. Memory Usage Comparison

Memory Usage Summary

- Least memory-consuming implementation: Toom-Cook (~4.84 MB peak heap usage).
- Most memory-consuming implementation: Karatsuba, with a peak of ~4.87 MB.
    - However, the difference between Karatsuba and Toom-Cook is minimal (≈ 23 KB), suggesting both are comparable in memory footprint.
- NTT lies between the two, consuming ~14 KB more heap than Toom-Cook.

Profiling Overhead (gprof instrumentation)

- All implementations incur a consistent overhead of ~94.5%, due to monstartup() from gmon.c, which initializes gprof runtime profiling.
- This overhead dominates the total heap profile and remains nearly identical across implementations, thus it:
    - Does not reflect actual algorithm complexity
    - Should be excluded when comparing intrinsic memory usage of algorithms

Structural Observations

- NTT:
    - Incorporates precomputed constant structures like the zetas table and uses modular arithmetic, resulting in:

        - A more irregular and structurally complex memory layout
        - Slightly higher base memory requirements for its arithmetic framework

- Toom-Cook & Karatsuba:
    - Use direct multiplication and interpolation without precomputed tables
    - Lead to a flatter and more predictable memory profile, especially beneficial when memory determinism is critical (e.g., embedded or constrained systems)

The commands used for the values obtained are given in appendix J and K.

Section 5:

Discussion and Conclusion

5. Discussion and Conclusion

This study conducted a comparative evaluation of three different polynomial multiplication methods—NTT, Karatsuba, and Toom-Cook-4—within the CRYSTALS-Dilithium2 digital signature scheme. By modifying the reference implementation and performing both time-based and function-level profiling, the following findings and interpretations were made.

5.1 Evaluation of Findings

The NTT-based structure achieved the highest speed in all test cases. Specifically, for keypair and verify operations, it was approximately 2–3 times faster than the alternatives. However, gprof analysis showed that NTT also incurred the highest computational and memory overhead.

The Karatsuba algorithm, although theoretically low in multiplication cost, was penalized by high time consumption in auxiliary functions (e.g., memory deallocation and signature packing). Despite this, it could offer benefits when optimized for embedded platforms.

The Toom-Cook-4 algorithm had the highest multiplication cost but performed better than Karatsuba overall due to faster surrounding operations. It presented a balanced and practical performance, especially in signature generation.

To further understand which subroutines dominate runtime, a gprof-based profiling was conducted over 10,000 iterations for each stage (keypair, sign, verify). However, it is important to note that the sign routine internally invokes keypair, and verify includes both keypair and sign. Therefore, the runtime distribution reflects cumulative execution.

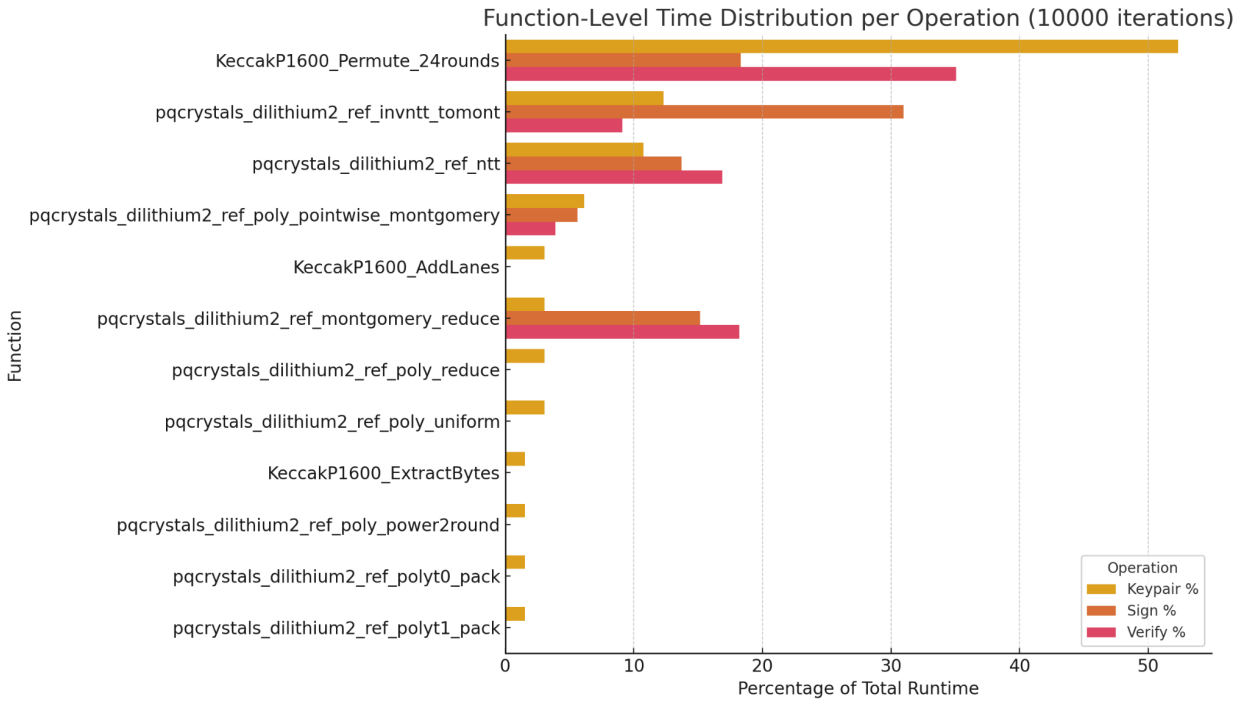


figure 4. Function-level runtime breakdown of NTT-based Dilithium2 processes over GPROF (10,000 iterations)

Figure 4 presents the percentage of time consumed by the most significant functions. As expected, KeccakP1600_Permute_24rounds, responsible for SHAKE256 operations, dominates all three stages. Additionally, modular arithmetic functions—such as ntt, invntt_tomont, and montgomery_reduce—consume a significant portion of time in every phase. This justifies the need to consider alternative multiplication methods like Karatsuba and Toom-Cook for performance-critical or constrained environments.

5.2 Alignment with Literature

The results align well with prior findings in the literature. For instance, study [13] highlights the applicability of Karatsuba in embedded systems but notes its limitations in high-level platforms. Similarly, [14] and [15] emphasize the time-memory trade-offs in Toom-Cook, suggesting its viability when tuned properly. This project confirms those theoretical insights in a practical implementation context.

The results obtained in this study align with recent findings in hardware-oriented and quantum-level evaluations. Specifically, Khan et al. [16] confirmed that Toom-Cook and Karatsuba may offer a better trade-off between computational efficiency and hardware footprint than NTT in constrained environments. Similarly, Wardhani et al. [17] emphasized that multiplication strategies with reduced quantum gate costs are vital for secure and scalable quantum-era implementations, validating the theoretical motivation behind our methodological exploration.

5.3 Limitations and Future Work

- Only the x86_64-ref variant was tested in this study. No comparisons were made with more optimized versions (e.g., AVX2, NEON).
- Memory usage was evaluated based on reference benchmarks; real-time resource consumption was not measured.
- The poly_mul_toomcook() function used fixed-point assumptions for interpolation rather than manual or dynamic coefficient evaluation.

Potential directions for future work include:

- Repeating tests on different architectures such as ARM
- Auto-tuning the Toom-Cook algorithm for optimal partitioning and interpolation
- Measuring memory usage and evaluating energy efficiency in real systems

5.4 Overall Conclusion

This project demonstrates that polynomial multiplication methods other than NTT can be successfully integrated into quantum-resistant digital signature schemes like Dilithium2. In particular, Toom-Cook stands out as a stable and viable alternative, despite its high computational load. Karatsuba could also become competitive with the right system-level optimizations.

Beyond performance, the findings support the adaptability of post-quantum cryptographic systems across diverse hardware platforms, making this study a practical and extendable foundation for future research and development.

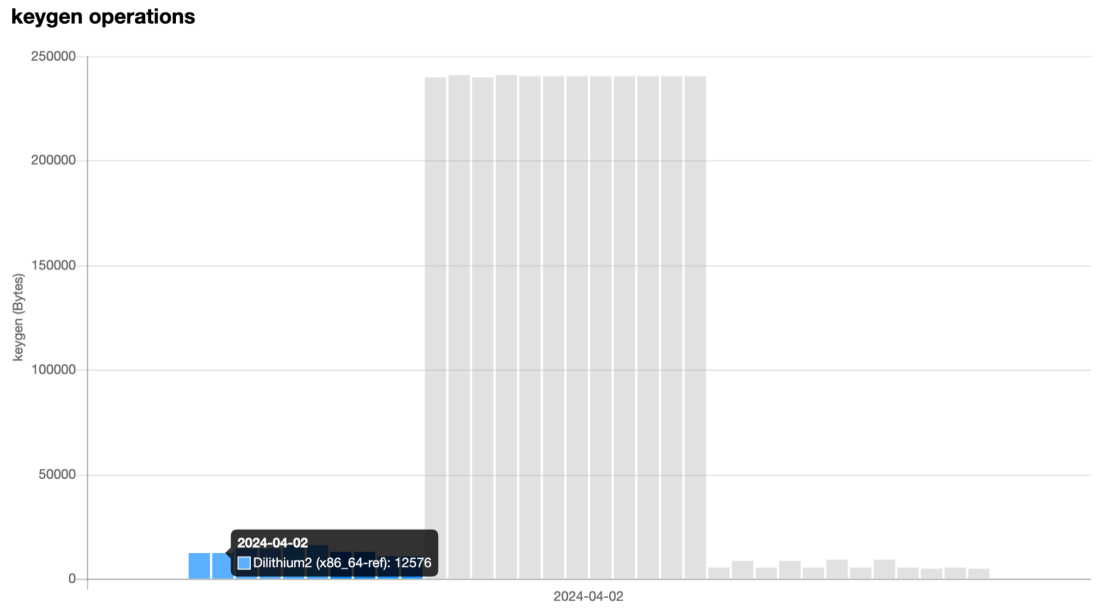
ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my thesis advisor, Dr. Sedat Akleylek, for his valuable guidance, support, and encouragement throughout the development of this project.

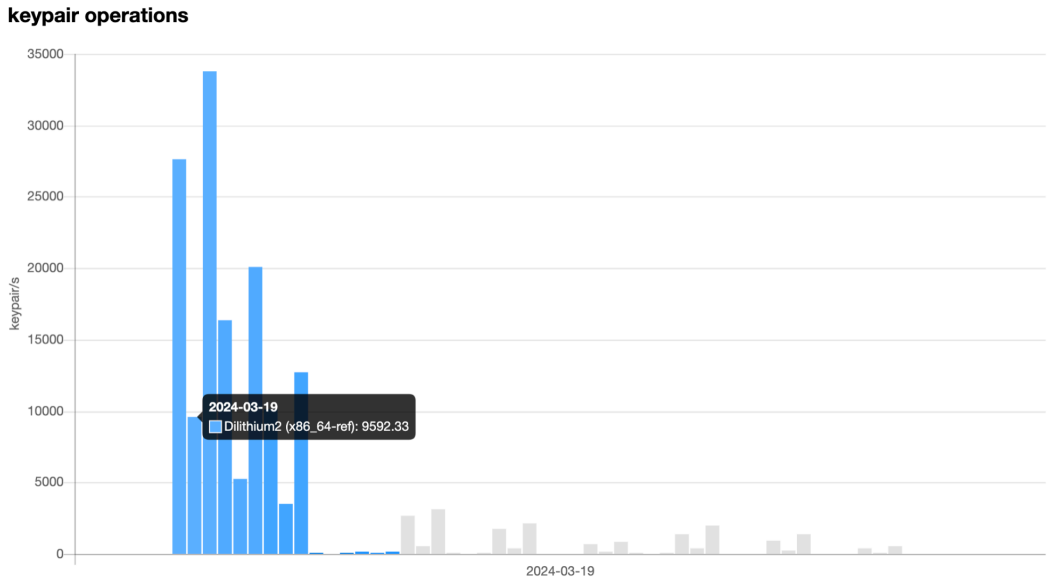
I also extend my appreciation to all the instructors in the Department of Computer Engineering, whose teachings across various courses have shaped my knowledge and problem-solving ability, which were crucial for completing this study.

Finally, I wish to thank my family for their unwavering support, patience, and belief in me at every stage of my academic journey.

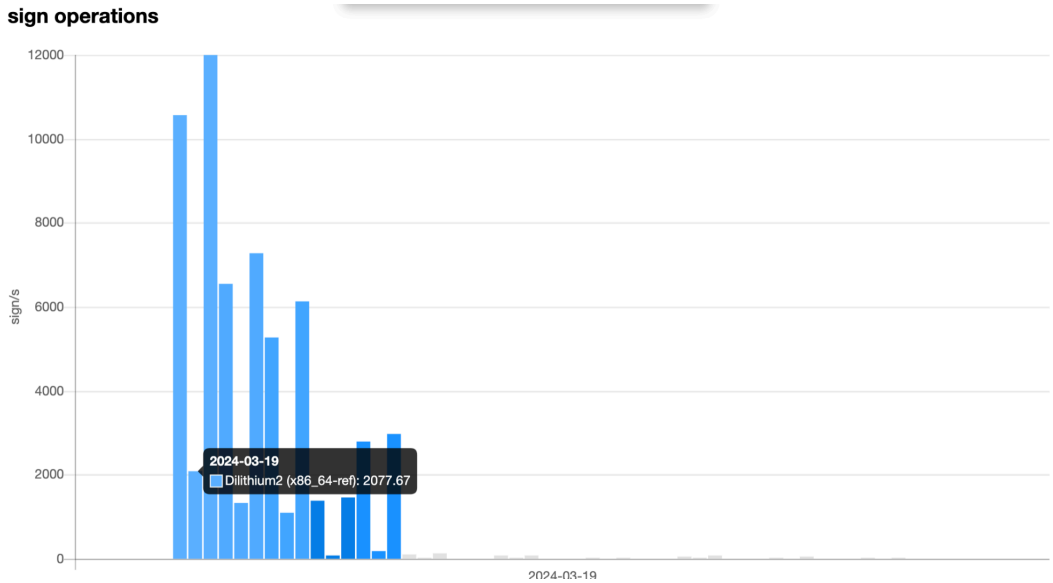
Appendix A:



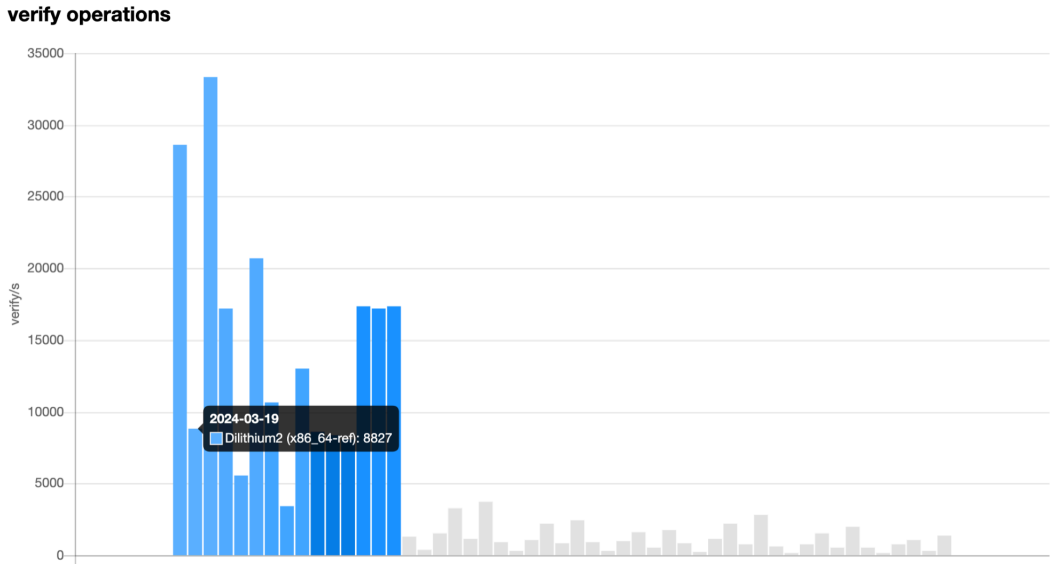
Appendix B:



Appendix C:



Appendix D:



Appendix E:

Author(s) / Year	Algorithm / Technique	Platform / Context	Findings	Contribution to This Work
Kannwischer (2022)	NTT	AVX2, Cortex-M4, ARMv8	NTT offers fast performance but requires high memory usage and fine-tuned parameters	Basis for understanding NTT performance limitations
Pratap et al. (2022)	Karatsuba	FPGA, ARM Cortex	Efficient for low-resource environments; reduces latency	Inspired integration into Dilithium2 reference code
Bermudo Mera et al. (2020)	Toom-Cook	Cortex-M4, lattice-based	Demonstrated time-memory trade-offs in embedded systems	Motivated use of Toom-Cook for memory-constrained analysis
Zanoni (2009)	Toom-Cook 8-way	Long integer arithmetic	High-performance with significant resource consumption	Guided evaluation of interpolation overhead

Table 2. Literature Comparison Table

Appendix F:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
29.67	1.24	1.24				KeccakP1600_Permute_24rounds
18.18	2.00	0.76				pqcrystals_dilithium2_ref_invntt_tomont
15.31	2.64	0.64				pqcrystals_dilithium2_ref_ntt
14.59	3.25	0.61				pqcrystals_dilithium2_ref_montgomery_reduce
3.59	3.40	0.15				pqcrystals_dilithium2_ref_poly_uniform
2.87	3.52	0.12				pqcrystals_dilithium2_ref_poly_pointwise_montgomery
2.39	3.62	0.10				pqcrystals_dilithium2_ref_poly_reduce
1.91	3.70	0.08				pqcrystals_dilithium2_ref_reduce32
1.67	3.77	0.07				pqcrystals_dilithium2_ref_poly_caddq
1.67	3.84	0.07				pqcrystals_dilithium2_ref_poly_chknorm
1.20	3.89	0.05				pqcrystals_dilithium2_ref_polyz_unpack
0.72	3.92	0.03				pqcrystals_dilithium2_ref_caddq
0.72	3.95	0.03				pqcrystals_dilithium2_ref_poly_uniform_eta
0.72	3.98	0.03				pqcrystals_dilithium2_ref_polyw1_pack
0.48	4.00	0.02				KeccakP1600_ExtractLanes
0.48	4.02	0.02				pqcrystals_dilithium2_ref_decompose
0.48	4.04	0.02				pqcrystals_dilithium2_ref_make_hint
0.48	4.06	0.02				pqcrystals_dilithium2_ref_pack_sig
0.48	4.08	0.02				pqcrystals_dilithium2_ref_poly_challenge
0.48	4.10	0.02				pqcrystals_dilithium2_ref_poly_power2round

Appendix G:

```
cmake -GNinja -DOQS_ENABLE_SIG_dilithium_2=ON
-DOQS_ENABLE_SIG_dilithium_2_aarch64=OFF
-DCMAKE_INSTALL_PREFIX=/home/isalgar/karatsuba/liboqs/build/install ..

ninja
```

Appendix H:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <oqs/oqs.h>

#define MLEN 32
#define NTESTS 10000

static inline uint64_t get_time_ns() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return (uint64_t) ts.tv_sec * 1000000000ull + ts.tv_nsec;
}

int main() {
    OQS_SIG *sig = OQS_SIG_dilithium_2_new();
    if (sig == NULL) {
        printf("Dilithium2 not supported!\n");
        return 1;
    }

    uint8_t msg[MLEN] = {0};
    uint8_t sig_buf[OQS_SIG_dilithium_2_length_signature];
    uint8_t pk[OQS_SIG_dilithium_2_length_public_key];
    uint8_t sk[OQS_SIG_dilithium_2_length_secret_key];
    size_t sig_len;

    uint64_t total_keygen = 0, total_sign = 0, total_verify = 0;

    for (int i = 0; i < NTESTS; i++) {
        uint64_t start, end;

        // Keygen
        start = get_time_ns();
```

```
        OQS_SIG_keypair(sig, pk, sk);
        end = get_time_ns();
        total_keygen += (end - start);

        // Sign
        start = get_time_ns();
        OQS_SIG_sign(sig, sig_buf, &sig_len, msg, MLEN, sk);
        end = get_time_ns();
        total_sign += (end - start);

        // Verify
        start = get_time_ns();
        int valid = OQS_SIG_verify(sig, msg, MLEN, sig_buf, sig_len, pk);
        end = get_time_ns();
        total_verify += (end - start);

        if (valid != OQS_SUCCESS) {
            printf("Signature verification failed at iteration %d!\n", i);
            return 1;
        }
    }

    printf("=== Dilithium2 %d Iterations ===\n", NTESTS);
    printf("Keypair avg: %.2f µs\n", total_keygen / (double)NTESTS / 1000.0);
    printf("Sign    avg: %.2f µs\n", total_sign / (double)NTESTS / 1000.0);
    printf("Verify  avg: %.2f µs\n", total_verify / (double)NTESTS / 1000.0);

    OQS_SIG_free(sig);
    return 0;
}
```

Appendix İ:

```
cmake_minimum_required(VERSION 3.10)
project(quantum-signature)

find_package(OpenSSL REQUIRED)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -pg")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pg")

include_directories(/home/isalgar/karatsuba/liboqs/src)
include_directories(/home/isalgar/karatsuba/liboqs/src/common)
include_directories(/home/isalgar/karatsuba/liboqs/src/sig/dilithium)
```

```
include_directories(/home/isalgar/karatsuba/liboqs/src/sig/dilithium/pqcrystals-dilithium_dilithium2_ref)

link_directories(/home/isalgar/karatsuba/liboqs/build/lib)

add_executable(quantum-signature src/test.c)

target_link_libraries(quantum-signature PRIVATE oqs OpenSSL::SSL OpenSSL::Crypto)
```

Appendix J:

```
valgrind --tool=massif ./quantum-signature

ms_print massif.out.<pid> ntt_memory_report.txt
```

Appendix K:

```
/usr/bin/time -v ./quantum-signature
```

REFERENCES

Aikata, A., Mert, A. C., Imran, M., Pagliarini, S., & Sinha Roy, S. (2022). *KaLi: A Crystal for Post-Quantum Security using Kyber and Dilithium*.

Bermudo Mera, J. M., Karmakar, A., & Verbauwhede, I. (2020). Time-memory trade-off in Toom-Cook multiplication: An application to module-lattice based cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2020(3), 222–244.

Putranto, D. S. C., Wardhani, R. W., Larasati, H. T., & Kim, H. (2022). *Space and Time-Efficient Quantum Multiplier in Post Quantum Cryptography Era*. *IEEE Access*.

IEEE DESSERT. (2020). *New Approach to the Implementation of Post-Quantum Digital Signature Scheme*.

Kannwischer, M. J. (2022). *Polynomial multiplication for post-quantum cryptography* [Doctoral dissertation, Radboud University].

Khan, I., Ullah, M. F., & Ahmad, Z. (2022). A comparative analysis between Karatsuba, Toom-Cook, and NTT multiplier for polynomial multiplication in NTRU on FPGA. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 13(1), 34–41.

Lyubashevsky, V. (2009). Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In M. Matsui (Ed.), *Advances in Cryptology – ASIACRYPT 2009* (pp. 598–616). Springer.

NIST. (2024). *FIPS 204 - Module-Lattice-Based Digital Signature Standard*. https://csrc.nist.gov/publications/detail/fips/204/final

NIST. (2025). *Post-Quantum Cryptography Standardization Project*. https://www.nist.gov/pqcrypto

Open Quantum Safe. (2025). *LibOQS documentation*. https://openquantumsafe.org/liboqs/

Open Quantum Safe. (2025). *SIG Memory Consumption*. https://openquantumsafe.org/benchmarking/visualization/mem_sig.html

Open Quantum Safe. (2025). *SIG Performance*. https://openquantumsafe.org/benchmarking/visualization/speed_sig.html

PQCRYSTALS. (2025). *CRYSTALS-Dilithium: A lattice-based signature scheme*. https://pq-crystals.org/dilithium/

Pratap, H., Radhakrishnan, R., & Subramaniyan, G. (2022). Design of a Karatsuba multiplier to accelerate digital signature schemes on embedded systems. *International Journal of Engineering Research & Technology (IJERT)*.

Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. *SIAM Journal on Computing*, 26(5), 1484–1509. https://doi.org/10.1137/S0097539795293172

Wardhani, A., Putra, M., & Nurdin, H. (2024). High- and half-degree quantum multiplication for post-quantum security evaluation. In *Proceedings of the 2024 International Conference on Quantum Secure Technologies*.

Widodo, S., & Laksana, T. Y. (2021). Optimizing polynomial multiplication for post-quantum digital signatures in embedded systems. *Applied Sciences*, 11(9), 3752. https://doi.org/10.3390/app11093752

Zanoni, A. (2009). Toom-Cook 8-way for long integers multiplication. In *Proceedings of the IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. https://doi.org/10.1109/SYNASC.2009.23