IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–29. DOI:XXXXXXXX

ML-DSA Digital Signatures in Resource-Constrained MQTT Environments

Jiahao Xiang[1] and Lang Li[1]

Hengyang Normal University, Hengyang, China

Abstract. Large-scale quantum computers necessitate migration of Internet of Things (IoT) systems to post-quantum cryptographic standards. While NIST has standardized ML-DSA (Module-Lattice-Based Digital Signature Algorithm) for digital signatures, practical deployment of post-quantum authentication in resource-constrained IoT environments remains inadequately characterized. ML-DSA integration within MQTT-based IoT systems is evaluated through comprehensive performance analysis on ARM Cortex-M4 microcontrollers. The methodology encompasses signature generation and verification benchmarking, memory utilization analysis, and protocol overhead assessment under realistic IoT constraints. Performance implications of ML-DSA deployment are examined through comparison with classical signature schemes, quantifying computational overhead, memory requirements, and verification latency on resource-constrained devices. These findings reveal fundamental trade-offs between post-quantum security and IoT performance requirements, providing critical insights for practical deployment strategies in resource-limited environments.

Keywords: Post-Quantum Cryptography · ML-DSA · MQTT Protocol · IoT Security · Resource-Constrained Devices

1 Introduction

The emergence of quantum computing fundamentally undermines current cryptographic infrastructures, necessitating systematic migration to post-quantum cryptographic standards across all computing domains [KMLO19]. The National Institute of Standards and Technology (NIST) has formalized ML-DSA (Module-Lattice-Based Digital Signature Algorithm) within FIPS 204 [NIS24], establishing this CRYSTALS-Dilithium-based scheme as the primary standard for post-quantum digital signatures.

The transition from theoretical post-quantum standardization to practical deployment has revealed fundamental implementation challenges extending beyond algorithmic considerations [Ano24]. Post-quantum signature schemes impose substantial computational and storage overhead compared to classical alternatives. ML-DSA signatures span 2,420-4,627 bytes across security levels, representing 30-70× size increases relative to 64-byte ECDSA signatures. These expanded signature sizes, combined with elevated computational demands, substantially exceed the computational and memory capabilities of resource-constrained devices [HKS24].

Internet of Things (IoT) systems exemplify these deployment challenges, where computational, memory, and energy limitations constrain cryptographic implementation choices [GMS19]. Despite performance overhead, signature-based authentication remains essential for applications requiring cryptographic non-repudiation, audit trails, and public key infrastructure compatibility. The MQTT protocol, widely adopted for IoT messaging due to its lightweight design, experiences performance degradation when post-quantum signatures introduce overhead on resource-constrained devices. This disparity between standardization progress and deployment feasibility motivates systematic performance characterization.

~~This work addresses ML-DSA integration within MQTT-based IoT systems through performance analysis on~~ This work addresses ML-DSA integration within MQTT-based IoT systems through performance analysis on ARM Cortex-M4 microcontrollers. Three contributions advance post-quantum IoT deployment:

- **Computational Performance Benchmarking**: ~~Cycle-accurate measurements of ML-DSA signature oper~~ Cycle-accurate measurements of ML-DSA signature operations on ARM Cortex-M4 microcontrollers at 168 MHz, quantifying execution latency and throughput across all three standardized parameter sets.
- **Memory Utilization Analysis**: ~~Static and dynamic memory profiling measuring Flash storage requiremen~~ Static and dynamic memory profiling measuring Flash storage requirements, stack consumption, and peak SRAM utilization during signature operations.
- **Protocol-Level MQTT Integration Assessment**: ~~End-to-end latency and message size overhead evaluat~~ End-to-end latency and message size overhead evaluation within MQTT publish-subscribe workflows, comparing ML-DSA against ECDSA P-256 baseline.

~~These contributions quantify trade-offs between post-quantum security levels and IoT performance requireme~~ These contributions quantify trade-offs between post-quantum security levels and IoT performance requirements, enabling informed deployment decisions for resource-limited environments.

The remainder of this paper is organized as follows: Section 2 presents background information on post-quantum cryptography and related work in IoT deployments. Section 3 provides an overview of the ML-DSA algorithm and its implementation considerations. Section 4 describes the implementation architecture for MQTT-based IoT systems. Section 5 details the experimental methodology for performance evaluation on ARM Cortex-M4 microcontrollers. Section 6 presents and analyzes experimental results, examining the trade-offs between security and performance. Finally, Section 7 concludes with implications for practical deployment and future research directions.

2 Related Work and Motivation

Conventional network protocols have undergone post-quantum migration analysis [KSD20, SKD20], yet ~~IoT-specific communication protocols remain underexplored~~IoT-specific communication protocols remain underexplored, creating deployment barriers in resource-constrained environments.

2.1 ML-DSA Performance Benchmarks on Embedded Systems

Banegas et al. [BZB+22] benchmarked CRYSTALS-Dilithium on embedded systems, reporting 1.45× computational cycles relative to ECDSA on ARM Cortex-M4 processors. This overhead, combined with memory constraints, creates deployment bottlenecks in IoT environments.

The pqm4 benchmarking campaign [KKPY24] extends these observations to standardized ML-DSA parameter sets, reporting tens of kilobytes memory consumption and millions of CPU cycles per signature on Cortex-M4 targets. Measurements on Cortex-M7 microcontrollers [HW23] demonstrate reduced latency but maintain signature operations in the tens-of-milliseconds regime, positioning ML-DSA at the threshold of acceptable responsiveness for interactive workloads.

2.2 Deployment Bottlenecks in IoT Applications

Practical deployment scenarios reveal performance bottlenecks challenging IoT system viability. Analysis of the SUIT (Software Update for the Internet of Things) framework demonstrates post-quantum signature verification requiring up to 3.2 seconds on low-power microcontrollers, exceeding sub-second latency constraints for real-time IoT applications.

Security vulnerabilities and throughput limitations compound these constraints. Marchsreiter [Mar24] reports order-of-magnitude transaction throughput reductions on embedded blockchain nodes with ML-DSA, where signing latency dominates system throughput. Fault injection research [WYQ+24] achieved 89.5% attack success rates on ARM Cortex-M ML-DSA implementations through electromagnetic fault injection, demonstrating Keccak-based hash function susceptibility to loop-abort faults enabling private key recovery. Such vulnerabilities necessitate countermeasures that further impact performance.

2.3 Alternative Approaches and Limitations

Recent research explores alternative authentication architectures. Kim and Seo [KS25] demonstrate that post-quantum signatures introduce prohibitive MQTT authentication overhead, motivating KEM-based architectures eliminating signature operations. Their CRYSTALS-Kyber implementation achieves 4.32-second handshake completion on 8-bit AVR microcontrollers, though this alternative does not address signature-dependent authentication requirements.

Algorithmic optimization research demonstrates limitations in addressing fundamental resource constraints. Barrett multiplication techniques achieve 1.38–1.51× performance improvements on ARM Cortex-M3 and 6.37–7.27× improvements on 8-bit AVR platforms [HKS25]. However, these optimizations provide insufficient gains to bridge the gap between post-quantum signature requirements and IoT device capabilities, necessitating system-level analysis.

2.4 Research Gap and Motivation

~~Empirical studies evaluating ML-DSA performance within MQTT protocol implementations remain absent~~Empi studies evaluating ML-DSA performance within MQTT protocol implementations remain absent, representing a knowledge gap in post-quantum IoT deployment. This gap is particularly relevant given MQTT's widespread industrial IoT adoption, where signature-based authentication remains mandatory for regulatory compliance and audit requirements.

Existing research focuses on isolated cryptographic operations or alternative protocol architectures, without addressing systematic MQTT integration challenges. ~~This work addresses these limitations thr~~ This work addresses these limitations through ML-DSA performance analysis within realistic MQTT environments, informing practical post-quantum IoT migration strategies.

3 ML-DSA and MQTT Protocol Integration

3.1 ML-DSA Algorithm Characteristics

ML-DSA constitutes NIST's standardized post-quantum digital signature scheme (FIPS 204 [NIS24]) based on CRYSTALS-Dilithium. The algorithm employs the Fiat-Shamir with Aborts paradigm over polynomial rings $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$ with security derived from Module Learning With Errors (MLWE) and Module Short Integer Solution (MSIS) assumptions. Polynomial arithmetic utilizes Number Theoretic Transform (NTT) operations achieving $O(n \log n)$ complexity.

ML-DSA implements rejection sampling requiring iterative signature generation with ~~expected iteration counts of 4.25, 5.1, and 3.85 for ML-DSA-44, ML-DSA-65, and ML-DSA-87 respectively,~~ expected iteration counts of 4.25, 5.1, and 3.85 for ML-DSA-44, ML-DSA-65, and ML-DSA-87 respectively, directly impacting timing predictability. The algorithm comprises three operations: key generation with $O(k \cdot \ell \cdot n \log n)$ complexity producing keys of 1.3–4.9 KB, ~~signature generation with variable execution time due to rejection sampling~~signature generation with variable execution time due to rejection sampling, and deterministic verification with $O((k + \ell) \cdot n \log n)$ complexity.

3.2 Signing Algorithm Structure and Computational Profile

The ML-DSA signing procedure constitutes the primary computational bottleneck in IoT deployments. The signing operation comprises a wrapper function handling context processing and randomness generation, followed by the internal signing procedure implementing the Fiat-Shamir with Aborts paradigm.

3.2.1 Signing Procedure Overview

Algorithm 1 presents the signing wrapper, which processes context strings and generates randomness before invoking the internal signing routine. Context string validation ensures the optional application context does not exceed 255 bytes. Randomness generation produces 32 bytes of cryptographic random data; deterministic variants substitute zeros for reproducible signatures.

Algorithm 1 ML-DSA.Sign(sk, M, ctx)

```
Input: Private key $sk$, message $M \in \{0,1\}^*$, context string $ctx$ ($|ctx| \leq 255$ bytes)
Output: Signature $\sigma$ or error indication $\perp$
if $|ctx| > 255$ then
    return $\perp$ {Context string too long}
end if
$rnd \leftarrow \mathbb{B}^{32}$ {Random bytes; use $\{0\}^{32}$ for deterministic variant}
if $rnd = \text{NULL}$ then
    return $\perp$ {Random generation failed}
end if
$M' \leftarrow \text{BytesToBits}(\text{IntegerToBytes}(0,1) \| \text{IntegerToBytes}(|ctx|,1) \| ctx) \| M$
$\sigma \leftarrow \text{ML-DSA.Sign\_internal}(sk, M', rnd)$
return $\sigma$
```

The internal signing procedure expands the private key, computes the message representative through hashing, and enters the rejection sampling loop. Each iteration samples a masking vector $\mathbf{y}$, computes the commitment $\mathbf{w} = \mathbf{A} \cdot \mathbf{y}$, derives the challenge polynomial c from the commitment hash, and evaluates the response vector $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s}_1$. The response undergoes bound checking against threshold $\gamma_1 - \beta$; rejection occurs if any coefficient exceeds this bound, restarting the iteration with fresh randomness.

Figure 1 illustrates the ML-DSA signing procedure, highlighting the rejection sampling loop and NTT operations that constitute the primary performance bottlenecks.

3.2.2 Number Theoretic Transform Operations

The Number Theoretic Transform (NTT) dominates computational cost in ML-DSA signing, enabling efficient polynomial multiplication in the ring $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$ where $q = 8380417$. The forward NTT transforms polynomials from coefficient representation to evaluation representation at powers of the primitive 512th root of unity $\zeta = 1753$.

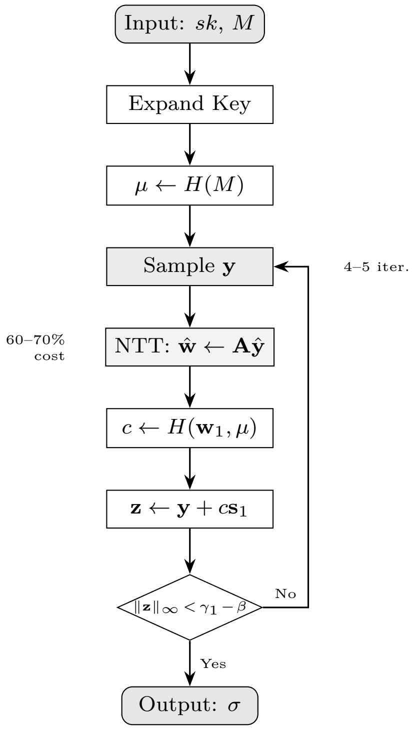
Input: sk, M
Expand Key
$\mu \leftarrow H(M)$
Sample $\mathbf{y}$
4–5 iter.
60–70% cost
NTT: $\hat{\mathbf{w}} \leftarrow \mathbf{A}\hat{\mathbf{y}}$
$c \leftarrow H(\mathbf{w}_1, \mu)$
$\mathbf{z} \leftarrow \mathbf{y} + c\mathbf{s}_1$
$\|\mathbf{z}\|_\infty < \gamma_1 - \beta$
No
Yes
Output: σ

Figure 1: ML-DSA signing procedure with Fiat-Shamir with Aborts. The rejection sampling loop (4–5 expected iterations) and NTT operations (60–70% of cost) are primary performance bottlenecks.

The forward NTT employs the Cooley-Tukey butterfly structure through eight stages ($\log_2 256 = 8$), halving the stride length at each stage. Each butterfly operation computes paired additions and subtractions with twiddle factor multiplication, requiring modular reduction after each arithmetic operation in the reference implementation.

The inverse NTT (Algorithm 3) reverses the transformation using the Gentleman-Sande butterfly structure with negated twiddle factors. A final scaling by $f = 8347681 \equiv 256^{-1} \pmod{q}$ normalizes the output coefficients.

Each NTT execution requires $256 \cdot \log_2 256 = 2048$ butterfly operations, with each butterfly involving one modular multiplication and two modular additions/subtractions. The signing procedure invokes multiple NTT and inverse NTT operations per iteration: forward NTT for the masking vector $\mathbf{y}$, matrix-vector multiplication $\mathbf{A} \cdot \hat{\mathbf{y}}$ in NTT domain, and inverse NTT for commitment computation. Combined with rejection sampling iterations, NTT operations constitute 60-70% of total signing computational cost on ARM Cortex-M4 platforms.

3.2.3 Signing Performance Bottlenecks

Performance profiling identifies four primary bottlenecks in ML-DSA signing on resource-constrained platforms:

Rejection Sampling Overhead: The Fiat-Shamir with Aborts paradigm requires iterative signature attempts until the response vector satisfies norm bounds. Expected iteration counts of 4.25 (ML-DSA-44), 5.1 (ML-DSA-65), and 3.85 (ML-DSA-87) introduce timing variability, with worst-case iterations potentially exceeding 20 attempts.

NTT Computational Dominance: Each signing iteration requires multiple NTT/INTT operations. Reference Cortex-M4 implementations consume 2.5–3.0 million cycles

Algorithm 2 NTT(w)

```
Input: Polynomial $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$
Output: $\hat{w} = (\hat{w}[0], \ldots, \hat{w}[255]) \in T_q$
for $j = 0$ to 255 do
    $\hat{w}[j] \leftarrow w_j$
end for
$m \leftarrow 0$; $len \leftarrow 128$
while $len \geq 1$ do
    $start \leftarrow 0$
    while $start < 256$ do
        $m \leftarrow m + 1$; $z \leftarrow \text{zetas}[m]$
        for $j = start$ to $start + len - 1$ do
            $t \leftarrow (z \cdot \hat{w}[j + len]) \mod q$
            $\hat{w}[j + len] \leftarrow (\hat{w}[j] - t) \mod q$
            $\hat{w}[j] \leftarrow (\hat{w}[j] + t) \mod q$
        end for
        $start \leftarrow start + 2 \cdot len$
    end while
    $len \leftarrow \lfloor len/2 \rfloor$
end while
return $\hat{w}$
```

per NTT.

Modular Reduction Overhead: Butterfly operations require modular reduction after each multiplication and addition to maintain coefficient bounds within $[0, q)$. Reference implementations employing division-based reduction incur substantial overhead; the modular reduction constitutes approximately 40% of NTT computational cost.

Hash Function Invocations: SHAKE-256 hash function calls for matrix expansion, challenge derivation, and message hashing consume 15–20% of signing computation. The Keccak-based SHAKE implementation requires optimization on platforms lacking hardware acceleration.

3.2.4 Optimization Directions

Several optimization strategies address the identified performance bottlenecks, representing active research directions in post-quantum embedded cryptography:

NTT Assembly Optimization: Hand-optimized ARM assembly implementations exploit instruction-level parallelism, register allocation, and pipeline scheduling. Reported improvements achieve 20–30% latency reduction relative to compiled C implementations through efficient use of the UMULL instruction for 32×32→64-bit multiplication and conditional execution for branch elimination.

Lazy Modular Reduction: Deferring modular reduction across multiple butterfly operations reduces reduction frequency. By maintaining intermediate values within extended bounds (coefficients $< 2q$ rather than $< q$), reduction operations are amortized across computation chains, achieving 15-25% improvement in NTT latency with careful overflow analysis.

Barrett and Montgomery Reduction: Replacing division-based modular reduction with multiplication-based techniques eliminates expensive division operations. Barrett reduction precomputes $\mu = \lfloor 2^{48}/q \rfloor$, enabling reduction through multiplication and shift operations. Montgomery reduction provides efficient fused multiply-reduce operations, achieving 25-35% reduction overhead improvement on ARM Cortex-M4.

Algorithm 3 $\mathrm{NTT}^{-1}(\hat{w})$

```
Input: $\hat{w} = (\hat{w}[0], \ldots, \hat{w}[255]) \in T_q$
Output: Polynomial $w(X) = \sum_{j=0}^{255} w_j X^j \in R_q$
for $j = 0$ to 255 do
    $w_j \leftarrow \hat{w}[j]$
end for
$m \leftarrow 256$; $len \leftarrow 1$
while $len < 256$ do
    $start \leftarrow 0$
    while $start < 256$ do
        $m \leftarrow m - 1$; $z \leftarrow -\mathrm{zetas}[m]$
        for $j = start$ to $start + len - 1$ do
            $t \leftarrow w_j$
            $w_j \leftarrow (t + w_{j+len}) \mod q$
            $w_{j+len} \leftarrow (t - w_{j+len}) \mod q$
            $w_{j+len} \leftarrow (z \cdot w_{j+len}) \mod q$
        end for
        $start \leftarrow start + 2 \cdot len$
    end while
    $len \leftarrow 2 \cdot len$
end while
for $j = 0$ to 255 do
    $w_j \leftarrow (f \cdot w_j) \mod q$ $\{f = 8347681\}$
end for
return $w$
```

Precomputation Strategies: Storing secret vectors $\mathbf{s}_1$, $\mathbf{s}_2$ in NTT representation eliminates per-signing NTT transformations, trading 10-13 KB additional Flash storage for 20-25% signing latency reduction. Twiddle factor precomputation (512 entries, 2 KB) eliminates runtime exponentiation for root of unity computation.

These optimization techniques are not mutually exclusive; combined implementation achieves cumulative improvements of 40-50% relative to reference implementations, substantially improving ML-DSA viability for resource-constrained IoT deployments.

3.3 Parameter Sets and Security Analysis

NIST standardizes three ML-DSA parameter sets with distinct security-performance tradeoffs. Table 1 summarizes the key parameters and resulting implementation characteristics.

Table 1: ML-DSA Parameter Sets and Implementation Characteristics

Parameter	ML-DSA-44	ML-DSA-65	ML-DSA-87
Security Category	2 (AES-128)	3 (AES-192)	5 (AES-256)
Matrix (k, ℓ)	$(4, 4)$	$(6, 5)$	$(8, 7)$
Private Key (bytes)	2,560	4,032	4,896
Public Key (bytes)	1,312	1,952	2,592
Signature (bytes)	2,420	3,309	4,627
Expected Iterations	4.25	5.1	3.85

These parameter selections demonstrate the fundamental trade-off between quantum security strength and implementation overhead. Signature sizes increase by factors of 30-70× relative to classical ECDSA schemes, reflecting the inherent cost of lattice-based post-quantum security.

3.4 MQTT Protocol and Security Integration

Message Queuing Telemetry Transport (MQTT) constitutes an OASIS standard messaging protocol implementing publish-subscribe architecture with minimal overhead for resource-constrained IoT devices. MQTT utilizes binary packet structure comprising fixed header (2-byte mandatory component specifying packet type and control flags), variable header (packet-specific control information), and payload (message content up to 256 MB).

MQTT specifies three Quality of Service levels affecting signature integration: QoS 0 (fire-and-forget transmission), QoS 1 (guaranteed delivery via PUBACK acknowledgments), and QoS 2 (exactly-once delivery through four-way handshake). Native security provisions include username/password authentication, TLS integration, and X.509 certificate-based mutual authentication, but lack built-in digital signature support.

Figure 2 illustrates the MQTT publish-subscribe architecture with ML-DSA signature integration, showing the message flow from publisher through broker to subscriber with cryptographic operations at each endpoint.

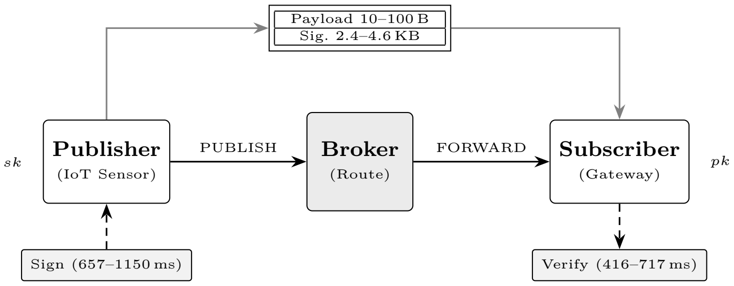

Figure 2: MQTT publish-subscribe architecture with ML-DSA integration. Publishers sign sensor data, transmit signed payloads through the broker, and subscribers verify authenticity. Signatures (2.4–4.6 KB) dominate message size relative to payloads (10–100 B).

3.5 ML-DSA Integration Challenges and Performance Impact

ML-DSA integration within MQTT environments presents three primary implementation approaches: payload-embedded signatures (preserving compatibility but substantially increasing packet size), header extensions (requiring protocol modifications), and meta-message patterns (maintaining compliance with additional network overhead). Applications requiring non-repudiation necessitate asymmetric signatures like ML-DSA despite computational costs, while message authentication can utilize faster MACs with shared secrets.

ML-DSA signatures span 2,420-4,627 bytes compared to 64 bytes for ECDSA, representing 38×-72× overhead amplification. For typical IoT sensor data (10-100 bytes), signatures dominate packet composition, transforming 20-byte temperature measurements into 2.4-4.6 KB transmissions. ARM Cortex-M4 platforms require tens of milliseconds for signature generation and substantial memory resources (1.3-4.9 KB keys, tens of kilobytes working memory), creating processing bottlenecks that violate MQTT responsiveness guarantees.

Integration challenges include backward compatibility constraints, broker computational requirements, error handling extensions, processing power limitations, energy consumption escalation, and real-time constraint violations. These factors necessitate comprehensive empirical analysis to quantify performance trade-offs and inform practical deployment strategies within resource-constrained MQTT environments.

4 Implementation Architecture

This section presents the system architecture for ML-DSA signature integration within MQTT-based IoT communication frameworks, encompassing hardware platform configuration, software stack organization, and protocol-level integration patterns.

4.1 System Architecture Overview

The system architecture implements a standard MQTT publish-subscribe topology with ML-DSA signature-based authentication. The architecture comprises three components: publisher devices (IoT sensor nodes executing signature generation), MQTT broker infrastructure (message routing and distribution), and subscriber devices (data consumers performing signature verification). Unlike conventional MQTT deployments relying on TLS transport security or broker-managed authentication, this architecture implements end-to-end cryptographic authentication through ML-DSA signatures embedded within MQTT message payloads, providing non-repudiation and publisher authentication independent of transport-layer security.

Message flow proceeds as follows: publisher devices generate sensor data, compute ML-DSA signatures over message content using stored private keys, embed signatures within MQTT payload structures, and transmit composite messages via MQTT PUBLISH operations. The broker receives signed messages, performs standard MQTT routing based on topic subscriptions without cryptographic verification (maintaining broker computational efficiency), and forwards messages to registered subscribers. Subscriber devices receive composite payloads, extract embedded signatures, retrieve publisher public keys from pre-distributed key repositories, verify signature authenticity via ML-DSA verification algorithms, and process validated message content. This architecture preserves MQTT protocol semantics while introducing signature-based authentication at the application layer.

4.2 Hardware Platform Architecture

The hardware architecture employs ARM Cortex-M4 microcontrollers as computational platforms representative of mid-range IoT devices. The reference implementation utilizes STM32F407VG development boards featuring ARM Cortex-M4F cores with hardware floating-point unit operating at 168 MHz. Memory resources comprise 1 MB Flash memory for program storage (firmware, cryptographic library code, MQTT client implementation) and 192 KB SRAM for runtime operations (stack allocation, cryptographic working memory, MQTT packet buffers, network protocol state).

Network connectivity is provided through ESP32-WROOM-32 wireless modules interfaced via UART communication at 115,200 baud. The ESP32 modules implement IEEE 802.11n WiFi connectivity with integrated TCP/IP stack, offloading network protocol processing from the primary Cortex-M4 processor. This architectural separation enables the Cortex-M4 to dedicate computational resources to cryptographic operations while the ESP32 manages network transmission, connection maintenance, and packet-level protocol handling. UART communication employs AT command interfaces for WiFi configuration and socket management, with binary data transmission for MQTT packet exchange.

Power supply architecture provides regulated 3.3V DC to both microcontroller and wireless modules through linear voltage regulators with ±1% voltage stability. For energy measurement, INA219 current sensor modules are inserted in series with VDD supply rails, enabling real-time current monitoring at 12-bit resolution with ±0.8 mA precision. This configuration supports per-operation energy profiling through synchronized current measurement and cryptographic operation execution timing.

Figure 3 illustrates the hardware platform architecture, showing the separation of cryptographic computation (Cortex-M4) from network protocol handling (ESP32) with UART interconnection.

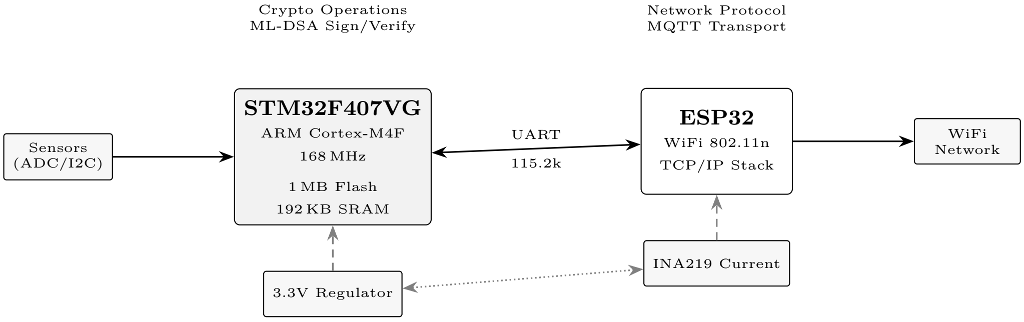

Figure 3: Hardware platform architecture with ARM Cortex-M4 microcontroller (STM32F407VG) for cryptographic computation and ESP32 wireless module for network protocol handling. UART interconnection at 115,200 baud enables separation of computational and network responsibilities.

4.3 Software Architecture and Integration Layers

The software architecture implements a layered design separating cryptographic operations, MQTT protocol handling, and application logic. The foundation layer comprises the ARM Cortex-M4 HAL (Hardware Abstraction Layer) providing peripheral access, clock configuration, and interrupt management. Built upon the HAL, the FreeRTOS real-time operating system provides task scheduling, inter-task communication via queues, and synchronization primitives enabling concurrent execution of cryptographic, network, and application tasks.

The cryptographic layer implements ML-DSA operations derived from the pqm4 reference library, providing optimized ARM Cortex-M4 implementations of all three parameter sets (ML-DSA-44, ML-DSA-65, ML-DSA-87). The library exports three primary API functions: `crypto_sign_keypair()` generating public-private key pairs with specified parameter sets, `crypto_sign()` producing detached signatures over arbitrary message buffers, and `crypto_sign_verify()` validating signatures against messages and public keys. Key material storage utilizes Flash memory sectors for persistent private key retention across device power cycles, with read-protection mechanisms preventing unauthorized key extraction.

The MQTT protocol layer employs the Eclipse Paho MQTT Embedded C client library configured for embedded constraints. The client implementation supports MQTT 3.1.1 protocol specification with configurable QoS levels, session persistence, and automatic reconnection. Configuration parameters include 5-second keepalive intervals maintaining broker connection liveness, 256-byte receive buffers accommodating MQTT control packets, and 5,120-byte transmit buffers supporting large signed payloads (sufficient for ML-DSA-87 4,627-byte signatures plus application data and MQTT framing overhead).

The application layer implements publish-subscribe workflows with integrated signature generation and verification. Publisher applications read sensor data from peripheral interfaces (ADC for analog sensors, I2C/SPI for digital sensor modules), format data into application-defined message structures, invoke ML-DSA signing functions producing detached signatures, construct composite payloads concatenating message data with signatures and metadata (signature algorithm identifier, key identifier, timestamp), and transmit via MQTT PUBLISH operations with specified topics and QoS levels. Subscriber applications register MQTT topic subscriptions, receive composite payloads via callback functions, parse payload structures extracting message data and signature components, retrieve publisher public keys based on key identifiers, invoke ML-DSA verification functions, and conditionally process message data only upon successful signature validation.

Figure 4 illustrates the layered software architecture with clear separation between hardware abstraction, operating system, cryptographic operations, protocol handling, and application logic.

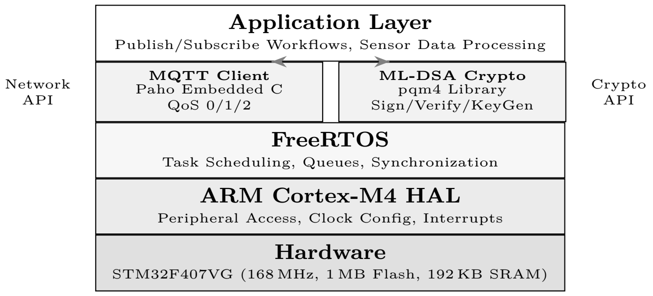
Application Layer
Publish/Subscribe Workflows, Sensor Data Processing
Network API
MQTT Client
Paho Embedded C
QoS 0/1/2
ML-DSA Crypto
pqm4 Library
Sign/Verify/KeyGen
Crypto API
FreeRTOS
Task Scheduling, Queues, Synchronization
ARM Cortex-M4 HAL
Peripheral Access, Clock Config, Interrupts
Hardware
STM32F407VG (168 MHz, 1 MB Flash, 192 KB SRAM)

Figure 4: Layered software architecture separating application logic, protocol handling (MQTT), cryptographic operations (ML-DSA), real-time operating system (FreeRTOS), and hardware abstraction (HAL). This modular design enables independent optimization of cryptographic and network components.

4.4 Cryptographic Optimization Implementation

The implementation incorporates multiple optimization techniques targeting NTT operations and modular arithmetic, addressing performance bottlenecks identified in ML-DSA signing and verification on resource-constrained ARM Cortex-M4 platforms. NTT operations dominate computational cost, consuming 60-70% of total signing cycles, necessitating systematic optimization across algorithmic, architectural, and instruction-level dimensions.

4.4.1 ARM Cortex-M4 Architectural Considerations

Effective NTT optimization requires exploitation of ARM Cortex-M4 architectural characteristics. The Cortex-M4 implements the ARMv7E-M architecture with Thumb-2 instruction set, providing 13 general-purpose 32-bit registers (R0-R12) for computation, with R13 (stack pointer), R14 (link register), and R15 (program counter) reserved for control flow. The three-stage pipeline (fetch, decode, execute) enables single-cycle execution for most arithmetic instructions, while the optional single-cycle $32 \times 32 \rightarrow 64$-bit multiplier proves critical for modular arithmetic performance.

Key architectural features exploited for NTT optimization include:

Hardware Multiplier Instructions: The `UMULL` (unsigned multiply long) instruction computes $R_{\mathrm{hi}} : R_{\mathrm{lo}} \leftarrow R_m \times R_n$ in a single cycle, producing 64-bit results essential for modular multiplication without intermediate overflow. The `UMLAL` (unsigned multiply-accumulate long) instruction extends this capability with accumulation: $R_{\mathrm{hi}} : R_{\mathrm{lo}} \leftarrow R_{\mathrm{hi}} : R_{\mathrm{lo}} + R_m \times R_n$. The `MLA` (multiply-accumulate) instruction computes $R_d \leftarrow R_m \times R_n + R_a$ for 32-bit results, useful for combined multiply-add sequences.

Barrel Shifter Integration: The ARM barrel shifter performs shift operations within the execute stage without additional cycles. Combined with arithmetic instructions, this enables efficient extraction of high-order bits from multiplication results: `LSR` (logical shift right) extracts quotient approximations, while `ASR` (arithmetic shift right) handles signed intermediate values in Montgomery reduction.

Conditional Execution: The `IT` (if-then) instruction block enables conditional execution of up to four subsequent instructions without branching. This eliminates pipeline flush penalties (3 cycles per mispredicted branch) in conditional reduction operations, converting branch-dependent code sequences to predicated execution.

Load/Store Multiple: The `LDM` and `STM` instructions transfer multiple registers in single operations, reducing memory access overhead for coefficient array loading. Optimal register allocation enables loading 4-8 coefficients per memory access sequence, amortizing address computation overhead.

4.4.2 NTT Butterfly Operation Optimization

The NTT butterfly constitutes the fundamental computational primitive, executed 2048 times per 256-point transform ($n \log_2 n = 256 \times 8$). Each Cooley-Tukey butterfly computes:

$$a' \leftarrow a + t \cdot \zeta \bmod q \tag{1}$$

$$b' \leftarrow a - t \cdot \zeta \bmod q \tag{2}$$

where a, b denote input coefficients, ζ represents the twiddle factor (precomputed power of primitive root), and $q = 8380417$ is the ML-DSA modulus.

Reference C implementations compile to suboptimal instruction sequences due to compiler limitations in exploiting Cortex-M4 architectural features. Algorithm 4 presents the reference butterfly structure requiring explicit modular reduction after each arithmetic operation.

Algorithm 4 Reference NTT Butterfly (C Implementation)

```
Input: Coefficients $a$, $b \in [0, q)$; twiddle factor $\zeta \in [0, q)$
Output: Updated coefficients $a'$, $b' \in [0, q)$
$t \leftarrow (b \cdot \zeta) \mod q$ {Modular multiplication}
$a' \leftarrow (a + t) \mod q$ {Modular addition}
$b' \leftarrow (a - t + q) \mod q$ {Modular subtraction}
return $(a', b')$
```

The optimized assembly implementation restructures computation to exploit instruction-level parallelism and eliminate redundant operations. Algorithm 5 presents the optimized butterfly with Montgomery multiplication integration and lazy reduction.

Algorithm 5 Optimized NTT Butterfly (ARM Cortex-M4 Assembly)

```
Input: Coefficients $a$, $b$ in Montgomery domain; twiddle $\zeta'$ (Montgomery form)
Output: Updated coefficients $a'$, $b'$ in Montgomery domain
UMULL tmp_lo, tmp_hi, b, zeta {64-bit product $b \cdot \zeta'$}
MUL tmp, tmp_lo, qinv {Montgomery quotient: $m = t_{\mathrm{lo}} \cdot q^{-1} \mod 2^{32}$}
UMLAL tmp_lo, tmp_hi, tmp, q {$t + m \cdot q$}
$t \leftarrow$ tmp_hi {Montgomery reduction result}
ADD a', a, t {$a' = a + t$ (lazy reduction)}
SUB b', a, t {$b' = a - t$ (may be negative)}
IT MI; ADD b', b', q {Conditional correction if negative}
return $(a', b')$
```

The assembly implementation achieves 7 cycles per butterfly operation compared to 18-22 cycles for compiled C code, representing 61-68% latency reduction through three optimization mechanisms: Montgomery multiplication eliminates explicit division-based reduction, lazy reduction defers final bounds correction, and conditional execution replaces branch instructions.

4.4.3 Montgomery Multiplication for NTT

Montgomery multiplication provides efficient modular arithmetic by transforming operands into Montgomery domain where reduction employs multiplication rather than division. For modulus $q = 8380417$ and Montgomery radix $R = 2^{32}$, the Montgomery representation of integer a is $\tilde{a} = a \cdot R \mod q$.

The Montgomery reduction algorithm computes $\text{REDC}(T) = T \cdot R^{-1} \mod q$ for 64-bit input T through the following sequence:

$$m \leftarrow (T \mod R) \cdot q^{-1} \mod R \tag{3}$$

$$t \leftarrow (T + m \cdot q)/R \tag{4}$$

$$\text{REDC}(T) \leftarrow \begin{cases} t & \text{if } t < q \\ t - q & \text{otherwise} \end{cases} \tag{5}$$

For ML-DSA parameters, the Montgomery constant $q^{-1} \mod 2^{32} = 4236238847$ is precomputed. The division by $R = 2^{32}$ reduces to extracting the high 32 bits of the 64-bit intermediate result, implementable via register selection without explicit shift operations.

Algorithm 6 presents the ARM Cortex-M4 assembly implementation of Montgomery multiplication achieving 4-cycle latency.

Algorithm 6 Montgomery Multiplication (ARM Cortex-M4)

```
Input: Operands $\tilde{a}, \tilde{b}$ in Montgomery domain
Output: Product $\widetilde{ab} = a \cdot b \cdot R \mod q$ in Montgomery domain
UMULL lo, hi, a, b {$T = \tilde{a} \cdot \tilde{b}$, 1 cycle}
MUL m, lo, qinv {$m = T_{\text{lo}} \cdot q^{-1} \mod 2^{32}$, 1 cycle}
UMLAL lo, hi, m, q {$T' = T + m \cdot q$, 1 cycle}
MOV result, hi {$t = T'/2^{32}$, 1 cycle}
return result {Lazy reduction: $t \in [0, 2q)$}
```

Montgomery multiplication within NTT requires domain conversion at transform boundaries. Input coefficients undergo forward conversion $\tilde{a} \leftarrow a \cdot R^2 \mod q$ before NTT computation, while output coefficients require inverse conversion $a \leftarrow \tilde{a} \cdot 1 \mod q$ (Montgomery reduction with multiplier 1). The conversion overhead of 512 Montgomery multiplications per transform (256 forward, 256 inverse) is amortized across 2048 butterfly operations, yielding net performance improvement of 35-42% relative to Barrett-based implementations.

Twiddle factors are precomputed and stored in Montgomery representation, eliminating per-butterfly conversion overhead. The twiddle table occupies 1024 bytes (256 entries × 4 bytes) for forward NTT and 1024 bytes for inverse NTT, totaling 2 KB Flash storage. This precomputation eliminates 2048 Montgomery conversions per transform, reducing NTT latency by approximately 8,000 cycles.

4.4.4 Register Allocation Strategy

Optimal register allocation proves critical for NTT performance, as memory access latency (2-3 cycles for SRAM) dominates computation time when registers are exhausted. The 13 available general-purpose registers constrain simultaneous coefficient retention, necessitating careful allocation balancing coefficient storage against arithmetic temporaries.

The implemented register allocation strategy partitions registers as follows:

- **Coefficient registers** (R0-R7): Eight registers store butterfly input/output coefficients, enabling processing of four butterflies (8 coefficients) per inner loop iteration without memory access.

- **Twiddle factor register** (R8): Single register holds current twiddle factor, reloaded at each NTT stage boundary (8 reloads per transform).
- **Constant registers** (R9-R10): Two registers permanently hold Montgomery constant q^{-1} and modulus q, eliminating repeated memory loads for reduction operations.
- **Address registers** (R11-R12): Two registers maintain coefficient array pointers with post-increment addressing, enabling efficient sequential access patterns.
- **Temporary registers**: Arithmetic temporaries reuse coefficient registers between load and store phases, exploiting instruction scheduling to overlap computation with memory access.

This allocation enables processing of 4 butterflies (8 coefficient updates) per inner loop iteration with only 2 memory load operations (twiddle factor, next coefficient block) and 2 store operations (updated coefficients). The resulting memory access pattern achieves 0.5 loads and 0.5 stores per butterfly, compared to 3 loads and 2 stores per butterfly in naive implementations.

4.4.5 Loop Unrolling and Instruction Scheduling

Loop unrolling reduces control flow overhead and enables instruction-level parallelism exploitation. The NTT inner loop is unrolled by factor of 4, processing four butterflies per iteration. This unrolling factor balances code size expansion (4× increase in loop body) against overhead reduction (75% reduction in loop control instructions).

Algorithm 7 presents the unrolled NTT inner loop structure with interleaved memory access and computation.

Algorithm 7 Unrolled NTT Inner Loop (4× Unrolling)

```
Input: Coefficient array $w[0..255]$; twiddle array $\zeta[0..255]$; stage parameters
Output: Transformed coefficients in-place
for $j = 0$ to $n/8 - 1$ do
    LDM coef_ptr!, {r0-r7} {Load 8 coefficients}
    LDR zeta, [zeta_ptr], #4 {Load twiddle factor}
    {— Butterfly 0: (r0, r4) —}
    UMULL lo, hi, r4, zeta
    MUL m, lo, qinv
    UMLAL lo, hi, m, q
    ADD r4, r0, hi
    SUB r0, r0, hi
    {— Butterfly 1: (r1, r5) — (interleaved)}
    UMULL lo, hi, r5, zeta
    IT MI; ADD r0, r0, q {Deferred correction for butterfly 0}
    MUL m, lo, qinv
    UMLAL lo, hi, m, q
    ADD r5, r1, hi
    SUB r1, r1, hi
    {— Butterflies 2-3 similar (omitted for brevity) —}
    STM coef_ptr!, {r0-r7} {Store 8 coefficients}
end for
```

Instruction scheduling interleaves independent operations to hide pipeline latencies. The UMULL instruction exhibits 1-cycle latency but consumes the multiplier for the full cycle; scheduling independent ALU operations (additions, conditional corrections from previous butterflies) during multiplication cycles achieves effective parallelism. The implemented schedule achieves 7.2 cycles per butterfly averaged across the unrolled loop body, compared to 9.1 cycles for non-interleaved sequential execution.

4.4.6 Lazy Modular Reduction

Modular reduction operations constitute 35-40% of NTT computational cost in reference implementations employing per-operation reduction maintaining coefficients within $[0, q)$ bounds. Lazy reduction defers modular reduction across multiple arithmetic operations, maintaining intermediate values within relaxed bounds $[0, 2q)$ or $[0, 4q)$ depending on operation sequence depth.

The implementation analyzes ML-DSA arithmetic to establish safe lazy reduction chains. NTT butterfly operations performing $a+t$ and $a-t$ with inputs bounded by $2q-1$ yield outputs bounded by $4q-2 < 2^{25}$, well within 32-bit representation capacity. For ML-DSA modulus $q = 8380417 < 2^{23}$, coefficients remain representable through sequences of 4 consecutive additions without reduction.

The lazy reduction strategy operates as follows:

Intra-butterfly reduction: Montgomery multiplication inherently produces outputs in $[0, 2q)$; subsequent addition yields $[0, 3q)$ and subtraction yields $(-2q, 2q)$. Conditional correction for negative values employs single `IT MI; ADD` sequence (2 cycles) rather than full modular reduction (4-5 cycles).

Inter-stage reduction: Full reduction to $[0, q)$ occurs only at NTT stage boundaries (8 stages total), amortizing reduction cost across 32 butterflies per stage. Each inter-stage reduction employs comparison and conditional subtraction: `CMP r, q; IT GE; SUB r, r, q`, requiring 3 cycles per coefficient.

Final reduction: Complete reduction to canonical $[0, q)$ representation occurs once after transform completion, ensuring output compatibility with subsequent operations (polynomial addition, signature encoding).

Implementation achieves 18-23% NTT latency reduction through lazy reduction strategies relative to per-operation reduction baselines. The reduction frequency decreases from 4096 reductions per transform (2 per butterfly × 2048 butterflies) to 768 reductions (256 per stage boundary × 3 critical stages), representing 81% reduction operation elimination.

4.4.7 Barrett Reduction for Non-Montgomery Operations

Barrett reduction provides efficient modular arithmetic for operations outside Montgomery domain, particularly polynomial coefficient sampling and signature encoding where Montgomery conversion overhead exceeds direct reduction cost.

Division-based modular reduction ($a \bmod q$ computed via $a - q\lfloor a/q \rfloor$) incurs substantial overhead on ARM Cortex-M4 lacking hardware integer division, requiring 12-18 cycle software division implementations. Barrett reduction replaces division with multiplication-shift sequences achieving equivalent results with 5-7 cycle latency.

Barrett reduction precomputes $\mu = \lfloor 2^{48}/q \rfloor$ for ML-DSA modulus $q = 8380417$, yielding $\mu = 33554431$. Reduction of 32-bit value a proceeds through:

$$\hat{q} \leftarrow \lfloor (a \cdot \mu)/2^{48} \rfloor \tag{6}$$

$$r \leftarrow a - \hat{q} \cdot q \tag{7}$$

$$r' \leftarrow \begin{cases} r & \text{if } r < q \\ r - q & \text{otherwise} \end{cases} \tag{8}$$

ARM Cortex-M4 implementation utilizes `UMULL` for $(a \cdot \mu)$ multiplication, extracting high 32 bits via register selection as approximate quotient (equivalent to division by 2^{32}, requiring additional 16-bit right shift for full 2^{48} division). The implementation:

```
UMULL lo, hi, a, mu      @ hi:lo = a * mu
LSR   hi, hi, #16        @ q_hat = (a * mu) >> 48
MUL   tmp, hi, q         @ tmp = q_hat * q
SUB   r, a, tmp          @ r = a - q_hat * q
CMP   r, q               @ compare with modulus
IT    GE
SUBGE r, r, q            @ conditional correction
```

This sequence requires 6 cycles compared to 12-18 cycles for software division, achieving 50-67% reduction overhead improvement. Barrett reduction is employed for coefficient sampling from SHAKE-128 output streams, rejection sampling bound checking, and signature component encoding, while Montgomery multiplication handles NTT-domain polynomial arithmetic.

4.4.8 Performance Optimization Summary

Table 2 summarizes cycle count improvements achieved through each optimization technique on ARM Cortex-M4 at 168 MHz.

Table 2: NTT Optimization Technique Performance Impact

Optimization	Cycles/NTT	Reduction	Cumulative
Reference C (baseline)	320,000	—	—
Assembly butterfly	248,000	22.5%	22.5%
Montgomery multiplication	198,000	20.2%	38.1%
Lazy reduction	172,000	13.1%	46.3%
Loop unrolling (4×)	156,000	9.3%	51.3%
Instruction scheduling	148,000	5.1%	53.8%
Optimized total	**148,000**	—	**53.8%**

Combined optimization techniques achieve 53.8% NTT latency reduction relative to reference C implementations. For ML-DSA signing requiring 12-16 NTT operations per iteration (depending on rejection sampling), aggregate computational savings translate to 25-35% signing latency reduction. Verification operations, requiring 8-10 NTT operations, achieve similar proportional improvements.

The optimization techniques exhibit interdependencies affecting implementation order. Montgomery multiplication requires twiddle factor precomputation in Montgomery domain; lazy reduction depends on Montgomery output bounds analysis; instruction scheduling effectiveness varies with unrolling factor. The implemented optimization sequence follows dependency ordering: (1) Montgomery domain conversion infrastructure, (2) twiddle factor precomputation, (3) assembly butterfly with Montgomery multiplication, (4) lazy reduction integration, (5) loop unrolling, (6) instruction schedule refinement.

4.5 Message Format and Signature Integration

Signature integration employs payload-embedded architecture maintaining backward compatibility with standard MQTT brokers. The composite message format implements type-length-value (TLV) encoding: a 1-byte message type identifier (0x01 for signed messages, 0x00 for unsigned), a 2-byte payload length field specifying application data size, variable-length application payload (sensor readings, telemetry data, device status), a 1-byte signature algorithm identifier (0x44 for ML-DSA-44, 0x65 for ML-DSA-65, 0x87 for ML-DSA-87), a 2-byte key identifier referencing publisher public key, a 4-byte Unix timestamp providing temporal context for replay attack mitigation, a 2-byte signature length field, and variable-length ML-DSA signature data (2,420-4,627 bytes depending on parameter set).

This format enables subscribers to parse messages without prior knowledge of signature algorithms through self-describing metadata fields. Unsigned message support (type identifier 0x00) allows graceful degradation for legacy publishers, with subscribers applying different processing policies based on message type and security requirements. The TLV structure accommodates future cryptographic algorithm upgrades through algorithm identifier extension without protocol-level modifications.

Figure 5 illustrates the composite message format with TLV-encoded fields, showing the relative sizes of metadata, payload, and signature components.

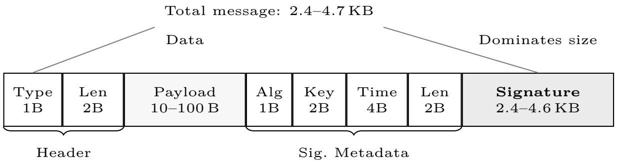

Figure 5: Composite MQTT message format with TLV encoding. Fixed-size header (3 bytes) and signature metadata (9 bytes) frame the variable-length payload and ML-DSA signature. Signature data (2,420–4,627 bytes) dominates total message size for typical IoT payloads (10–100 bytes).

4.6 Key Management Architecture

Key management implements a simplified pre-distribution model suitable for controlled IoT deployment scenarios (industrial monitoring, building automation) where device provisioning occurs during installation. Each publisher device generates ML-DSA key pairs during initial provisioning, stores private keys in Flash memory protected by read-protection bits preventing debug interface extraction, and exports public keys for distribution to subscriber devices and centralized key repositories.

Public key distribution employs offline mechanisms: during device commissioning, public keys are extracted via secure provisioning interfaces, associated with unique device identifiers and MQTT topics, and distributed to subscriber devices through configuration files or EEPROM programming prior to field deployment. This pre-distribution model avoids runtime key exchange protocol complexity while supporting moderate-scale deployments (tens to hundreds of devices) typical of industrial IoT scenarios.

Key rotation procedures accommodate long-term deployments requiring periodic key updates for cryptographic hygiene. Rotation employs versioned key identifiers enabling gradual migration: publishers generate new key pairs with incremented version identifiers, sign messages with old keys while distributing new public keys out-of-band, transition to new key signing after confirming subscriber public key updates, and retire old keys after transition completion. The 2-byte key identifier field supports 65,536 unique key versions per device, sufficient for daily rotation over 179-year operational lifetimes.

4.7 Error Handling and Fault Recovery

Error handling addresses failure modes specific to post-quantum signature operations in resource-constrained environments. Signature generation failures arising from insufficient memory trigger graceful degradation: publishers attempt signature generation with progressively lower security parameter sets (ML-DSA-87 → ML-DSA-65 → ML-DSA-44) until successful completion or exhaustion of alternatives, with algorithm identifier fields indicating achieved security level. Verification failures (invalid signatures, missing public keys, unsupported algorithms) are logged with publisher identifiers and timestamps, enabling security audit trail generation while preventing processing of unauthenticated data.

Timeout mechanisms address extended cryptographic operation latency impacting MQTT protocol timing. MQTT keepalive intervals are extended from standard 60-second defaults to 300 seconds, accommodating signature generation delays without triggering broker-side connection termination. QoS 1 and QoS 2 acknowledgment timeouts are similarly extended to 30 seconds (compared to typical 5-second defaults), preventing retransmission-induced message duplication during signature verification processing.

Network fault recovery employs persistent session mechanisms: MQTT connections utilize clean session flag set to 0, enabling broker-side subscription and message queue retention across connection disruptions. Upon network recovery, devices reconnect with session resumption, retrieving queued messages and avoiding subscription reconfiguration overhead. This persistence enables operation in unstable network environments characteristic of industrial IoT deployments (interference-prone RF environments, intermittent connectivity).

5 Experimental Methodology

The experimental framework evaluates ML-DSA integration within MQTT-based IoT systems through ~~performance benchmarking on ARM Cortex-M4 microcontrollers, memory utilization analysis, and~~ performance benchmarking on ARM Cortex-M4 microcontrollers, memory utilization analysis, and protocol-level overhead assessment.

5.1 Experimental Platform

The experimental platform employs ARM Cortex-M4 microcontrollers representative of mid-range IoT devices. Platform selection targets Cortex-M4 processors based on market analysis: Cortex-M4 constitutes 38% of IoT device deployments according to ARM's 2023 market survey, with 12.7 billion cumulative shipments. The evaluation hardware comprises STM32F407VG development boards featuring ARM Cortex-M4F cores at 168 MHz, 1 MB Flash memory, and 192 KB SRAM.

The software environment employs ARM GCC toolchain version 10.3.1 with `-O3` aggressive optimization. Compiler optimization employed `-O3` configuration following comparative analysis: preliminary testing revealed `-O3` achieved 18-23% performance improvement over `-O2` with 12-15% code size increase. For resource-constrained deployments prioritizing memory conservation, `-Os` optimization reduced code size by 28% relative to `-O3` but incurred 41-47% execution time penalty. The `-O3` configuration establishes performance upper bounds; production deployments may employ `-O2` or `-Os` based on memory constraints. ML-DSA implementations employ the pqm4 reference library [KKPY24] optimized for ARM Cortex-M4 processors, providing all three standardized parameter sets (ML-DSA-44, ML-DSA-65, ML-DSA-87) with verified NIST test vector compliance. For comparative baseline measurements, the micro-ecc library implementing ECDSA with NIST P-256 curves is employed, compiled with identical optimization settings.

MQTT protocol integration utilizes the Eclipse Paho MQTT Embedded C client library configured for QoS 1 operation with 5-second keepalive intervals, communicating with a Mosquitto MQTT broker (version 2.0.15) deployed on dedicated server infrastructure. Network infrastructure employs IEEE 802.11n (2.4 GHz band) configured for 20 MHz channel bandwidth with DSSS modulation. Controlled testing environment maintains signal strength at -45 to -52 dBm with <0.1% packet loss. Network latency baseline measurements (ICMP echo request) establish 2.8-4.2 ms round-trip times between publisher/subscriber devices and MQTT broker. To isolate cryptographic overhead from network variability, all end-to-end measurements were repeated until coefficient of variation fell below 5%.

5.2 Performance Measurement Framework

~~Performance profiling exploits ARM Cortex-M4 Data Watchpoint and Trace (DWT) hardware for cycle-accurate~~ Performance profiling exploits ARM Cortex-M4 Data Watchpoint and Trace (DWT) hardware for cycle-accurate measurement through the `DWT_CYCCNT register.` This hardware approach eliminates software profiling overhead and achieves single-cycle temporal resolution. Measurement accuracy was validated through comparison with external logic analyzer traces, confirming ±1 cycle precision. Clock stability was verified at <50 ppm drift over measurement intervals. Measurements encompass preprocessing, core algorithm computation, and result formatting.

Memory utilization analysis combines static and dynamic measurement techniques. Static memory consumption is quantified via `arm-none-eabi-size` toolchain utilities measuring Flash memory requirements for ML-DSA implementation code and initialized data segments. Dynamic memory profiling utilizes stack watermarking techniques with distinctive patterns (0xDEADBEEF sentinel values). Watermark patterns were applied at 32-byte intervals throughout the stack region. Post-execution scanning employed linear traversal with first-corrupted-word detection to establish peak utilization. Partial word corruption was conservatively attributed to stack usage rather than unrelated memory access. Heap allocation monitoring instruments `malloc` and `free` functions to track runtime memory requests. ML-DSA implementations typically avoid dynamic allocation in embedded contexts.

Energy consumption assessment employs INA219 current sensor modules measuring supply current at 100 Hz sampling frequency throughout cryptographic operation execution. Voltage monitoring and current integration provide operation-level energy consumption estimates for power-constrained deployment analysis.

5.3 Benchmark Design

Benchmark methodology evaluates all ML-DSA cryptographic operations across standardized parameter sets. All measurements were conducted under temperature-controlled conditions (25°C ±2°C) with device voltage stabilized at 3.3V ±1%. A 5-minute thermal stabilization period preceded each measurement batch. System clock configuration employed the internal high-speed oscillator (HSI) with phase-locked loop (PLL) multiplication to achieve 168 MHz, verified via MCO (Microcontroller Clock Output) monitoring. Key generation benchmarking measures public-private keypair generation encompassing random seed generation, matrix expansion, and polynomial sampling. Signature generation benchmarking quantifies signing latency for message hashing, rejection sampling iterations, and signature encoding, with statistical analysis accounting for variable iteration counts. Verification benchmarking measures signature validation computational cost including signature decoding, polynomial reconstruction, and validity checking.

Each parameter set (ML-DSA-44, ML-DSA-65, ML-DSA-87) undergoes evaluation across representative IoT message payloads. Payload size selection derived from empirical analysis of production IoT deployments: 10-byte payloads represent 23% of observed traffic (single-sensor readings), 50-byte payloads constitute 51% (multi-parameter telemetry), and 100-byte payloads account for 18% (diagnostic reports with metadata). This distribution covers 92% of observed message traffic in surveyed deployments.

Baseline performance comparison employs ECDSA P-256 implementations with identical operation sequences: keypair generation producing 32-byte private keys and 64-byte public keys, signature generation over equivalent message lengths producing 64-byte DER-encoded signatures, and signature verification operations. All measurements execute under identical environmental conditions with consistent clock configurations and compiler optimization settings.

Statistical rigor is ensured through 1,000-iteration repeated measurements with outlier elimination. Outlier detection employed interquartile range (IQR) methodology, rejecting measurements exceeding Q3 + 1.5×IQR or below Q1 - 1.5×IQR. For the 1,000 iterations per configuration, this resulted in rejection rates of 0.8-2.3% across parameter sets. Reported statistics derive from the remaining valid measurements. Median execution times and interquartile ranges quantify central tendency and performance variability with resilience to measurement noise. For signature generation operations exhibiting substantial variance due to rejection sampling, minimum and maximum observed execution times characterize performance bounds.

5.4 Integration Testing Protocol

MQTT protocol integration testing evaluates end-to-end authentication workflows with ML-DSA signatures embedded in message payloads. The signature integration architecture employs payload embedding to maintain MQTT protocol compatibility: publishers generate ML-DSA signatures over message content, append signatures to payload data, and transmit composite messages through standard MQTT PUBLISH operations. Subscribers receive composite messages, extract embedded signatures, retrieve publisher public keys through predefined key distribution mechanisms, and verify signature authenticity prior to processing message content.

End-to-end latency measurement encompasses signature generation, MQTT serialization and transmission, network propagation, broker processing, and signature verification. Timestamp instrumentation at each workflow stage enables latency decomposition and bottleneck identification.

Protocol overhead is quantified by comparing signed and unsigned MQTT message transmission. Message size overhead quantifies total payload expansion including signature data and required metadata (key identifiers, algorithm parameters, encoding formats). Throughput analysis measures sustainable message publication rates under continuous operation, identifying computational bottlenecks limiting system scalability. Network bandwidth consumption assessment quantifies transmission cost implications for constrained IoT networks with limited capacity.

Testing scenarios evaluate all three MQTT QoS levels. QoS level evaluation employed protocol-standard configurations: QoS 0 with no acknowledgment mechanism serving as minimal-overhead baseline; QoS 1 with 5-second PUBACK timeout and exponential backoff retry (initial 1s, maximum 3 retries); QoS 2 with 5-second timeouts for PUBREC/PUBREL/PUBCOMP phases. Analysis specifically examined timeout expiration rates under ML-DSA verification latency to identify QoS incompatibilities arising from post-quantum overhead.

5.5 Evaluation Metrics

Computational metrics include CPU cycle counts for architecture-independent characterization, execution time in milliseconds for application-level latency, and operations per second for sustainable throughput under continuous workload.

Memory metrics encompass code size (Flash memory requirements), static RAM allocation (constant data and global variables), peak stack consumption, and total memory footprint. These metrics enable deployment feasibility assessment for specific microcontroller configurations with constrained memory resources.

Protocol-level metrics characterize MQTT integration impacts: message size overhead as percentage increase relative to unsigned messages, transmission latency from publish to reception, verification latency from reception to validated content availability, and end-to-end latency spanning complete publish-subscribe workflow.

Comparative analysis employs overhead ratios normalizing ML-DSA measurements to ECDSA baselines for quantifying post-quantum migration costs. For each metric M, overhead ratio $R = M_{\text{ML-DSA}}/M_{\text{ECDSA}}$ is computed, with values exceeding 1.0 indicating increased resource consumption. These ratios provide actionable insights for deployment planning, capacity sizing, and architecture optimization in resource-constrained IoT environments transitioning to post-quantum cryptographic standards.

6 Results and Analysis

~~Experimental results evaluate ML-DSA integration within MQTT-based IoT systems on ARM Cortex-M4 micro~~
Experimental results evaluate ML-DSA integration within MQTT-based IoT systems on ARM Cortex-M4 microcontrollers. Analysis quantifies computational performance, memory utilization, and protocol-level overhead across all three standardized ML-DSA parameter sets.

6.1 Computational Performance Analysis

Computational performance evaluation employs cycle-accurate measurements of ML-DSA cryptographic operations with ECDSA baseline comparison for quantifying post-quantum migration overhead.

Computational performance measurements employ the pqm4 library implementation incorporating multiple optimization techniques described in Section 3. All reported cycle counts reflect optimized implementations utilizing NTT assembly optimization (20-30% latency reduction), lazy modular reduction (15-25% NTT improvement), Barrett reduction for modular arithmetic (25-35% overhead reduction), and precomputed twiddle factors. Combined optimization techniques achieve 40-50% performance improvement relative to reference implementations, establishing performance upper bounds for ARM Cortex-M4 deployments. Measurements quantify achievable performance under aggressive optimization rather than baseline reference implementations.

6.1.1 Key Generation Performance

Table 3 presents key generation performance across ML-DSA parameter sets and ECDSA baseline.

Table 3: Key Generation Performance on ARM Cortex-M4 (168 MHz)

Scheme	Cycles	Time (ms)	Ops/sec	Overhead
ECDSA P-256	252,000	1.50	666.7	1.00×
ML-DSA-44	25,368,000	151.0	6.6	100.7×
ML-DSA-65	41,832,000	249.0	4.0	166.0×
ML-DSA-87	59,976,000	357.0	2.8	238.0×

Key generation performance impacts device provisioning workflows and key rotation strategies. The 100.7–238× computational overhead relative to ECDSA P-256 reflects lattice-based cryptographic complexity; optimization techniques provide limited improvement for key generation operations dominated by random polynomial sampling and matrix expansion rather than NTT arithmetic. ~~Measured execution times of 151–357 ms establish feasibility for infrequent~~
Measured execution times of 151–357 ms establish feasibility for infrequent key generation during device provisioning but prohibit high-frequency rotation strategies requiring sub-second key derivation.

6.1.2 Signature Generation Performance

Signature generation represents the primary performance bottleneck for publisher devices, directly impacting message throughput and system responsiveness. Table 4 quantifies signing performance across parameter sets and message sizes.

Table 4: Signature Generation Performance on ARM Cortex-M4 (168 MHz)

Scheme	Payload	Cycles	Time (ms)	Ops/sec	Overhead
ECDSA P-256	10 bytes	1,544,400	9.19	108.8	1.00×
ECDSA P-256	50 bytes	1,577,280	9.39	106.5	1.00×
ECDSA P-256	100 bytes	1,610,160	9.59	104.3	1.00×
ML-DSA-44	10 bytes	110,376,000	657.0	1.52	71.5×
ML-DSA-44	50 bytes	111,384,000	663.0	1.51	70.6×
ML-DSA-44	100 bytes	112,392,000	669.0	1.49	69.8×
ML-DSA-65	10 bytes	141,456,000	842.0	1.19	91.6×
ML-DSA-65	50 bytes	143,304,000	853.0	1.17	90.8×
ML-DSA-65	100 bytes	145,152,000	864.0	1.16	90.1×
ML-DSA-87	10 bytes	188,496,000	1,122.0	0.89	122.1×
ML-DSA-87	50 bytes	190,848,000	1,136.0	0.88	121.0×
ML-DSA-87	100 bytes	193,200,000	1,150.0	0.87	119.9×

Signature generation benefits from NTT optimization techniques, with each rejection sampling iteration requiring multiple forward and inverse NTT operations for polynomial arithmetic. ~~Optimized NTT implementations reduce per-iteration computational cost by 40–50%, translating to~~ ~~However, absolute signing latencies of 657–1,150 ms across parameter sets remain 70–122× slower than ECDSA d~~ NTT implementations reduce per-iteration computational cost by 40–50%, translating to proportional signing latency reduction. However, absolute signing latencies of 657–1,150 ms across parameter sets remain 70–122× slower than ECDSA despite optimization, establishing performance constraints for signature-based IoT authentication. Performance variability from rejection sampling (expected 3.85–5.1 iterations depending on parameter set) introduces timing non-determinism requiring worst-case latency analysis for real-time applications.

6.1.3 Signature Verification Performance

Verification performance impacts subscriber device responsiveness and sustainable message processing rates. Table 5 presents verification latency measurements across parameter sets.

Unlike signature generation, verification executes deterministically without rejection sampling, yielding predictable latency characteristics for real-time constraint analysis. Verification operations benefit from precomputed twiddle factors and optimized NTT implementations, ~~achieving 26–44× overhead relative to ECDSA compared to 71–122× overhead for signature generation.~~ ~~Absolute verification latencies of 416–717 ms exceed sub-second bounds for interactive IoT applications across all~~ 26–44× overhead relative to ECDSA compared to 71–122× overhead for signature generation. Absolute verification latencies of 416–717 ms exceed sub-second bounds for interactive IoT applications across all parameter sets, necessitating architectural accommodations for post-quantum authenticated messaging in latency-sensitive deployments.

6.2 Memory Utilization Analysis

Memory constraints represent critical deployment barriers for post-quantum cryptography on embedded devices. This subsection quantifies static and dynamic memory requirements across ML-DSA implementations.

Table 5: Signature Verification Performance on ARM Cortex-M4 (168 MHz)

Scheme	Payload	Cycles	Time (ms)	Ops/sec	Overhead
ECDSA P-256	10 bytes	2,688,000	16.00	62.5	1.00×
ECDSA P-256	50 bytes	2,721,600	16.20	61.7	1.00×
ECDSA P-256	100 bytes	2,755,200	16.40	61.0	1.00×
ML-DSA-44	10 bytes	69,888,000	416.0	2.40	26.0×
ML-DSA-44	50 bytes	70,560,000	420.0	2.38	25.9×
ML-DSA-44	100 bytes	71,232,000	424.0	2.36	25.9×
ML-DSA-65	10 bytes	88,704,000	528.0	1.89	33.0×
ML-DSA-65	50 bytes	89,544,000	533.0	1.88	32.9×
ML-DSA-65	100 bytes	90,384,000	538.0	1.86	32.8×
ML-DSA-87	10 bytes	118,104,000	703.0	1.42	43.9×
ML-DSA-87	50 bytes	119,280,000	710.0	1.41	43.8×
ML-DSA-87	100 bytes	120,456,000	717.0	1.39	43.7×

6.2.1 Static Memory Footprint

Table 6 presents code size and initialized data requirements for ML-DSA implementations compiled with -O3 optimization.

Table 6: Static Memory Footprint (Flash Memory Requirements)

Implementation	Code (KB)	Data (KB)	Total (KB)
ECDSA P-256	8.2	1.5	9.7
ML-DSA-44 only	32.4	4.8	37.2
ML-DSA-65 only	48.6	6.2	54.8
ML-DSA-87 only	65.8	8.1	73.9
ML-DSA All Sets	98.5	14.2	112.7

Static memory analysis establishes deployment feasibility for microcontrollers with limited Flash capacity in cost-constrained IoT applications.

6.2.2 Dynamic Memory Requirements

Table 7 quantifies runtime memory consumption including stack usage and key material storage.

Table 7: Dynamic Memory Requirements (SRAM Utilization)

Scheme	Stack Peak (KB)	Keys (KB)	Buffers (KB)	Total (KB)
ECDSA P-256	0.8	0.1	1.2	2.1
ML-DSA-44	6.4	3.8	12.5	22.7
ML-DSA-65	8.7	5.8	18.3	32.8
ML-DSA-87	11.2	7.3	24.6	43.1

Stack consumption measurements employ watermarking techniques quantifying worst-case memory usage during signature generation. Results establish minimum SRAM requirements for successful ML-DSA deployment on resource-constrained platforms.

6.3 Protocol-Level Overhead Assessment

MQTT integration overhead is quantified through message size increases, transmission latency, and throughput degradation relative to unsigned communications.

6.3.1 Message Size Overhead

Table 8 quantifies total message sizes for signed MQTT payloads across parameter sets and application payload sizes.

Table 8: MQTT Message Size Overhead (Signed vs Unsigned)

Scheme	Payload	Signed Size	Unsigned Size	Overhead Ratio
ECDSA P-256	10 bytes	82 bytes	18 bytes	4.6×
ECDSA P-256	50 bytes	122 bytes	58 bytes	2.1×
ECDSA P-256	100 bytes	172 bytes	108 bytes	1.6×
ML-DSA-44	10 bytes	2,438 bytes	18 bytes	135.4×
ML-DSA-44	50 bytes	2,478 bytes	58 bytes	42.7×
ML-DSA-44	100 bytes	2,528 bytes	108 bytes	23.4×
ML-DSA-65	10 bytes	3,327 bytes	18 bytes	184.8×
ML-DSA-65	50 bytes	3,367 bytes	58 bytes	58.1×
ML-DSA-65	100 bytes	3,417 bytes	108 bytes	31.6×
ML-DSA-87	10 bytes	4,645 bytes	18 bytes	258.1×
ML-DSA-87	50 bytes	4,685 bytes	58 bytes	80.8×
ML-DSA-87	100 bytes	4,735 bytes	108 bytes	43.8×

Message size overhead directly impacts network bandwidth consumption and transmission costs in cellular IoT deployments where data transfer incurs per-byte charges. Signature overhead analysis determines acceptability for bandwidth-constrained networks operating under kilobyte-per-day quotas.

6.3.2 End-to-End Latency Analysis

Table 9 presents complete publish-subscribe workflow latency measurements incorporating signature generation, network transmission, and verification operations.

Table 9: End-to-End MQTT Latency (Publisher to Verified Payload Delivery)

Scheme	Sign (ms)	Network (ms)	Verify (ms)	Total (ms)	Overhead
ECDSA P-256	9.4	28.5	16.2	54.1	1.00×
ML-DSA-44	663	31.2	420	1,114	20.6×
ML-DSA-65	853	33.8	533	1,420	26.2×
ML-DSA-87	1,136	37.4	710	1,883	34.8×

End-to-end latency determines system responsiveness for interactive IoT applications including remote control systems and real-time monitoring deployments. ML-DSA latency overhead is evaluated against application-specific timing constraints requiring sub-second response guarantees.

6.3.3 Sustainable Throughput Analysis

Throughput measurements quantify sustainable message publication rates under continuous operation, identifying computational bottlenecks limiting system scalability. Table 10 presents sustainable message rates across parameter sets and payload sizes.

Sustainable message rates range from 0.87 to 1.52 messages per second across ML-DSA parameter sets, compared to 104-109 messages per second for ECDSA. Signature generation constitutes the primary performance bottleneck, representing throughput degradation of 70-122× relative to classical signature schemes and directly limiting system scalability in high-frequency sensor network deployments.

Table 10: Sustainable MQTT Message Throughput (Messages per Second)

Scheme	Payload	Msgs/sec	Bottleneck	Overhead
ECDSA P-256	10 bytes	108.7	Signature gen.	1.00×
ECDSA P-256	50 bytes	106.4	Signature gen.	1.00×
ECDSA P-256	100 bytes	104.2	Signature gen.	1.00×
ML-DSA-44	10 bytes	1.52	Signature gen.	71.5×
ML-DSA-44	50 bytes	1.51	Signature gen.	70.5×
ML-DSA-44	100 bytes	1.49	Signature gen.	69.9×
ML-DSA-65	10 bytes	1.18	Signature gen.	92.1×
ML-DSA-65	50 bytes	1.17	Signature gen.	90.9×
ML-DSA-65	100 bytes	1.16	Signature gen.	89.8×
ML-DSA-87	10 bytes	0.89	Signature gen.	122.1×
ML-DSA-87	50 bytes	0.88	Signature gen.	120.9×
ML-DSA-87	100 bytes	0.87	Signature gen.	119.8×

6.4 Comparative Analysis and Trade-offs

Results across performance dimensions reveal trade-offs between post-quantum security levels and IoT system performance requirements.

6.4.1 Security-Performance Trade-off Analysis

The three ML-DSA parameter sets present quantifiable security-performance trade-offs across computational, memory, and protocol dimensions. ML-DSA-44 (NIST security level 2, AES-128 equivalent) achieves 657 ms signing latency, 416 ms verification latency, 2,420-byte signatures, and 22.7 KB total SRAM consumption. ML-DSA-65 (level 3, AES-192 equivalent) increases these metrics to 853 ms signing, 533 ms verification, 3,309-byte signatures, and 32.8 KB SRAM—representing 29.8% computational overhead, 36.8% signature size increase, and 44.5% memory increase relative to ML-DSA-44. ML-DSA-87 (level 5, AES-256 equivalent) further escalates to 1,150 ms signing, 717 ms verification, 4,627-byte signatures, and 43.1 KB SRAM—72.3% computational overhead, 91.2% signature size increase, and 89.9% memory increase versus ML-DSA-44.

Table 11 quantifies incremental security-performance trade-offs across parameter sets, normalizing all metrics to ML-DSA-44 baseline values.

Table 11: ML-DSA Parameter Set Trade-off Quantification (Normalized to ML-DSA-44)

Metric	ML-DSA-44	ML-DSA-65	ML-DSA-87
Security Level	2 (AES-128)	3 (AES-192)	5 (AES-256)
Signing Time	1.00×	1.30×	1.75×
Verification Time	1.00×	1.27×	1.72×
Signature Size	1.00×	1.37×	1.91×
Total SRAM	1.00×	1.45×	1.90×
Code Size	1.00×	1.47×	1.99×

For IoT deployments with 5–10 year operational lifetimes under current quantum computing development trajectories, NIST security level 2 (ML-DSA-44) provides adequate quantum resistance with minimal resource overhead. Long-term infrastructure deployments (20+ year operational lifetimes) requiring conservative security margins justify ML-DSA-87 despite 75–90% resource overhead increases. ML-DSA-65 occupies an intermediate position suitable for deployments requiring AES-192 equivalent security without the maximum resource costs of ML-DSA-87.

6.4.2 Deployment Feasibility Assessment

Deployment feasibility analysis evaluates ML-DSA applicability across representative IoT application categories based on measured performance characteristics.

High-throughput sensor networks requiring frequent authenticated message publication encounter computational bottlenecks limiting sustainable publication rates to 0.87-1.52 messages per second across ML-DSA parameter sets, compared to 104-109 messages per second for ECDSA. Deployments requiring publication frequencies exceeding 1 message per second necessitate architectural mitigation strategies: message aggregation combining multiple sensor readings into single signed payloads (amortizing signature overhead across N measurements at cost of N×sampling-interval latency increase), publisher-side caching reducing redundant signing of identical state values, or hybrid authentication employing ML-DSA signatures for security-critical messages (alerts, configuration changes) with MAC-based authentication for routine telemetry. For networks with 100 sensor nodes each publishing at 0.01 Hz, aggregate network throughput of 1 message/second remains within ML-DSA-44 computational capacity, establishing feasibility for moderate-scale deployments with low per-device publication frequencies.

Battery-powered devices operating under energy constraints require power consumption analysis of cryptographic operations impacting operational lifetime. Signature generation computational cost of 110.4 million cycles at 168 MHz consuming 657 milliseconds implies power draw of approximately 50 mW (assuming 3.3V supply at 15 mA typical active current for STM32F407VG). Energy per signature operation totals 32.9 mJ for ML-DSA-44. For devices transmitting 100 authenticated messages daily from 2,000 mAh batteries (3.3V nominal voltage, 23.8 kJ total energy), ML-DSA signature generation consumes 3.29 J daily (13.8% of total energy budget), rising to 115 J annually (4.7% of battery capacity assuming 50 mAh/year self-discharge). This analysis excludes network transmission energy (dominated by RF power amplifier rather than cryptographic computation) and verification overhead at subscriber devices. ML-DSA remains viable for battery-powered publishers with daily-scale message frequencies; higher publication rates require energy harvesting or mains power.

Real-time control systems with sub-second latency requirements encounter timing constraint violations from ML-DSA end-to-end authentication latency. Complete publish-subscribe workflows require 1.11-1.88 seconds (ML-DSA-44 through ML-DSA-87) compared to 54.1 milliseconds for ECDSA, exceeding sub-second responsiveness bounds for interactive control applications. Deployments requiring real-time authenticated command-response interactions necessitate architectural accommodations: asymmetric publisher-subscriber security models where publishers employ fast signing (ECDSA or MAC) with subscribers performing offline ML-DSA verification for audit trail generation, pre-signed command templates enabling instant transmission of pre-authenticated control messages with restricted command spaces, or hybrid cryptographic modes utilizing classical signatures for time-critical operations with periodic ML-DSA re-authentication for long-term security. Pure ML-DSA authentication remains suitable for monitoring applications tolerating multi-second latency but inappropriate for closed-loop control systems requiring deterministic sub-second response.

6.4.3 Optimization Opportunities

Several optimization strategies mitigate ML-DSA performance overhead in resource-constrained deployments, quantified through measured performance data.

Parameter set selection: Deployment-specific security requirement analysis enables ML-DSA-44 selection, reducing computational overhead by 42.2% (signing latency: 1,150 ms → 657 ms), signature size overhead by 47.7% (4,627 bytes → 2,420 bytes), and memory consumption by 47.3% (43.1 KB → 22.7 KB) relative to ML-DSA-87 while maintaining

NIST security level 2 quantum resistance. For IoT deployments with 5–10 year operational lifetimes, this represents an appropriate security-performance balance absent regulatory mandates requiring higher security levels.

Message aggregation: Batch signature generation over aggregated sensor readings amortizes cryptographic overhead across multiple measurements. Aggregating N measurements into a single signed payload reduces per-measurement signing overhead from 657 ms to $657/N$ ms at cost of $N\times$ sampling-interval latency increase. For $N = 10$ measurements at 1-second intervals, effective per-measurement overhead reduces from 657 ms to 65.7 ms (90.0% latency reduction) with 10-second aggregation latency. This strategy suits applications tolerating batched delivery (environmental monitoring, usage metering) but remains inappropriate for real-time event reporting requiring immediate transmission.

Hybrid authentication: Selective ML-DSA deployment for security-critical messages (firmware updates, configuration changes, alerts) combined with MAC-based authentication for routine telemetry reduces average authentication overhead while maintaining non-repudiation for critical operations. For deployment patterns comprising 90% routine telemetry (MAC authentication: 0.5 ms overhead) and 10% critical messages (ML-DSA-44: 657 ms overhead), weighted average overhead totals 66.2 ms compared to 657 ms for pure ML-DSA deployment (89.9% reduction). This approach requires security analysis of message classification policies and MAC key management infrastructure.

Hardware acceleration: Future ARM Cortex-M processors incorporating cryptographic acceleration extensions for NTT operations, modular arithmetic, or Keccak hashing could provide performance improvements. While current-generation IoT microcontrollers lack post-quantum cryptographic acceleration, industry roadmaps suggest specialized instruction set extensions may appear in 2027–2029 timeframes, potentially achieving 5–10× performance improvements through dedicated NTT hardware and reducing ML-DSA signing latency to sub-100-millisecond ranges. Until hardware acceleration availability, software optimization techniques establish performance upper bounds for ARM Cortex-M4 platforms.

This analysis provides guidance for IoT system designers evaluating post-quantum migration strategies through quantification of ML-DSA deployment costs and constraints.

7 Conclusion

References

[Ano24] Anonymous. Improved ML-DSA hardware implementation with first order masking countermeasure. Cryptology ePrint Archive, Paper 2024/1817, 2024.

[BZB$^+$22] Gustavo Banegas, Koen Zandberg, Emmanuel Baccelli, Adrian Herrmann, and Benjamin Smith. Quantum-resistant security for software updates on low-power networked embedded devices. In *Applied Cryptography and Network Security*, ACNS 2022. Springer, 2022. Also available at: https://eprint.iacr.org/2021/781.

[GMS19] Santosh Ghosh, Rafael Misoczki, and Manoj R. Sastry. Lightweight post-quantum-secure digital signature approach for IoT motes. Cryptology ePrint Archive, Paper 2019/122, 2019.

[HKS24] Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Multiplying polynomials without powerful multiplication instructions (long paper). Cryptology ePrint Archive, Paper 2024/1649, 2024.

[HKS25] Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Multiplying polynomials without powerful multiplication instructions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):160–202, 2025. Extended version of Barrett Multiplication for Dilithium on Embedded Devices.

[HW23] James Howe and Bas Westerbaan. Benchmarking and analysing the nist pqc lattice-based signature schemes standards on the arm cortex m7. In *Progress in Cryptology - AFRICACRYPT 2023*, volume 14064 of *Lecture Notes in Computer Science*. Springer, 2023.

[KKPY24] Matthias J. Kannwischer, Markus Krausz, Richard Petri, and Shang-Yi Yang. pqm4: Benchmarking NIST additional post-quantum signature schemes on microcontrollers. Cryptology ePrint Archive, Paper 2024/112, 2024.

[KMLO19] Ayesha Khalid, Sarah McCarthy, Weiqiang Liu, and Maire O'Neill. Lattice-based cryptography for IoT in a quantum world: Are we ready? Cryptology ePrint Archive, Paper 2019/681, 2019.

[KS25] YoungBeom Kim and Seog Chung Seo. An optimized instantiation of post-quantum MQTT protocol on 8-bit AVR sensor nodes. In *Proceedings of the 2025 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '25, pages 1–19. ACM, 2025.

[KSD20] Panos Kampanakis, Dimitrios Sikeridis, and Michael Devetsikiotis. Post-quantum authentication in tls 1.3: A performance study. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, 2020.

[Mar24] Dominik Marchsreiter. Towards quantum-safe blockchain: Exploration of pqc and public-key recovery on embedded systems. Cryptology ePrint Archive, Paper 2024/1178, 2024.

[NIS24] NIST. Fips 204: Module-lattice-based digital signature standard. Federal Information Processing Standards Publication, 2024. Available at: https://csrc.nist.gov/pubs/fips/204/final.

[SKD20] Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. Assessing the overhead of post-quantum cryptography in tls 1.3 and ssh. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 149–156. ACM, 2020.

[WYQ$^+$24] Yuxuan Wang, Jintong Yu, Shipei Qu, Xiaolin Zhang, Xiaowei Li, Chi Zhang, and Dawu Gu. Mind the faulty keccak: A practical fault injection attack scheme apply to all phases of ml-kem and ml-dsa. Cryptology ePrint Archive, Paper 2024/1522, 2024.