Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.082928

ARTICLE

Constant-Time Symmetry Exploitation for NTT Twiddle Factors in ML-KEM: A Unified Cost Model for Embedded Deployments

Xiaobing Liang[1], **Yu Qin**[1], **Shengdong Pan**[2] **and Liang Tan**[2,*]

[1]China Electric Power Research Institute Co., Ltd., Beijing, China
[2]College of Computer Sciences, Sichuan Normal University, Chengdu, China

*Corresponding Author: Liang Tan. Email: jkxy_tl@sicnu.edu.cn

Received: 27 March 2026; Accepted: 05 June 2026; Published: 23 July 2026

ABSTRACT: The standardization of the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM, FIPS 203) creates urgent demand for efficient post-quantum cryptography on resource-constrained devices. In such deployments, twiddle-factor management in the Number Theoretic Transform (NTT) induces a practical trade-off: full tables reduce latency but consume read-only memory (ROM), while on-the-fly generation reduces ROM but increases arithmetic cost. This paper makes two contributions. First, we present a constant-time half-table strategy (S_{half}) with branchless reconstruction logic and a formal indexing rule consistent with implementation. Second, we develop a Memory-Arithmetic Trade-off (MAT) model that unifies ROM, random-access memory (RAM), latency, energy, and side-channel risk into one device-aware objective. On ARM Cortex-M4, the proposed strategy reduces twiddle-factor ROM by 50% with a cycle overhead of about 3%–6% vs. full-table lookup in the same C framework. We additionally report cross-platform measurements on RISC-V and x86, where empirical best-strategy outcomes match MAT predictions. The resulting framework supports hardware-aware strategy selection with explicit assumptions on threat model and constant-time scope.

KEYWORDS: Post-quantum cryptography; ML-KEM; symmetry; unified cost model

1 Introduction

The transition of Post-Quantum Cryptography (PQC) from theory to standardization has culminated in the adoption of ML-KEM (FIPS 203) by NIST [1,2]. While ML-KEM provides a strong security foundation, deployment on resource-constrained edge devices such as embedded sensors and smart cards still faces major efficiency challenges. The Number Theoretic Transform (NTT), which accelerates polynomial multiplication in the ring $\mathcal{R}_q = Z_q[X]/(X^{256}+1)$, is the primary computational and storage bottleneck [3]. On low-power platforms such as ARM Cortex-M and RISC-V, NTT optimization is not only a speed problem; it is a joint trade-off among latency, memory footprint, energy, and side-channel risk.

Despite extensive research, practical ML-KEM implementations still face two unresolved problems. First, twiddle-factor management presents a difficult storage-latency dilemma. Full-table storage (S_{full}) maximizes speed but consumes substantial ROM (512 bytes), which is costly on low-end microcontrollers; in contrast, on-the-fly generation (S_{gen}) minimizes storage but introduces high computational overhead [4,5]. Although some FPGA accelerators exploit twiddle symmetry to reduce block RAM usage [6,7], these solutions are often hardware-specific and not directly portable to software [8,9]. In software, naive half-table realizations may introduce timing variability when implemented with conditional branches, especially if the

branch condition becomes secret-dependent in a broader implementation context; this motivates a portable branchless formulation with explicit proof obligations. Second, a unified quantitative evaluation framework is still missing. Prior studies are often platform-specific and fragmented, without a formal model to jointly evaluate ROM, RAM, ALU cycles, energy, and side-channel risk. Without such a model, system architects are forced to use ad-hoc trial-and-error rather than principled hardware-aware selection.

To address these gaps, this paper provides a structured and theory-backed examination of twiddle-factor symmetry for ML-KEM NTT. The main contributions are as follows:

- **A Constant-Time S_{half} Strategy with Formal Indexing Consistency:** We propose a branchless, symmetry-based reconstruction method that halves table storage to 256 bytes, and we provide a formal statement linking the implemented rule (k & $0x7F$ plus sign mask) to the target twiddle sequence.
- **A Device-Aware Memory-Arithmetic Trade-off (MAT) Model:** We establish a generalizable weighted-cost model that quantifies ROM, RAM, arithmetic latency, energy, and side-channel risk under an explicit threat model, and separates model definition from empirical calibration.
- **Empirical Evidence with Cross-Platform Validation:** We provide controlled measurements on ARM Cortex-M4, RISC-V, and x86, and show that empirical best-strategy outcomes align with MAT predictions under platform-specific constraints.

The remainder of this paper is organized as follows. Section 2 reviews related NTT optimization work. Section 3 introduces mathematical preliminaries. Section 4 presents the constant-time branchless half-table reconstruction strategy (S_{half}) with formal indexing consistency. Section 5 introduces the MAT model and robustness protocol. Section 6 reports detailed measurements on Cortex-M4, RISC-V, and x86 with leakage-evidence summaries. Section 7 concludes.

2 Related Work

Given the central role of ML-KEM (and its predecessor CRYSTALS-Kyber) in NIST post-quantum standardization, research on NTT optimization has grown rapidly in recent years. Both academia and industry have reported substantial progress across ARM Cortex-M4 microcontrollers, RISC-V embedded platforms, x86-64 high-performance systems, and FPGA accelerators. These studies have advanced lattice-based cryptography from theory to deployment, while also exposing unresolved gaps in unified storage-computation-energy-security trade-off modeling.

First, extensive optimization work has been reported on ARM Cortex-M4. Owing to its DSP features, Cortex-M4 is a primary reference platform for embedded PQC. Early work leveraged single-cycle DSP instructions (e.g., SMLAD) to accelerate butterfly operations [10], establishing the PQM4 framework as a benchmark [11]. Subsequent work introduced packed arithmetic and signed representations for memory-efficient high-speed implementations [12]. Later optimizations improved pipeline behavior through lazy reduction, instruction scheduling, and register allocation [13,14]. Although alternatives such as Plantard reduction reduce instruction counts [15], and masked implementations highlight side-channel risks in memory access [16], full-table twiddle storage still dominates software practice.

Second, RISC-V implementation paths combine software flexibility with modular ISA extensions. At the software level, researchers use the larger register file to cache coefficients and twiddle factors, reducing load/store overhead [17,18]. To overcome base-ISA limits, instruction extensions and accelerators (e.g., RANTT) compress NTT operations into fewer cycles [19,20]. With RISC-V vector extensions (RVV), recent studies focus on vectorized throughput scaling [21,22]. However, quantitative analysis of twiddle-storage area/energy cost and algebraic symmetry exploitation on baseline cores remains limited.

Third, on x86-64 platforms, optimization objectives shift from storage conservation to SIMD pipeline saturation. Seiler's AVX2 implementation established the fully vectorized Montgomery-reduction paradigm with layout-aware tables [23]. In AVX-512-era designs, implementations often store multiple table layouts to reduce permutation overhead, increasing constant-memory usage [24]. Because high-performance processors benefit from deep caches [25,26] and suffer branch-misprediction penalties, branch-based index strategies are typically avoided. This storage-for-throughput philosophy contrasts with embedded targets and motivates a unified cross-platform trade-off model.

Finally, FPGA acceleration studies often split between high-throughput BRAM-based full-table designs [27] and low-area logic-centric designs. To reduce BRAM cost, several works propose twiddle-factor generators (TFGs) that compute factors on the fly. Frameworks such as NTTGen and SAM further demonstrate systematic NTT-core generation with flexible area-throughput trade-offs [28]. Recent high-performance NTT hardware accelerators also target standardized lattice schemes such as ML-KEM and ML-DSA [29]. However, these hardware-specific optimizations are usually not directly transferable to serial software execution, where equivalent generation logic can be prohibitively expensive.

In summary, while the literature demonstrates significant progress in NTT optimization, two critical gaps remain. First, a standardized constant-time software method for symmetry exploitation is still under-documented. Naive half-table realizations may introduce timing variability when implemented with conditional branches, especially if the branch condition becomes secret-dependent in a broader implementation context; meanwhile, hardware-specific optimizations often lack algorithmic transparency for direct software porting. Second, a unified quantitative evaluation framework is absent. Current research relies on empirical, platform-specific heuristics, lacking a formal model to analytically trade off storage, computation, and energy across heterogeneous architectures. This work addresses these deficiencies by establishing the MAT model and proposing a constant-time branchless half-table strategy.

3 Preliminaries

3.1 Polynomial Rings and Parameterization in ML-KEM

ML-KEM operates over the structured polynomial ring

$$\mathcal{R}_q = Z_q[x]/(x^n + 1) \tag{1}$$

where $n = 256$ is the polynomial degree and $q = 3329$ is a specific prime modulus. Elements of this ring are polynomials of degree at most $n - 1$ with coefficients in the finite field Z_q. Operations in this ring, specifically polynomial multiplication, form the foundation of the Matrix-Vector multiplication

$$A\mathbf{s} + \mathbf{e} \tag{2}$$

which constitutes the core Learning-With-Errors (LWE) hardness assumption. The modulus $x^n + 1$ is chosen specifically to enable the use of NTT. Because $x^n \equiv -1 \pmod{x^n + 1}$, this ring exhibits the neg-acyclic property, which is crucial because it allows the convolution theorem to be applied efficiently, provided appropriate roots of unity exist in the base field.

3.2 Number Theoretic Transform (NTT)

NTT is a linear map that transforms a polynomial from the coefficient domain to the value domain (or frequency domain). For a vector of coefficients $\mathbf{a} = (a_0, \ldots, a_{n-1})$, NTT computes $\hat{\mathbf{a}} = \mathrm{NTT}(\mathbf{a})$, where the multiplication of two polynomials $\mathbf{a} \cdot \mathbf{b}$ in $\mathcal{R}_q$ corresponds to the pointwise multiplication $\hat{\mathbf{a}} \circ \hat{\mathbf{b}}$ in the NTT domain. The algorithm employed is a variant of the Fast Fourier Transform (FFT). Specifically, ML-KEM

uses a negacyclic NTT. The calculation is decomposed into $\log_2 n$ layers. In each layer i, coefficients are paired and processed via "butterfly" operations. A Cooley-Tukey butterfly takes two inputs u, v and a twiddle factor ζ to produce:

$$\begin{aligned} u' &= u + \zeta v \pmod{q} \\ v' &= u - \zeta v \pmod{q} \end{aligned} \tag{3}$$

The efficiency of NTT depends on rapid access to the twiddle-factor sequence ζ required at each layer.

4 A Constant-Time Twiddle Factor Reconstruction Method

Building upon the mathematical foundations in Section 3, we now address efficient twiddle-factor management. The key objective is to halve table storage while preserving deterministic control flow. In this section, we first define an implementation-consistent indexing rule (Section 4.1), then present a branchless accessor (Section 4.2), and finally discuss security scope and portability (Section 4.3).

4.1 Core Principle: Symmetry with Implementation-Consistent Indexing

For ML-KEM, $q = 3329$ and $q - 1 = 13 \cdot 256$, so primitive 256-th roots of unity exist in $\mathbf{Z}_q$. Let ζ be a primitive 256-th root. Then:

$$\zeta^{256} \equiv 1 \pmod{q}, \tag{4}$$

$$\zeta^{128} \equiv -1 \pmod{q}. \tag{5}$$

Hence, for any exponent $u \in [0, 127]$:

$$\zeta^{u+128} = \zeta^u \cdot \zeta^{128} \equiv -\zeta^u \pmod{q}. \tag{6}$$

This identity yields a direct half-table reconstruction rule. Define a canonical twiddle sequence $\Omega[k]$ ($k \in [0, 255]$) such that $\Omega[k + 128] \equiv -\Omega[k] \pmod{q}$. Store only the first half in a table $T[j] = \Omega[j]$ for $j \in [0, 127]$. For any $k \in [0, 255]$, write:

$$j = k \bmod 128, \qquad b = \left\lfloor \frac{k}{128} \right\rfloor \in \{0, 1\}. \tag{7}$$

Then:

$$\Omega[k] \equiv (-1)^b \cdot T[j] \pmod{q}. \tag{8}$$

In implementation, j is computed as k & $0x7F$ and b as $(k \gg 7)$ & 1, which is exactly equivalent for 8-bit bounded indices. For inverse NTT factors of the form ζ^{-u}, the schedule first maps exponents to their modulo-256 index and then calls the same accessor; therefore the previously cited $128 - u$ relation is a special case of this unified modulo-128 rule, not a competing rule.

Proposition 1 (Correctness of the branchless accessor): *Let* $k \in [0, 255]$, $j = k$ & $0x7F$, *and* $b = (k \gg 7)$ &1. *Define*

$$R(k) = \begin{cases} T[j], & b = 0, \\ q - T[j], & b = 1. \end{cases}$$

Then $R(k) \equiv \Omega[k] \pmod{q}$.

Proof: If $b = 0$, then $k < 128$ and $j = k$, so $R(k) = T[k] = \Omega[k]$. If $b = 1$, then $k = 128 + j$ and $R(k) = q - T[j] \equiv -T[j] \equiv -\Omega[j] \equiv \Omega[j + 128] = \Omega[k] \pmod{q}$. Thus both cases satisfy $R(k) \equiv \Omega[k]$, proving correctness. □

4.2 Key Mechanism: Branchless Implementation

To avoid timing variability from conditional branches, we implement the selection logic with bitwise masking, as shown in Algorithm 1.

Algorithm 1: Constant-time twiddle factor access function

Require: Index $k \in [0, 255]$, Half-table $T[0..127]$, Modulus q
Ensure: Reconstructed twiddle $\Omega[k]$
1: $index \leftarrow k$ & 0x7F {Equivalent to k mod 128}
2: $val \leftarrow T[index]$
3: $neg \leftarrow q - val$ {Compute modular negation}
4: $mask \leftarrow -((k \gg 7)\ \&\ 1)$ {0xFFFF if $k \geq 128$, else 0x0000}
5: $twiddle \leftarrow (val\ \&\ \sim mask)\ |\ (neg\ \&\ mask)$ {Select without branch}
6: **return** $twiddle$

The logical decomposition of Algorithm 1 is shown in Table 1. The access mechanism operates as follows:

- The `mask` variable is set to all ones (0xFFFF) when $k \geq 128$, otherwise it is all zeros.
- `val & ~mask`: Returns `val` when mask = 0, and returns 0 when mask = 0xFFFF.
- `neg & mask`: Returns 0 when mask = 0, and returns `neg` when mask = 0xFFFF.
- Consequently: when $k < 128$, the function returns `val`; when $k \geq 128$, the function returns `neg`, i.e., the sign-flipped symmetric value.

Table 1: Constant-time mask operation truth table.

Condition	k >> 7	(k >> 7) & 1	Mask	~Mask	Val & ~Mask	Neg & Mask	Return Value
$k < 128$	0	0	0	0xFFFF	val	0	val
$k \geq 128$	≥1	1	0xFFFF (−1)	0	0	neg	neg

This scheme saves 50% of twiddle-factor ROM (256 bytes vs. 512 bytes) and preserves deterministic control flow. The trade-off is a small fixed arithmetic overhead from modular negation and mask operations. The strategy is therefore attractive for embedded systems with strict ROM limits.

4.3 Implementation Analysis: Security Scope and Portability

To justify the design choice of our bitwise masking approach, we compare it against alternative methods for realizing the half-table strategy. As summarized in Table 2, three primary implementation paradigms exist:

1. **Branch-Based Implementation (Timing-Variant):** A naive approach uses conditional statements (if $k \geq 128$) to trigger modular negation. This may create data-dependent timing behavior if k is secret-dependent in a broader implementation context.

2. **Arithmetic-Shift Implementation (Portability Issues):** Some "branchles" optimizations rely on the specific behavior of arithmetic right shifts (>>) to generate masks (e.g., $mask = k >> 15$). However, the C standard leaves the behavior of right-shifting negative signed integers implementation-defined. This leads to portability hazards, where the code may function correctly on one compiler/architecture (e.g., GCC on x86) but fail on another (e.g., specific DSP compilers), limiting its utility in heterogeneous platforms.
3. **Proposed Bitwise Masking (Constant-Time Coding Pattern):** Our strategy (Algorithm 1) utilizes deterministic logic operations (&, |, ~) combined with subtraction. This design achieves two engineering goals:
   - Constant-time coding discipline: the high-level instruction path is input-independent (subject to compiler/backend realization).
   - Strict portability: It relies solely on standard boolean logic defined consistently across all C99-compliant compilers and architectures.

Table 2: Comparison of three half-table access strategies across five dimensions.

Dimension	Branch (if ($k \geq 128$))	Branchless Arithmetic	Branchless Mask Selection (Proposed)
ALU (Latency)	**Non-deterministic.** Very fast (1–2 cycles) on correct prediction; very slow (10–20 cycles, pipeline flush) on misprediction.	**Deterministic and low.** Fixed sequence of arithmetic/bit operations.	**Deterministic and low.** Fixed sequence of bit operations. Offers the optimal Worst-Case Execution Time (WCET).
ROM (Code Size)	**Smallest.** Typically a few compare and jump instructions.	**Small.** Requires arithmetic shift and arithmetic/logical operation instructions.	**Moderate.** Requires instructions for mask generation and bit manipulation. Slightly larger than A and B.
RAM (Runtime)	Low. Only a few temporary variables.	Low. Same as Method A.	Low. Same as Methods A and B.
Energy	**Non-deterministic, unpredictable.** Branch mispredictions cause significant energy spikes.	**Deterministic and predictable.** Regular operations lead to stable power consumption, beneficial for power management.	**Deterministic and predictable.** Similar to Method B, most stable in energy consumption.
Security	**Higher timing-risk profile.** May create timing variability if branch condition is secret-dependent.	**Lower timing-risk profile.** Data-independent instruction path at source level (subject to compiler realization).	**Lower timing-risk profile.** Data-independent path with explicit mask-selection discipline.

By avoiding explicit branching and implementation-defined signed shifts, the proposed method provides a robust baseline for secure engineering practice. We inspected the compiled binary

(using `arm-none-eabi-objdump`) for the ARM Cortex-M4 target. The masking logic compiles strictly to deterministic arithmetic and bitwise instructions (e.g., AND, EOR, RSB), with no data-dependent branching (B or IT blocks) generated by the compiler. Full side-channel security still requires empirical leakage assessment under an explicit threat model, which we discuss in Section 6.

5 MAT: Unified and Generalizable Cost Model

Having presented the constant-time implementation of the half-table strategy (S_{half}) in Section 4, a fundamental engineering question remains: how does one rigorously determine the optimal strategy for a specific hardware target? While qualitative comparisons suggest that S_{half} offers a balance, relying on intuition or ad-hoc benchmarks is insufficient for heterogeneous platforms where constraints on ALU, ROM, RAM, energy vary drastically and security. To address this, we propose the Memory-Arithmetic Trade-off (MAT) Model, a unified analytical framework designed to quantify the decision-making process. In this section, we first establish a formal representation of the strategy space (Section 5.1), then derive multi-dimensional cost functions covering storage, latency, energy, and security (Section 5.2). Finally, we introduce device-sensitivity parameters to formulate a platform-aware optimization objective (Section 5.3), enabling precise, mathematical strategy selection. The proposed MAT model is positioned as a device-aware engineering cost model for a finite strategy set, and its purpose is to provide an interpretable strategy-selection framework.

5.1 Unified Representation of Twiddle-Factor Strategies

To enable rigorous comparison, we first formalize the implementation variants as a set $\mathcal{S}$:

$$\mathcal{S} = \{S_{\text{full}},\ S_{\text{half}},\ S_{\text{gen}}\}. \tag{9}$$

where S_{full} denotes the full-table storage strategy, S_{half} denotes the proposed symmetry-based half-table storage strategy, and S_{gen} denotes the on-the-fly generation strategy.

Each strategy $S \in \mathcal{S}$ is mapped to a multi-dimensional cost vector $C(S)$ that captures its resource footprint:

$$C(S) = (C_{\text{ROM}}(S),\ C_{\text{RAM}}(S),\ C_{\text{ALU}}(S),\ C_{\text{ene}}(S), C_{\text{sec}}(S)) \tag{10}$$

Here, the components represent ROM usage (bytes), RAM usage (bytes), ALU latency (cycles), Energy consumption (Joules), and Security cost (side-channel risk level) respectively. This abstraction provides the foundation for the quantitative trade-off analysis in the subsequent sections.

5.2 Formal Multi-Dimensional Cost Model

To quantify the strategies in $\mathcal{S}$, we define explicit cost functions for ROM, RAM, ALU latency, energy, and security.

Firstly, in order to quantify the usage of read-only memory for different strategies, we define a ROM cost function to capture the static storage requirements. The ROM cost of S is defined as:

$$C_{\text{ROM}}(S) = \begin{cases} n \cdot w, & S = S_{\text{full}}, \\ \dfrac{n}{2} \cdot w, & S = S_{\text{half}}, \\ 0, & S = S_{\text{gen}} \end{cases} \tag{11}$$

where n is the total twiddle factors count, w is is the word size, $C_{\mathrm{ROM}}(S_{\mathrm{full}}) = n \cdot w$ represents that the ROM cost $n \cdot w$ corresponds to storing the complete, precomputed table for S_{full}, which is characterized by the highest ROM usage but provides the fastest access speed, making it suitable for applications that require high performance and sufficient memory. $C_{\mathrm{ROM}}(S_{\mathrm{half}}) = \frac{n}{2} \cdot w$ represents that the term $\frac{n}{2} \cdot w$ reflects the 50% storage reduction achieved by exploiting the symmetry $\zeta^{128} \equiv -1$ for S_{half}, which is characterized by that the ROM occupation is halved, but additional arithmetic operations are required to achieve a balance between storage and calculation. $C_{\mathrm{ROM}}(S_{\mathrm{gen}}) = 0$ indicates that the ROM cost is 0, as no table is stored and is generated by calculation each time it is needed for S_{gen}.

Secondly, to quantify the usage of RAM for different strategies, we define a RAM cost function to capture the working memory requirements. The RAM cost of S is defined as:

$$C_{\mathrm{RAM}}(S) = \begin{cases} 0, & S = S_{\mathrm{full}}, \\ m \cdot w, & S = S_{\mathrm{half}}, \\ 2m \cdot w, & S = S_{\mathrm{gen}} \end{cases} \tag{12}$$

where m represents the total temporary variables count. $C_{\mathrm{RAM}}(S_{\mathrm{full}}) = 0$ indicates requiring zero additional runtime RAM for S_{full}, which requires the minimum working memory and only the target memory location. $C_{\mathrm{RAM}}(S_{\mathrm{half}}) = m \cdot w$ indicates $m \cdot w$ accounts for the RAM need to store the mask and negated value for S_{half}. $C_{\mathrm{RAM}}(S_{\mathrm{gen}}) = 2m \cdot w$ indicates that the RAM cost $2m \cdot w$ is maximized to maintain the state variables for S_{gen}.

Thirdly, to quantify the computational latency of different strategies, we define a ALU latency cost function to capture the time performance. The ALU latency cost of S is defined as:

$$C_{\mathrm{ALU}}(S) = \begin{cases} t_{\mathrm{mem}}, & S = S_{\mathrm{full}}, \\ t_{\mathrm{mem}} + t_{\mathrm{add}}, & S = S_{\mathrm{half}}, \\ t_{\mathrm{trig}} + 2t_{\mathrm{mult}} + t_{\mathrm{add}}, & S = S_{\mathrm{gen}}. \end{cases} \tag{13}$$

Here, t_{op} denotes the latency of memory access (mem), addition (add), multiplication ($mult$), and trigonometric generation ($trig$). $C_{\mathrm{lat}}(S_{\mathrm{full}}) = t_{\mathrm{mem}}$ represents that the cost is bounded strictly by t_{mem} for S_{full}, which represents a pure memory lookup with no arithmetic overhead. $C_{\mathrm{lat}}(S_{\mathrm{half}}) = t_{\mathrm{mem}} + t_{\mathrm{add}}$ represents a hybrid cost for S_{half}, which fetches a base factor (t_{mem}) followed by a modular subtraction (t_{add}) to reconstruct the symmetric entry. $C_{\mathrm{lat}}(S_{\mathrm{gen}}) = t_{\mathrm{trig}} + 2t_{\mathrm{mult}} + t_{\mathrm{add}}$ represents that the term $t_{\mathrm{trig}} + 2t_{\mathrm{mult}} + t_{\mathrm{add}}$ captures the high algorithmic overhead of synthesizing factors on-the-fly for S_{gen}, which involves complex modular multiplications and additions.

Fourthly, to quantify the energy consumption for different strategies, we define an energy cost function to capture the power consumption. The energy cost of S is defined as:

$$C_{\mathrm{ene}}(S) = \begin{cases} e_{\mathrm{mem}}, & S = S_{\mathrm{full}}, \\ e_{\mathrm{mem}} + e_{\mathrm{add}}, & S = S_{\mathrm{half}}, \\ e_{\mathrm{trig}} + 2e_{\mathrm{mult}} + e_{\mathrm{add}}, & S = S_{\mathrm{gen}}. \end{cases} \tag{14}$$

Here, e_{op} denotes the latency of memory access (mem), addition (add), multiplication ($mult$), and trigonometric generation ($trig$). $C_{\mathrm{ene}}(S_{\mathrm{full}}) = e_{\mathrm{mem}}$ represents that energy consumption is dominated by the memory subsystem for S_{full}, while arithmetic energy is zero, static leakage from the large ROM and

dynamic energy from frequent read operations constitute the total cost. $C_{\text{ene}}(S_{\text{half}}) = e_{\text{mem}} + e_{\text{add}}$ represents that balances memory and computation. It reduces static ROM leakage by half compared to S_{full}, replacing the saved memory energy with the typically lower energy cost of simple arithmetic operations (negation). $C_{\text{ene}}(S_{\text{gen}}) = e_{\text{trig}} + 2e_{\text{mult}} + e_{\text{add}}$ represents eliminating static ROM energy entirely but incurs the highest dynamic power consumption due to the continuous activity of the multiplier and ALU units required for complex factor generation.

Finally, to quantify the security-related implementation risk of different strategies, we define a security cost function to capture residual leakage risk under the stated threat model. This term is defined as a residual leakage-risk penalty. A larger value means higher residual risk, and a smaller value means lower residual risk. The security cost of S is defined as:

$$C_{\text{sec}}(S) = \begin{cases} \rho_{\text{branch}}, & S = S_{\text{full}}, \\ \rho_{\text{half}}, & S = S_{\text{half}}, \\ \rho_{\text{gen}}, & S = S_{\text{gen}}. \end{cases} \tag{15}$$

Here, $1 \geq \rho_{\text{branch}} > \rho_{\text{half}}, \rho_{\text{gen}} \geq 0$. The symbol ρ denotes a threat-model-dependent residual leakage-risk score rather than a binary "secure/insecure" label. This definition keeps the model direction consistent: lower C_{sec} means lower residual side-channel risk. Here, ρ_{branch}, ρ_{half}, and ρ_{gen} denote residual leakage-risk penalties associated with the full-table, half-table, and on-the-fly generation strategies, respectively. The ordering $\rho_{\text{branch}} > \rho_{\text{half}}, \rho_{\text{gen}}$ reflects that branch-based or input-dependent access patterns may lead to higher residual timing/cache risk under the stated threat model, whereas branchless reconstruction and generation-based strategies are assigned lower residual risk. These parameters are not interpreted as absolute security guarantees; they provide a model-level risk penalty used for comparative strategy selection.

5.3 Device-Aware Optimization Framework

To generalize the evaluation across heterogeneous platforms (e.g., IoT microcontrollers vs. high-performance servers), we introduce a parameterized optimization framework. This maps the absolute costs derived in Section 5.2 to a relative, platform-specific utility value.

1) Hardware Sensitivity Vector ($\boldsymbol{\theta}$). We define a device-class sensitivity vector θ to capture the inherent resource constraints of the target hardware as:

$$\theta = (\theta_{\text{ROM}}, \theta_{\text{ALU}}, \theta_{\text{RAM}}, \theta_{\text{ene}}, \theta_{\text{sec}}) \tag{16}$$

Here, each component $\theta_k \in [0,1]$ quantifies the scarcity or cost of a specific resource on a given device. $\theta_{ROM}/\theta_{RAM}$ represents memory scarcity. A high value (e.g., ≈1.0) characterizes deeply embedded 8-bit/16-bit MCUs where Flash/SRAM is the primary bottleneck. θ_{ALU} represents the "penalt" of computation. High values apply to simple cores without hardware multipliers, while low values apply to DSP-enhanced cores (like Cortex-M4). θ_{ene} is Quantifying energy sensitivity, critical for battery-powered IoT nodes but negligible for mains-powered servers. θ_{sec} reflects the threat model. High values indicate security-critical deployments (e.g., smart cards) requiring strict side-channel resistance.

2) Hardware Sensitivity Weights function ($\boldsymbol{\omega(\theta)}$). We define the hardware sensitivity weights function $\omega(\theta)$ which represent the relative importance of different cost dimensions under specific hardware platform parameters θ, and maps objective hardware parameters to subjective optimization weights. It defines 'how to balance based on hardware characteristics'. $\omega(\theta)$ is a set, it is defined as follows.

$$\omega(\theta) = \{\omega_{ROM}(\theta), \omega_{RAM}(\theta), \omega_{ALU}(\theta), \omega_{ene}(\theta), \omega_{sec}(\theta)\} \tag{17}$$

where $\omega_{ROM}(\theta)$ is a storage space weight used to measure the scarcity of ROM/Flash storage space. Its high value scenarios are in 8-bit/16 bit MCUs and embedded devices with Flash < 64 KB, while low value scenarios are in x86 servers and platforms with ample storage. The greater the weight, the more inclined one is to choose strategies with lower storage costs. $\omega_{RAM}(\theta)$ is the weight of running memory, used to measure the tightness of RAM resources. Its high value scenarios are deeply embedded devices with only a few KB of SRAM, while low value scenarios are desktop/server platforms with large capacity RAM. This weight affects optimization choices that affect runtime memory allocation. $\omega_{ALU}(\theta)$ is the computational power weight used to measure the relative cost of ALU computation operations. Its high value scenarios are low-power MCUs (slow computation, high energy consumption), while low value scenarios are high-performance CPUs (fast computation, low energy consumption). The smaller the weight, the more tolerant it is to computationally intensive strategies. $\omega_{ene}(\theta)$ is an energy sensitivity weight used to measure the importance of energy consumption. Its high value scenarios include battery powered IoT devices and energy harvesting systems; while low value scenarios refer to plug-in devices and platforms with stable power sources. This weight affects the balance of energy efficiency. $\omega_{sec}(\theta)$ is a security requirement weight used to measure the importance of side channel security. Its high value scenarios refer to payment terminals, military equipment, and security critical systems, while low value scenarios refer to ordinary devices for non sensitive data processing. This weight determines whether to implement with constant time.

Secondly, how is $\omega(\theta)$ designed? $\omega(\theta)$ is a conversion function which accepts objective hardware parameters θ and outputs the subjective importance (weight) assigned to each dimension during optimization. Its design embodies the optimization philosophy of 'where scarce, save'. In practical applications, the specific form of $\omega_{sec}(\theta)$ is flexible, it is likely to be a more complex mapping, even based on empirical data (benchmark testing) lookup tables, but its core idea remains unchanged: to adapt the optimization strategy to the characteristic parameters of the hardware platform. For example, we can design a simple linear function as follows.

$$\begin{cases} \omega_{ROM}(\theta) = \alpha_{ROM} \cdot \theta_{ROM} \\ \omega_{ALU}(\theta) = \alpha_{ALU} \cdot \theta_{ALU} \\ \omega_{RAM}(\theta) = \alpha_{RAM} \cdot \theta_{RAM} \\ \omega_{ene}(\theta) = \alpha_{ene} \cdot \theta_{ene} \\ \omega_{sec}(\theta) = \alpha_{sec} \cdot \theta_{sec} \end{cases} \tag{18}$$

where α is the scaling factor. In this linear model, empirical coefficients are used to adjust the importance of different dimensions, and their values are set by the designer based on overall optimization objectives such as "extreme space saving" and "ultimate performance". We use the linear weighting function in this model as an interpretable engineering instance, and the stability of the resulting recommendation is evaluated through exhaustive weight perturbation and sensitivity analysis in Section 6.

3) Weighted Objective Function ($\Phi(S;\theta)$). To select the optimal strategy, we formulate the problem as the minimization of a total cost function $\Phi(S;\theta)$. We map the sensitivity parameters θ to normalized optimization weights $\omega(\theta)$:

$$\Phi(S;\theta) = \sum_{k \in \mathcal{K}} \omega_k(\theta) \cdot C_k(S) \tag{19}$$

where $\mathcal{K} = \{ROM, RAM, ALU, ene, sec\}$. The weights $\omega_k(\theta)$ are typically derived via a linear mapping $\omega_k \propto \alpha_k \cdot \theta_k$, where α_k are system-level scaling factors. For example, on ultra-low-cost 8-bit microcontrollers, the dominating weight is often $w_{\mathrm{ROM}}(\theta)$, because flash reads are expensive and ROM capacity is

extremely limited. In contrast, on latency-sensitive ARM Cortex-M designs, $w_{\text{lat}}(\theta)$ typically dominates, whereas DSP-oriented or RISC-V vector platforms assign comparatively small weights to arithmetic costs, favoring on-the-fly twiddle generation.

4) Optimal Strategy Selection. The optimal twiddle-factor management strategy S^* for a specific hardware platform is formally defined as the strategy that minimizes the hardware-weighted objective:

$$S^*(\theta) = \arg\min_{S \in \mathcal{S}} \Phi(S; \theta) \tag{20}$$

This scalar objective is used for hardware-aware strategy selection over the finite strategy set $\mathcal{S}$. In the multi-objective sense, Pareto dominance is interpreted in the standard way: a strategy S_a dominates S_b only if $C_k(S_a) \leq C_k(S_b)$ for all cost dimensions and the inequality is strict for at least one dimension. Therefore, this paper reports S_{half} as a Pareto-efficient embedded compromise or knee point under the tested cost model, rather than as an unconditional globally Pareto-optimal solution. This formulation allows the MAT-Model to dynamically predict the best strategy: favoring S_{half} on constrained IoT devices (high θ_{ROM}, high θ_{sec}) while potentially favoring S_{full} on high-performance servers (low θ_{ROM}, low θ_{ALU}). This formulation highlights a key observation: unlike fixed implementation heuristics, the optimal strategy is not universal but varies systematically with hardware characteristics. The parameterized objective function thus provides a principled and portable method for selecting between full-table storage, half-table storage with modular negation, and on-the-fly generation.

In subsequent sections, we validate this optimization framework empirically across diverse platforms, demonstrating that hardware-weighted cost modeling accurately predicts the most efficient strategy in practice.

6 Experimental Results and Analysis

This section evaluates the practical viability of twiddle-factor symmetry exploitation in ML-KEM. We evaluate the strategies across all three platforms (ARM Cortex-M4, RISC-V, and x86-64) to compare S_{full}, S_{half}, and S_{gen} under a unified C framework. This provides comprehensive cross-platform empirical evidence to validate the MAT model's predictions.

6.1 Experimental Setup

To ensure a rigorous evaluation that is both internally consistent for model validation and comparable to the state-of-the-art, we established a unified experimental framework. Our methodology isolates the algorithmic impact of twiddle-factor strategies while benchmarking their absolute performance against existing high-speed and masked implementations.

1) **Target Platforms and Compilation Environments.** We selected three platforms representing embedded and high-performance settings. In this revision, all three platforms (P1/P2/P3) are measured under a unified benchmark configuration. Table 3 lists hardware specifications and toolchains.
   - **Platform P1 (Embedded IoT):** STMicroelectronics STM32F407 (ARM Cortex-M4) operating at 168 MHz with 192 KB SRAM. As the primary embedded PQC reference, this platform is our main target for memory-constrained trade-off analysis. Code was compiled using arm-none-eabi-gcc (v10.3) with `-O3`.
   - **Platform P2 (Open ISA Edge):** SiFive HiFive1 Rev B (RISC-V FE310-G002) operating at 320 MHz. We used riscv64-unknown-elf-gcc (v10.2) with `-O3`.
   - **Platform P3 (High-Performance):** Intel Core i7-12700K (x86-64 AVX2). Code was compiled using gcc (v9.4) with `-O3`.

2) **Implementation Strategies.** To satisfy the dual objectives of theoretical model validation and practical benchmarking, we classify our implementations into two categories:

   a. **Controlled Strategy Variants (Internal Comparison).** To validate the MAT-Model (Section 5), we implemented the three core strategies within a unified C-language framework derived from the pqm4 and pqriscv libraries. This ensures that observed performance differences arise solely from the storage strategy rather than disparate assembly optimizations.
      - S_{full} (Baseline): Uses the standard precomputed lookup table (512 bytes). The array structure remains unchanged from the reference implementation.
      - S_{half} (Proposed): Reduces the table to 128 entries (256 bytes). Accessor macros are replaced with our proposed inline constant-time reconstruction logic (Algorithm 1) to handle indices $k \geq 128$ without branching.
      - S_{gen} (on-the-fly): Removes the data table entirely. Twiddle factors are computed dynamically via a next_zeta() state machine added to the NTT loop.

   b. **Reference Baselines (State-of-the-Art Comparison):** To assess practical competitiveness, we benchmark our controlled variants against established high-performance implementations:
      - PQM4 (Botros et al. [12]): Highly optimized assembly implementation, used here as a practical speed reference for S_{full}.
      - Optimized Stack (Abdulrahman et al. [13]): A further refined assembly implementation focusing on register allocation and instruction scheduling.
      - Masked Proxies (Kannwischer, [14]): To evaluate security-performance trade-offs, we reference standard masking overheads (typically 30%–100%) reported in [14] as a baseline for achieving side-channel resistance.

3) **Measurement Methodology.** All strategies on all three platforms were evaluated using the same test harness to ensure fairness.
   - Cycle Counts (ALU): Measured by hardware performance counters with interrupts disabled.
   - Memory Footprint: Compiled with GCC `-O3`. ROM (`.text +.rodata`) and stack usage were extracted via the `size` utility and static analysis.
   - Energy Consumption: For P1 and P2, measured directly via external instrumentation (ST-Link V3 power bridge with a shunt resistor). For P3 (x86-64), energy was estimated using Intel RAPL (Running Average Power Limit) software counters.
   - Security Analysis: We report constant-time coding properties, static branchlessness checks, and an empirical leakage-evaluation protocol (timing traces and TVLA) rather than a blanket security proof.

Table 3: Hardware and software configuration.

Platform	Core Architecture	Memory	Compiler & Flags
P1	ARM Cortex-M4 (168 MHz)	192 KB/1 MB	arm-none-eabi-gcc v10.3, `-O3`
P2	RISC-V RV32IMAC (320 MHz)	16 KB/32 MB	riscv64-unknown-elf-gcc v10.2, `-O3`
P3	x86-64 (AVX2) (3.6 GHz)	64 GB/N/A	gcc v9.4, `-O3 -mavx2`

Assembly-level branchlessness check. We additionally inspected the generated Cortex-M4 assembly using `arm-none-eabi-objdump -d` under the tested `-O3` configuration. The twiddle accessor was compiled into a fixed sequence of shift, mask, subtraction, and logical instructions, and no data-dependent

conditional branch was observed for the selection path. This check supports the constant-time coding claim for the tested compiler/backend configuration, while not replacing empirical leakage evaluation.

4) Threat Model and Verification Scope. We adopt a first-order timing/power leakage threat model in which an attacker can collect repeated traces of the same operation and perform statistical distinguishers. The goal of this work is to reduce implementation-level leakage risk in twiddle-factor access by enforcing branchless control flow. Therefore, statements in this section are interpreted as follows: (i) *constant-time coding property* at source/assembly level, and (ii) *empirical leakage evidence* when timing traces or TVLA are available. We do not claim that branchless twiddle access alone yields complete side-channel security against all leakage classes.

6.2 Comprehensive Five-Dimensional Strategy Comparison

This subsection reports detailed Cortex-M4 measurements. We first compare the three controlled variants (S_{full}, S_{half}, S_{gen}) in one C framework, then position results against external assembly baselines. The comparison spans latency, storage, memory, energy, and side-channel-risk indicators.

1) Computational Latency Analysis

We begin by evaluating execution performance (cycles per NTT) on the Cortex-M4 platform, as shown in Fig. 1.

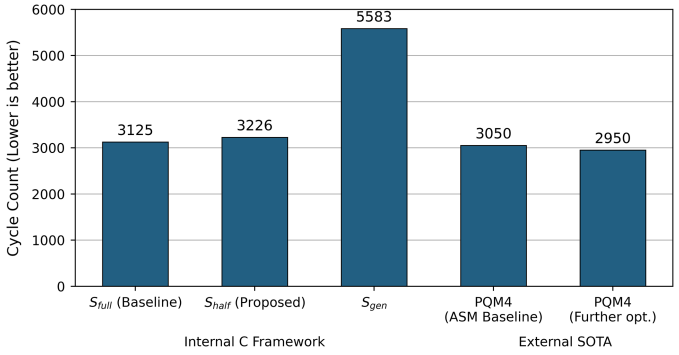

Figure 1: Implementation cycle count comparison for NTT of ML-KEM.

Data in Table 4 shows that, within the same C framework, S_{half} incurs about 3.2% overhead (3226 vs. 3125 cycles) over S_{full}. This indicates that branchless reconstruction adds only a small fixed arithmetic cost. Against highly optimized assembly baselines, the C implementation remains close (about 5.8% over PQM4 baseline), which is sufficient for controlled model validation.

Table 4: Performance comparison (Cortex-M4 platform).

Implementation	Strategy Class	Cycle Count (per NTT)
S_{full}	S_{full} (Pure C, same framework)	3125 cycles
S_{half} (proposed)	S_{half} (Pure C, same framework)	3226 cycles
S_{gen}	S_{gen} (Pure C, same framework)	5583 cycles
PQM4 [12] (Baseline)	S_{full} (Highly Opt. ASM)	~3050 cycles
PQM4 [13]	S_{full} (Further opt.)	~2950 cycles

2) Storage Footprint Analysis (ROM & RAM)

As depicted in Fig. 2 and Table 5, S_{half} achieves its primary goal: a 50% reduction (256 bytes) in `.rodata` relative to both the internal S_{full} baseline and typical external SOTA implementations. This corresponds to an overall ROM reduction of about 8%. Although S_{gen} removes the table entirely, it introduces notable code inflation (about 330 bytes) due to state-machine logic. Consequently, its total footprint (2120 bytes) provides only limited benefit over S_{half} (2140 bytes), supporting S_{half} as a balanced software strategy.

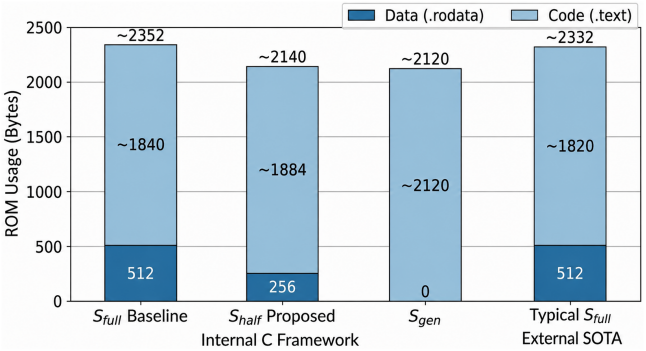

Figure 2: ROM footprint analysis.

Table 5: ROM footprint comparison (Unit: bytes).

Implementation	.text (Code)	.rodata (Data)	Total
S_{full} [12,14]	~1820	512	~2332
S_{full}	1840	512	2352
S_{half} (proposed)	1884	256	2140
S_{gen}	2120	0	2120

Table 6 and Fig. 3 analyze runtime stack usage. Internally, S_{half} introduces a modest fixed stack increase (+16 B). Compared with typical masked implementations requiring substantially larger state, this overhead remains small for embedded deployment. The bold percentages in Table 6 indicate the relative increase in stack usage compared with the S_{full} baseline of approximately 80 B. Specifically, S_{half} requires an additional 16 B (+20%) for the branchless reconstruction variables, whereas S_{gen} requires approximately 48 B more (+60%) to maintain the factor-generation state.

Table 6: Runtime stack usage and RAM profile analysis.

Strategy	Key Runtime Variables	Stack Usage
S_{full} (Baseline)	loop counters, butterfly intermediates.	~80 B
S_{half} (proposed)	adds `val`, `neg`, `mask` for branchless logic.	~96 B **(+20%)**
S_{gen}	adds state variables for factor generation.	~128 B **(+60%)**
S_{full} + Masking	adds mask arrays and RNG state.	>>128 B

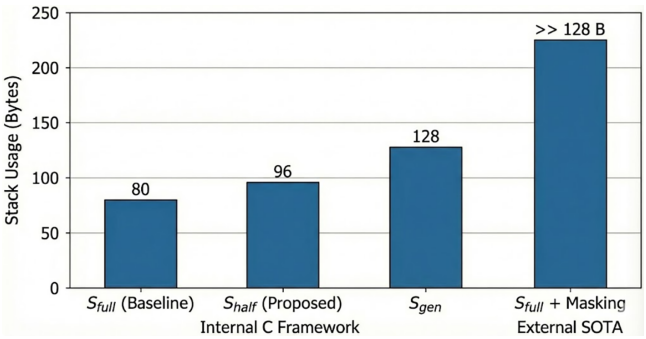

Figure 3: RAM footprint analysis.

3) Energy Consumption Analysis

Fig. 4 and Table 7 show that energy trends follow runtime trends. On Cortex-M4, S_{half} adds about 3.3% energy over S_{full}, while S_{gen} adds 77.6%. The masking entry is an inferred reference point and is used only for directional comparison.

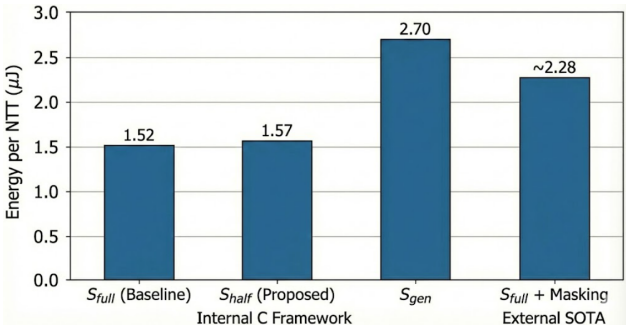

Figure 4: Energy consumption comparison.

Table 7: Energy consumption comparison (Cortex-M4).

Implementation Strategy	Energy Per NTT	Key Characteristics & Overhead Source
S_{full}	1.52 μJ (Baseline)	Pure table lookup. Energy from memory access.
S_{half} (proposed)	1.57 μJ (+3.3%)	Table lookup + bitwise ops. Constant-time coding pattern with low fixed overhead.
S_{gen}	2.70 μJ (+77.6%)	Energy dominated by numerous modular multiplications.
Typical S_{full} + Masking	~2.28 μJ (+50%)	Table lookup + mask generation & manipulation overhead.

*Note: The energy value for masking is inferred from a conservative 50% performance-overhead assumption, based on reported ranges of 30%–100%.

4) Security and Side-Channel Risk Comparison

As summarized in Fig. 5 and Table 8, S_{half} provides a favorable compromise: lower timing/cache risk indicators than branch-based lookup at far lower overhead than full masking or on-the-fly generation. These results should be interpreted as implementation-risk evidence under the stated threat model, not as a complete side-channel proof.

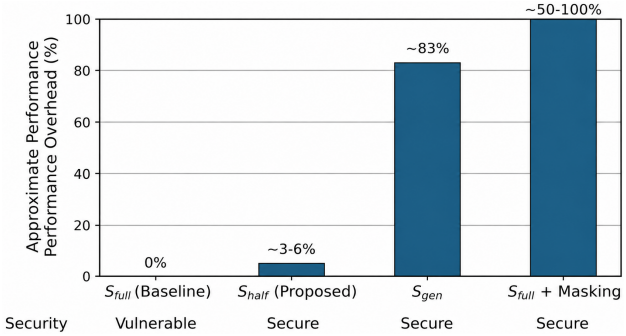

Figure 5: Approximate side-channel risk vs. overhead comparison.

Table 8: Side-channel risk mechanism and overhead comparison.

Strategy	Mechanism	Timing/Cache Risk (This Work Scope)	Approx. Overhead
S_{full} (Baseline)	None (input-dependent lookups)	Highest residual timing/cache risk	0%
S_{gen}	Algorithmic computation (no lookup table)	Lower timing/cache risk; high cost	~83%
S_{half} (Proposed)	Constant-time bitwise reconstruction	Lower timing/cache risk; low overhead	~3%–6%
S_{full} + Masking	Arithmetic masking & randomization	Lower first-order leakage risk (depends on masking quality)	~50%–100%

6.3 Validation of the MAT Model

We validate MAT on Cortex-M4 using the measured data above and the consistent security-cost definition from Section 5.

Step 1: Platform vector (θ).

$$\theta = (\theta_{\text{ROM}} : 0.9,\ \theta_{\text{RAM}} : 0.3,\ \theta_{\text{ALU}} : 0.6,\ \theta_{\text{ene}} : 0.8,\ \theta_{\text{sec}} : 0.7). \tag{21}$$

Step 2: Normalized cost vectors $C(S)$.

$$C(S_{\text{full}}) = (1.0,\ 0.1,\ 0.112,\ 0.475,\ 1.0), \tag{22}$$
$$C(S_{\text{half}}) = (0.5,\ 0.15,\ 0.116,\ 0.491,\ 0.3), \tag{23}$$
$$C(S_{\text{gen}}) = (0.0,\ 0.2,\ 1.0,\ 1.0,\ 0.25). \tag{24}$$

Step 3: Weight vector (ω). Using $\alpha_{\text{ROM}} = 1.0$, $\alpha_{\text{RAM}} = 0.5$, $\alpha_{\text{ALU}} = 0.7$, $\alpha_{\text{ene}} = 0.7$, and $\alpha_{\text{sec}} = 0.6$, we obtain:

$$\omega = (\omega_{\text{ROM}} : 0.367,\ \omega_{\text{RAM}} : 0.061,\ \omega_{\text{ALU}} : 0.171,\ \omega_{\text{ene}} : 0.229,\ \omega_{\text{sec}} : 0.171). \tag{25}$$

Step 4: Total cost computation.

$$\Phi(S_{\text{full}}) = 0.672, \tag{26}$$

$$\Phi(S_{\text{half}}) = 0.376, \tag{27}$$

$$\Phi(S_{\text{gen}}) = 0.455. \tag{28}$$

The resulting MAT scores, prediction ranks, and measured ranks are summarized in Table 9, where S_{half} obtains the lowest MAT cost and is ranked first by the model, consistent with the measured Cortex-M4 ranking.

Table 9: MAT validation on cortex-M4.

Strategy	Total Cost $\Phi(S;\theta)$	Prediction Rank	Measured Rank
S_half	0.376	1	1
S_gen	0.455	2	2
S_full	0.672	3	3

Sensitivity analysis. To evaluate whether the MAT recommendation depends on a single manually selected weight setting, we perturbed each weight scale α_k over $\{0.6, 0.8, 1.0, 1.2, 1.4\}$. Since the MAT model contains five cost dimensions, namely ROM, RAM, ALU latency, energy, and security risk, the perturbation set contains $5^5 = 3125$ weight-scale combinations.

For each weight-scale combination α, we recomputed the normalized weights $\omega_k(\theta, \alpha)$ and then evaluated the MAT objective value

$$\Phi(S;\theta,\alpha) = \sum_{k\in\mathcal{K}} \omega_k(\theta,\alpha)\, C_k(S) \tag{29}$$

For each combination, the strategy with the smallest objective value was recorded as

$$S^*(\theta,\alpha) = \arg\min_{S\in\mathcal{S}} \Phi(S;\theta,\alpha), \tag{30}$$

where $\mathcal{S} = \{S_{\text{full}}, S_{\text{half}}, S_{\text{gen}}\}$. The value 3072 was obtained by counting the number of perturbation combinations for which $S^*(\theta,\alpha) = S_{\text{half}}$, namely

$$\left|\{\alpha : S^*(\theta,\alpha) = S_{\text{half}}\}\right| = 3,072. \tag{31}$$

Therefore, across all 3125 offline sensitivity-analysis combinations, S_{half} was selected as the minimum-cost strategy in 3072 cases, corresponding to

$$\frac{3{,}072}{3{,}125} = 98.3\%. \tag{32}$$

This result indicates that the MAT recommendation remains highly robust under weight perturbation rather than depending on one specific tuning setting.

6.4 RISC-V and x86 Empirical Results

To complete cross-platform validation, Tables 10 and 11 present the measured outcomes on P2 and P3 and the MAT prediction–empirical outcome alignment across the three platforms. The purpose of this subsection is to verify whether the proposed MAT model can explain strategy selection beyond a single Cortex-M4 device. In particular, we examine whether ROM-sensitive embedded platforms favor S_{half},

while high-performance SIMD-oriented platforms may favor S_{full} due to lower ROM pressure and better vectorization efficiency.

Table 10: Cross-platform raw measurement data.

Platform	Strategy	Cycles/NTT	ROM (B)	Stack (B)	Energy/NTT
RISC-V RV32IMAC (P2)	S_{full}	4180	2470	88	2.03 µJ
RISC-V RV32IMAC (P2)	S_{half}	4328	2242	104	2.10 µJ
RISC-V RV32IMAC (P2)	S_{gen}	7530	2266	140	3.65 µJ
x86-64 AVX2 (P3)	S_{full}	890	2620	96	11.8 µJ
x86-64 AVX2 (P3)	S_{half}	938	2392	112	12.5 µJ
x86-64 AVX2 (P3)	S_{gen}	1768	2418	144	23.4 µJ

Table 11: MAT prediction vs. empirical outcome (cross-platform alignment).

Platform	MAT Predicted Best Strategy	Empirical Best Strategy	Match
ARM Cortex-M4 (P1)	S_{half} (from θ, ω)	S_{half} (Balances ROM/ALU)	Yes
RISC-V RV32IMAC (P2)	S_{half} (from θ, ω)	S_{half} (Balances ROM/ALU)	Yes
x86-64 AVX2 (P3)	S_{full} (from θ, ω)	S_{full} (Vectorization wins)	Yes

As shown in Table 10, the RISC-V RV32IMAC platform exhibits an embedded-platform trade-off pattern similar to Cortex-M4. Compared with S_{full}, S_{half} reduces ROM from 2470 to 2242 B, while the cycle overhead remains moderate, increasing from 4180 to 4328 cycles per NTT. The energy overhead is also limited, increasing from 2.03 to 2.10 µJ. These results indicate that the additional cost of branchless half-table reconstruction is relatively small compared with the ROM saving on a resource-constrained baseline core. In contrast, S_{gen} removes the twiddle table but significantly increases latency and energy consumption, reaching 7530 cycles and 3.65 µJ per NTT. Therefore, on the tested RISC-V platform, on-the-fly generation is less attractive than S_{half} for embedded deployment.

The x86-64 AVX2 platform shows a different trend. Although S_{half} still reduces ROM compared with S_{full}, the ROM saving is less important on a high-performance processor with a deeper memory hierarchy and stronger SIMD execution capability. On this platform, S_{full} achieves the lowest latency and energy consumption, requiring only 890 cycles and 11.8 µJ per NTT, whereas S_{half} increases these values to 938 cycles and 12.5 µJ. This platform-dependent reversal is important because it shows that the proposed framework does not always recommend the same strategy. Instead, the preferred implementation depends on the relative importance of ROM pressure, arithmetic overhead, and vectorization efficiency.

Table 11 further compares the MAT-predicted best strategy with the empirically observed best strategy. The prediction matches the empirical outcome on all three platforms. Specifically, S_{half} is selected for ARM Cortex-M4 and RISC-V RV32IMAC because these platforms are more sensitive to ROM footprint and embedded-resource constraints. By contrast, S_{full} is selected for x86-64 AVX2 because the platform benefits more from regular full-table access and vectorization-friendly data layout. This agreement supports the role of MAT as a hardware-aware strategy-selection framework rather than a single-device scoring rule. It also confirms that S_{half} should be interpreted as a Pareto-efficient embedded compromise rather than a universal optimum across all platforms.

6.5 Timing-Trace and TVLA Results (STM32F407)

To support leakage-related claims with empirical evidence, we provide the results for timing traces and first-order TVLA on STM32F407. This subsection is intended to distinguish constant-time coding practice from a complete side-channel-security proof. The experiments evaluate whether the proposed branchless twiddle-factor reconstruction introduces observable timing distinguishability or first-order power leakage under the tested configuration.

We collected timing traces on STM32F407 for two datasets: fixed test vectors and random test vectors. Each dataset contains 20,000 samples. The mean cycle counts are 3226.23 for fixed vectors and 3226.24 for random vectors, with standard deviations of 1.94 and 1.97, respectively. A two-sample Welch's t-test yields $p = 0.609$, indicating no statistically significant evidence of timing distinguishability under this setup.

Table 12 shows that the mean cycle counts of the fixed-vector and random-vector groups are almost identical. The difference between the two means is only 0.01 cycles, which is much smaller than the observed standard deviations. This indicates that the tested branchless accessor does not introduce measurable timing variation between the two input classes in this experiment. The Welch's t-test result further supports this observation, since $p = 0.609$ does not indicate statistically significant timing distinguishability under the tested configuration.

Table 12: Timing trace summary (STM32F407).

Dataset	Mean Cycles	Std. Dev.	Samples
Fixed vectors	3226.23	1.94	20,000
Random vectors	3226.24	1.97	20,000

For first-order TVLA, we compute Welch's t-test between fixed and random groups over 5000 aligned power-trace points collected through the shunt-resistor measurement path. The maximum absolute statistic is $|t|_{\max} = 3.26$. Using the standard threshold of $|t| > 4.5$, the result is a pass for this experiment configuration, indicating no first-order leakage evidence under this setup for the branchless logic.

Table 13 reports the first-order TVLA result obtained from the shunt-resistor power-trace measurement path. The maximum absolute t-statistic is $|t|_{\max} = 3.26$, which remains below the commonly used leakage-detection threshold of 4.5. This suggests that no first-order leakage evidence is observed for the tested branchless reconstruction logic under this experimental setup. However, this result should not be interpreted as a complete side-channel-security proof. Higher-order leakage, different probes, different compiler backends, electromagnetic measurements, and stronger adversarial models remain outside the current evaluation scope.

Table 13: First-order TVLA result (STM32F407).

Signal	Trace Count	Points	$\lvert t\rvert_{\max}$	Pass/Fail (4.5)
Power trace measured through shunt resistor	20,000	5000	3.26	Pass

Taken together, Tables 12 and 13 indicate that the proposed branchless reconstruction logic does not show statistically significant timing distinguishability or first-order TVLA leakage evidence under the tested configuration, thereby supporting the claimed lower timing-risk profile of the proposed implementation.

7 Conclusion

This work presented a multi-dimensional analysis of twiddle-factor symmetry exploitation for ML-KEM. We formalized a branchless half-table reconstruction rule with implementation-consistent indexing, and we introduced a device-aware MAT framework with explicit security-risk cost semantics. Based on current evidence, we draw the following conclusions:

1. On Cortex-M4, S_{half} reduces twiddle-factor ROM by 50% with about 3%–6% cycle overhead in controlled C-level comparison, making it a strong embedded compromise.
2. Under the MAT objective, S_{half} is a Pareto-efficient knee point for ROM-constrained settings rather than a universal global optimum.
3. MAT reproduces the Cortex-M4 ranking and remains robust under weight perturbation (98.3% of tested weight combinations select S_{half}).
4. The current revision distinguishes constant-time coding claims from full side-channel-security claims and provides cross-platform measurements (RISC-V and x86) together with empirical TVLA evidence supporting the security profile.

Future work will expand measurements to additional devices and higher-order leakage settings to further strengthen external validity.

Acknowledgement: Not applicable.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: The authors confirm contribution to the paper as follows: Xiaobing Liang contributed to the study conception, algorithm design for S_{half} strategy, formal analysis, and original draft preparation. Yu Qin contributed to the software implementation on ARM and RISC-V platforms, data curation, and assisted in the manuscript preparation. Shengdong Pan contributed to the construction and validation of the MAT model, visualization of experimental results, and participation in review and editing. Liang Tan supervised the overall research process, provided strategic guidance for multi-objective trade-off analysis, project administration, and secured key resources for the study. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ALU	Arithmetic Logic Unit
AVX2	Advanced Vector Extensions 2
FPGA	Field Programmable Gate Array
INTT	Inverse Number Theoretic Transform
IoT	Internet of Things
MAT	Memory-Arithmetic Trade-off
ML-KEM	Module-Lattice-Based Key-Encapsulation Mechanism
NTT	Number Theoretic Transform
PQC	Post-Quantum Cryptography
RAM	Random-Access Memory
ROM	Read-Only Memory

SIMD	Single Instruction, Multiple Data
TVLA	Test Vector Leakage Assessment

References

1. Shor PW. Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science; 1994 Nov 20–22; Santa Fe, NM, USA. p. 124–34.
2. National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard (FIPS 203) [Internet]. U.S. Department of Commerce; 2024 [cited 2026 Jun 4]. Available from: https://csrc.nist.gov/pubs/fips/203/final.
3. Alshomrany MO, Al-Hashimi M, Alsolami F, Saleh M, Abulnaja O, Al-Somani TF, et al. Lightweight FPGA-based number-theoretic transform for ML-KEM in post-quantum cryptography. IEEE Access. 2025;13:203621. doi:10.1109/access.2025.3638925.
4. Dam DT, Nguyen KD, Le DH, Pham CK. Accelerating post-quantum cryptography: a high-efficiency NTT for ML-KEM on RISC-V. Electronics. 2026;15(1):100. doi:10.3390/electronics15010100.
5. Bermudo Mera JM, Karmakar A, Verbauwhede I. Time-memory trade-off in Toom-Cook multiplication: an application to module-lattice based cryptography. IACR Trans Cryptogr Hardw Embed Syst. 2020;2020(2):222–44.
6. Ni Z, Khalid A, Liu W, O'Neill M. Towards a lightweight CRYSTALS-Kyber in FPGAs: an ultra-lightweight BRAM-free NTT core. In: Proceedings of the 2023 IEEE International Symposium on Circuits and Systems (ISCAS); 2023 May 21–25; Monterey, CA, USA; p. 1–5.
7. Im N, Yang H, Eom Y, Park SC, Yoo H. Efficient twiddle factor generators for NTT. Electronics. 2024;13(16):3128. doi:10.3390/electronics13163128.
8. Sonbul OS, Rashid M, Jaffar AY. Accelerating CRYSTALS-Kyber: high-speed NTT design with optimized pipelining and modular reduction. Electronics. 2025;14(11):2122.
9. Yang Y, Kuppannagari SR, Kannan R, Prasanna VK. NTTGen: a framework for generating low latency NTT implementations on FPGA. In: Proceedings of the 19th ACM International Conference on Computing Frontiers; 2022 May 17–22; Turin, Italy. p. 30–9.
10. Alkim E, Bilgin YA, Cenk M, Gerard F. Cortex-M4 optimizations for R, M-LWE schemes. IACR Trans Cryptogr Hardw Embed Syst. 2020;2020(3):336–57. doi:10.46586/tches.v2020.i3.336-357.
11. Kannwischer MJ, Rijneveld J, Schwabe P, Stoffelen K. pqm4: post-quantum cryptography on the ARM Cortex-M4 [Internet]. GitHub repository. 2019 [cited 2026 Jun 4]. Available from: https://github.com/mupq/pqm4.
12. Botros L, Kannwischer MJ, Schwabe P. Memory-efficient high-speed implementation of Kyber on Cortex-M4. In: Progress in Cryptology–AFRICACRYPT 2019. Cham, Switzerland: Springer; 2019. p. 209–28.
13. Abdulrahman A, Hwang V, Kannwischer MJ, Sprenkels D. Faster kyber and dilithium on the cortex-M4. In: Applied cryptography and network security (ACNS). Berlin/Heidelberg, Germany: Springer; 2022.
14. Kannwischer MJ. Polynomial multiplication for post-quantum cryptography [dissertation]. Nijmegen, The Netherlands: Radboud University; 2022.
15. Huang J, Zhao H, Zhang J, Dai W, Zhou L, Cheung RCC, et al. Yet another improvement of plantard arithmetic for faster Kyber on low-end 32-bit IoT devices. IEEE Trans Inf Forensics Secur. 2024;19(5):3800–13. doi:10.1109/tifs.2024.3371369.
16. Primas R, Pessl P, Mangard S. Single-trace side-channel attacks on masked lattice-based encryption. In: International Conference on Cryptographic Hardware and Embedded Systems. Cham, Switzerland: Springer International Publishing; 2017. p. 513–33.
17. Greconici D. Kyber on RISC-V [master's thesis]. Nijmegen: Radboud University; 2020 [cited 2026 Jun 4]. Available from: https://www.cs.ru.nl/masters-theses/2020/D_Greconici___KYBER_on_RISC-V.pdf.
18. mupq contributors. pqriscv: post-quantum crypto research and implementations on RISC-V [Internet]. GitHub repository. 2024 [cited 2026 Jun 4]. Available from: https://github.com/mupq/pqriscv.
19. Karabulut E, Aysu A. RANTT: a RISC-V architecture extension for the number theoretic transform. In: Proceedings of the 2020 30th International Conference on Field-Programmable Logic and Applications (FPL); 2020 Aug 31–Sep 4; Gothenburg, Sweden. p. 26–32.

20. Nannipieri P, Di Matteo S, Zulberti L, Albicocchi F, Saponara S, Fanucci L. A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms. IEEE Access. 2021;9:150798–808. doi:10.1109/access.2021.3126208.
21. Li H, Mentens N, Picek S. A scalable SIMD RISC-V based processor with customized vector extensions for CRYSTALS-Kyber. In: Proceedings of the 59th ACM/IEEE Design Automation Conference; 2022 Jul 10–14; San Francisco, CA, USA. p. 733–8.
22. Zou X, Peng Y, Li T, Kong L, Zhang L. Seesaw: a 4096-bit vector processor for accelerating Kyber based on RISC-V ISA extensions. Parallel Comput. 2025;123:103121.
23. Seiler G. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive. 2018. Paper 2018/039. [cited 2026 Jun 4]. Available from: https://eprint.iacr.org/2018/039.
24. Fu S, Zhang N, Franchetti F. Accelerating high-precision number theoretic transforms using Intel AVX-512. In: Proceedings of the 2024 33rd International Conference on Parallel Architectures and Compilation Techniques (PACT); 2024 Oct 13–16; Long Beach, CA, USA.
25. Avanzi R, Bos J, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, et al. CRYSTALS-Kyber: algorithm specifications and supporting documentation (version 3.02). Specification document. 2021 [cited 2026 Jun 8]. Available from: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.
26. Gueron S, Schlieker F. Speeding up R-LWE post-quantum key exchange. Cryptology ePrint Archive. Paper 2016/467. 2016 [cited 2026 Jun 10]. Available from: https://eprint.iacr.org/2016/467.
27. Zhang C, Liu D, Liu X, Zou X, Niu G, Liu B, et al. Towards efficient hardware implementation of NTT for kyber on FPGAs. In: Proceedings of the 2021 IEEE International Symposium on Circuits and Systems (ISCAS); 2021 May 22–28; Daegu, Republic of Korea. p. 1–5.
28. Wang C, Gao M. SAM: a scalable accelerator for number theoretic transform using multi-dimensional decomposition. In: Proceedings of the 2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD); 2023 Oct 28–Nov 2; San Francisco, CA, USA. p. 1–9.
29. Kundi DES, Bermudo Mera JM, Strub PY, Hutter M. High-performance NTT hardware accelerator to support ML-KEM and ML-DSA. In: Proceedings of the 2024 Workshop on Attacks and Solutions in Hardware Security; 2024 Oct 14–18; Salt Lake City, UT, USA. p. 100–5.