RISQrypt: Fast, Secure and Agile Hardware-Software Co-Design for Post-Quantum Cryptography

Tolun Tosun[1], Atıl Utku Ay[1], Quinten Norga[2], Suparna Kundu[2], Melik Yazıcı[1], Erkay Savaş[1] and Ingrid Verbauwhede[2]

[1] Sabanci University, Istanbul, Türkiye
[2] COSIC, KU Leuven, Leuven, Belgium

Abstract. In this paper, we present RISQrypt, the first unified architecture in the literature that implements ML-KEM (f.k.a. CRYSTALS-Kyber) and ML-DSA (f.k.a. CRYSTALS-Dilithium), two standardized lattice-based post-quantum cryptography (PQC) algorithms, with masking. RISQrypt is a hardware–software co-design framework that integrates dedicated cryptographic accelerators to speed up polynomial arithmetic, hashing, and mask-conversion operations, the latter being one of the primary bottlenecks in masked implementations of lattice-based PQC. Our design achieves low latency while providing both theoretical and practical side-channel security through formal proofs in the t-probing model and practical experiments. Specifically, the masked decapsulation of ML-KEM-768 requires **109K** clock cycles, while masked signing of ML-DSA-65 requires **1230K** clock cycles on average. These results demonstrate **11.3**× time-performance improvement over existing masked implementations. Our time performance results for unprotected functions also outperform the existing work; achieving speed-ups of **10.64**× and **8.94**× for ML-KEM's encapsulation and decapsulation algorithms, respectively, and **1.14**× and **2.23**× for ML-DSA's signature generation and verification algorithms, respectively, when compared to existing designs with similar resource usage. In addition, prior designs are more limited in scope, generally supporting only a single scheme and lacking either the side-channel protection or the unified, crypto-agile framework that enables support for both ML-KEM and ML-DSA as in our architecture. Leveraging the HW/SW co-design approach, the proposed architecture can be reprogrammed through firmware to support other (and future) PQC standards and algorithms sharing similar computational building blocks. As proof of concept, we demonstrate this capability by implementing FN-DSA (f.k.a. Falcon) and SLH-DSA (f.k.a. SPHINCS+).

Keywords: Masking · Kyber · Dilithium · Side-Channel Analysis · Hardware Acceleration · Lattice-Based Cryptography · RISC-V

1 Introduction

The security of currently deployed public-key cryptosystems depends on the computational hardness of number theory-based problems, such as the (elliptic curve) discrete logarithm problem used in the Diffie-Hellman key exchange [DH76, Kob87] and digital signatures [Sch90], as well as the integer factorization problem underlying RSA [RSA78]. Although they are conjectured to be resistant to all currently known cryptanalytic methods on classical computers, research has demonstrated that Shor's algorithm [Sho94] could efficiently solve them in polynomial time if a sufficiently powerful quantum computer were available.

Post-Quantum Cryptography To address the emerging security challenges posed by quantum computing, the National Institute of Standards and Technology (NIST) initiated a standardization process for post-quantum public-key cryptographic (PQC) algorithms in 2016. This initiative aims to establish quantum-resistant cryptographic standards, including digital signature algorithms (DSAs) and key encapsulation mechanisms (KEMs).

As of now, NIST has selected two KEMs: CRYSTALS-Kyber [BDK+17, BDK+22] (lattice-based, standardized as ML-KEM [Nat24b]) and HQC (code-based), alongside three DSAs: CRYSTALS-Dilithium [DKL+18, DKL+22] (lattice-based, standardized as ML-DSA [Nat24a]), Falcon (lattice-based), and SPHINCS+ (hash-based, standardized as SLH-DSA). Lattice-based cryptographic (LBC) constructions [Reg05, Ajt96], relying on the computational hardness of lattice problems, have emerged as leading candidates for post-quantum security, offering an exceptional balance between efficiency, scalability, and provable robustness against quantum adversaries.

Within this category, ML-KEM relies on the Module Learning With Errors (MLWE) problem, while ML-DSA is based on both MLWE and the Module Short Integer Solution (MSIS) problems. Both schemes originate from the same family of schemes (e.g. CRYSTALS), sharing core structural similarities and employing closely related computational primitives, thereby enabling seamless integration within a unified implementation framework, as a result.

Performance of Lattice-Based PQC As noted in the specification document of Kyber and Dilithium, the main sources of performance bottlenecks arise from hash computations and multiplication operations in polynomial rings. A widely adopted approach to alleviating these bottlenecks is hardware acceleration, in which computationally intensive tasks are offloaded from the host general-purpose processor to dedicated and optimized hardware units.

Keccak [BDPA12] is a cryptographic algorithm that serves as the foundation of the SHA-3 (Secure Hash Algorithm 3) standard, introduced by NIST. Its flexible sponge construction allows Keccak to support both fixed-length hash functions and extendable-output functions (XOFs), which can produce outputs of arbitrary length. This versatility makes it particularly well-suited for a broad range of cryptographic applications, including lattice-based schemes. Hardware acceleration of Keccak is a mature research area, with numerous well-established designs reported in the literature such as [GSM17, ZSS+21].

To enhance the efficiency of polynomial multiplication in lattice-based cryptographic algorithms, the Number Theoretic Transform (NTT) [AB75], an integer variant of the Discrete Fourier Transform (DFT), is commonly employed to improve computational performance. In the literature, a multitude of studies have proposed hardware accelerators for the NTT operation [FVR+22, MBB+23, NDMZ+21, MCL+23, WZZ+24, WWL+24, XL21, HHLW20, GLK21, DMMM23, JGCS24, KNAH22, ZZW+22, BNG21, WZCG22, CKRG+24]. While several of these works integrate support directly into the processor data-path using a tightly coupled approach [MBB+23, NDMZ+21, WWL+24], others adopt a loosely coupled design in which the accelerator operates as an independent unit connected to the processor core [FVR+22, WZZ+24, MCL+23]. Although the NTT constitutes the primary bottleneck in polynomial arithmetic, accelerating the remaining operations, including polynomial additions and subtractions, point-wise multiplications, divisions, pseudo-random sampling, and encoding/decoding, is also essential to achieve state-of-the-art performance.

Side-Channel Attacks and Countermeasures Side-channel attacks (SCAs) pose significant security threats to the implementation of cryptographic algorithms, with masking serving as the primary countermeasure. As PQC algorithms have only recently been standardized, their resistance to side-channel attacks has been investigated in a relatively limited number of studies. For example, in [FVR+22], dedicated operations required for

masked implementations were integrated into the instruction set architecture (ISA) to enhance resilience against SCAs.

There are two main masking types: arithmetic masking and Boolean masking. Many cryptographic schemes, especially the lattice-based PQC, require the use of both methods simultaneously, necessitating frequent and secure conversions between them. However, these arithmetic-to-Boolean (A2B) and Boolean-to-arithmetic (B2A) conversions are computationally intensive and resource consuming, underscoring the importance of accelerating masking conversions to maintain both security and efficiency. Consequently, A2B & B2A conversions emerge as a third major bottleneck in masked implementations of lattice-based PQC, alongside polynomial arithmetic and hashing.

Hardware Acceleration In the *loosely coupled* approach in hardware acceleration, the hardware unit operates as an independent module connected to the processor via a data bus. In this configuration, the unit typically includes its own cache memory or register file, as well as a dedicated control unit. This approach preserves the original ISA, ensuring software compatibility and simplifies integration into existing toolchains and hardware platforms. On the other hand, in the *tightly coupled* approach in hardware acceleration, a specialized hardware unit is integrated directly into the processor's datapath and is accessed through custom instruction(s). While this approach achieves low communication latency and minimal control overhead, it requires modifications to the original ISA and typically provides more limited acceleration gains compared to loosely coupled accelerators, unless vector instructions with significant area overhead are used as in [AOP+25]. RISC-V is an open and extensible ISA developed at UC Berkeley. Its modular RISC design supports a wide range of platforms, from simple embedded systems to high-performance and energy-efficient computing. Owing to its open-source and extensible nature, many cryptographic acceleration works build on RISC-V cores [FVR+22, MBB+23, NDMZ+21, WZZ+24, MCL+23].

Our proposed solution, **RISQrypt**, is based on a HW/SW co-design framework and relies on dedicated, loosely coupled hardware accelerators for hashing, polynomial arithmetic, and mask conversions.

1.1 Related Work

Several studies have investigated the efficient implementation of lattice-based PQC schemes, some of which are summarized in Table 1. These works can be broadly categorized into three groups: (i) software-only (SW), (ii) hardware-only (HW), and (iii) hardware–software co-design (HW/SW) solutions. SW solutions are agile, as the software can be re-compiled and re-programmed, but they inherently lack hardware acceleration and therefore cannot achieve high performance. Here, we use the term agility to refer ability of a design to support different parameter sets (e.g., moduli or polynomial degrees) either at run time or at software compile time, without requiring changes on the hardware at design time, where modifications are costly due to tape-out. HW solutions, on the other hand, offer high performance but are typically not agile, since algorithmic steps and parameters are hard-coded for Kyber/ML-KEM or Dilithium/ML-DSA. HW/SW co-design approaches enable the combination of agility and high performance; however, some existing works still fix their implementations to Kyber/ML-KEM, Dilithium/ML-DSA, or both. Overall, existing solutions either lack a unified architecture supporting both lattice-based schemes, do not address side-channel security, or provide only limited protection.

1.2 Contributions

In this work, we address all these challenges and close the existing gap in the literature. In summary:

Table 1: Overview of efficient implementations of lattice-based PQC in the literature.

Work	Supported Scheme	Class	SCA Protection	Hardware Acceleration	Agility
RISQrypt (This Work)	K, D	HW/SW	✓	✓	✓
[FVR+22]	K	HW/SW	✓	✓	✓
[WZZ+24]	K, D	HW/SW	✢	✓	
[KSFS24]	D	HW/SW		✓	✓
[MBB+23, NDMZ+21]	K, D	HW/SW		✓	
[DMMM23, WWL+24]	K	HW/SW		✓	
[MCL+23]	D	HW/SW		✓	
[AOP+25]	K, D	HW/SW		✓	✓
[JGCS24, KNAH22]	K	HW	✓	✓	
[JGCS24]	K	HW	▲	✓	
[BAA23]	K, D	HW	❖	✓	
[BNKU+26]	K, D	HW	✢	✓	
[XL21, HHLW20, GLK21]	K	HW		✓	
[ZZW+22, BNG21, WZCG22]	D	HW		✓	
[RRCC26]	D	HW	✓	✓	
[CKRG+24, AMI+23]	K, D	HW		✓	
[BC22, HKL+22, CGMZ23]	K	SW	✓		✓
[CGL+24, CGTZ23]	D	SW	✓		✓

✢ SCA protection limited to polynomial arithmetic. ❖ SCA protection limited to Kyber/ML-KEM. K: Kyber or ML-KEM, D: Dilithium or ML-DSA. ▲ hiding only.

- We present a unified architecture that supports both ML-KEM and ML-DSA with integrated side-channel protections, protecting all sensitive operations. The proposed design is unique due to it being *agile*: it can be reprogrammed to support ML-KEM or ML-DSA with optional side-channel countermeasures, or even to implement other algorithms built from similar computational blocks, *e.g.*, SLH-DSA and FN-DSA, as demonstrated in this work. We provide an in-depth security and performance analysis of the architecture, showing that our solution achieves superior performance, flexibility, and security over prior art. Although no prior work provides the same set of qualitative features, our design still outperforms existing implementations that support significantly fewer functionalities by up to an order of magnitude in both time performance and area–time product.

- We propose an efficient and fully programmable polynomial arithmetic accelerator, NTT-Lite, designed to support the core operations of ML-KEM and ML-DSA, as well as a wide range of cryptographic schemes based on modular and polynomial arithmetic. The accelerator features a dual-mode operation that efficiently supports both full- and half-word modes, as required by ML-KEM and ML-DSA, respectively. Additionally, we demonstrate its configurability by using it to accelerate non-linear arithmetic in masked ML-KEM and ML-DSA implementations.

- We propose a general-purpose, efficient, and high-throughput mask conversion accelerator (X2X), which supports up to 32-bit A2B & B2A conversions as well as other masked algorithms. We maximize the performance of our masked ML-KEM and ML-DSA implementations by carefully selecting, designing and mapping masked algorithms, including the novel MaskedZeroTestVec gadget for checking if an array is

non-zero, to the proposed hardware accelerators (NTT-Lite, X2X), without the need for additional operations at the software level. By identifying masking techniques from literature that suit the proposed hardware and instruction set, the need for software-level data processing and additional randomness is minimized. The security of our masked gadgets is verified theoretically and experimentally through the Test Vector Leakage Assessment (TVLA) methodology.

2 Preliminaries

This section briefly introduces the notation and reviews the essential background on the lattice-based PQC standards ML-KEM and ML-DSA, formerly known as Kyber and Dilithium, the number-theoretic transform, and the fundamentals of Boolean masking for side-channel protection.

2.1 Notation

Throughout the paper, bold letters, such as $\mathbf{a}$ are used to denote vectors and matrices. The elements of vectors and matrices are referred using square brackets (e.g., $\mathbf{a}[i]$ or $\mathbf{a}[i][j]$). $\mathbb{B}^l$ denotes the set of l-byte strings. The i-th bit of integers are accessed using square brackets, such as $a[i]$ for $a \in \mathbb{Z}_{2^b}$ and $0 \leq i < b$. The $||$ operator denotes concatenation. The symbol $\leftarrow$ denotes uniform random sampling from the set on the right-hand side, e.g., $a \leftarrow \mathbb{Z}$.

The polynomial ring $\mathcal{R}_q = \mathbb{Z}_q/(x^n + 1)$ consists of polynomials with coefficients in $\mathbb{Z}_q$, which form a residue class ring obtained by the division by the cyclotomic polynomial $x^n + 1$, where n is a power of two.

The notations $s^{\{0:d\}_A}$ and $s^{\{0:d\}_B}$ denote the complete set of shares of a d^{th}-order arithmetic- and Boolean-masked variable s, respectively. When a single index appears in the superscript, such as $s^{\{i\}_A}$, it refers specifically to share i. If $s^{\{0:d\}_A}$ (or $s^{\{0:d\}_B}$) is provided as input or output of a function, the function receives or produces all shares of the variable. Likewise, when this notation appears in an arithmetic or Boolean operation, it indicates that the same operation is applied independently to each share.

For A2B & B2A conversions, we use a subscript to indicate the modulus type. For example, $\mathsf{B2A}_{2^k}$ denotes a Boolean-to-arithmetic conversion with a power-of-two modulus, whereas $\mathsf{B2A}_q$ denotes the conversion modulo an arbitrary modulus, not necessarily a power of two.

2.2 ML-KEM

ML-KEM is a lattice-based KEM built upon the hardness of the MLWE problem, which offers a favorable balance between security and efficiency. The scheme operates over the polynomial ring $\mathcal{R}_{\mathsf{KEM},q} = \mathbb{Z}_q/(x^n + 1)$ with $n = 256$ and modulus $q = 3329$. By exploiting structured module lattices, the ML-KEM scheme achieves compact key sizes and efficient polynomial arithmetic. Polynomial multiplications are accelerated using the NTT, and noise terms are sampled from centered binomial distributions to ensure both performance and security.

ML-KEM achieves security against chosen ciphertext attacks (CCA) as it is constructed by applying the Fujisaki–Okamoto (FO) transform to the chosen plaintext attack-secure (CPA) $\mathsf{K\text{-}PKE}$ scheme. Following this approach, a shared secret is derived by hashing both the ciphertext and an ephemeral secret, and a verification step ensures robustness against ciphertext tampering.

The key decapsulation procedure of ML-KEM is presented in Algorithm 1, Algorithm 2, and Algorithm 3. We refer the reader to the ML-KEM specification [Nat24b] for a detailed description of the remaining functions, including key generation, encapsulation, and the

Algorithm 1 K-PKE.Encrypt($\mathsf{ek_{PKE}}, m, r$) [Nat24b]

Input: encryption key $\mathsf{ek_{PKE}} \in \mathbb{B}^{384k+32}$.
Input: message $m \in \mathbb{B}^{32}$.
Input: randomness $r \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$. ▷ $d_u = 10, d_v = 4$ for ML-KEM-768

$N \leftarrow 0$
$\hat{\mathbf{t}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{ek_{PKE}}[0 : 384k])$
$\rho \leftarrow \mathsf{ek_{PKE}}[384k : 384k + 32]$
for $i = 0$ to $k - 1$ **do**
  for $j = 0$ to $k - 1$ **do**
    $\hat{\mathbf{A}}[i][j] \leftarrow \mathsf{SampleNTT}(\rho||j||i)$
  end for
end for
for $i = 0$ to $k - 1$ **do**
  $\mathbf{y}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(r, N))$, $N \leftarrow N + 1$
end for
for $i = 0$ to $k - 1$ **do**
  $\mathbf{e}_1[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$, $N \leftarrow N + 1$ ▷ $\eta_2 = 2$ for ML-KEM-768
end for
$e_2[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$
$\hat{\mathbf{y}} \leftarrow \mathsf{NTT}(\mathbf{y})$
$\hat{\mathbf{u}} = \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^\top \circ \hat{\mathbf{y}}) + \mathbf{e_1}$
$\mu \leftarrow \mathsf{Decompress}_1(\mathsf{ByteDecode}_1(m))$
$v = \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^\top \circ \hat{\mathbf{y}}) + e_2 + \mu$
$c_1 = \mathsf{ByteEncode}_{d_u}(\mathsf{Compress}_{d_u}(\mathbf{u}))$, $c_2 = \mathsf{ByteEncode}_{d_v}(\mathsf{Compress}_{d_v}(v))$
return $c \leftarrow (c_1||c_2)$

Algorithm 2 K-PKE.Decrypt($\mathsf{dk_{PKE}}, c$) [Nat24b]

Input: decryption key $\mathsf{dk_{PKE}} \in \mathbb{B}^{384k}$.
Input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Output: message $m \in \mathbb{B}^{32}$.

$c_1 \leftarrow c[0 : 32d_u k]$, $c_2 \leftarrow c[32d_u k : 32(d_u k + d_v)]$
$\mathbf{u}' \leftarrow \mathsf{Decompress}_{d_u}(\mathsf{ByteDecode}_{d_u}(c_1))$, $v' \leftarrow \mathsf{Decompress}_{d_v}(\mathsf{ByteDecode}_{d_v}(c_2))$
$\hat{\mathbf{s}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{dk_{PKE}})$
$w \leftarrow v' - \mathsf{NTT}^{-1}(\hat{\mathbf{s}}^\top \circ \mathsf{NTT}(\mathbf{u}'))$
return $m \leftarrow \mathsf{ByteEncode}_1(\mathsf{Compress}_1(w))$

internal subroutines. $\mathsf{ByteDecode}_\Delta$ maps a sequence of bits (message, key, output of a hash function) to polynomial coefficientsof Δ-bit in $\mathbb{Z}_q[x]$, with $\mathsf{ByteEncode}_\Delta$ performing the inverse operation. Several functions are implemented using SHA-3, including: G, H, J, PRF. Finally, SamplePolyCBD samples a polynomial from a centered binomial distribution in range $[-\eta, \eta]$, where η is a small integer. Compress and Decompress are defined as:

$$\mathsf{Compress}_q(x, d) = \left\lceil \frac{2^d}{q} \cdot x \right\rfloor \bmod 2^d \qquad \mathsf{Decompress}_q(x, d) = \left\lceil \frac{q}{2^d} \cdot x \right\rfloor. \tag{1}$$

where $\lceil x \rfloor$ denotes rounding x to the nearest integer.

2.3 ML-DSA

ML-DSA is a lattice-based digital signature scheme relying on the hardness of the MLWE

Algorithm 3 $\mathsf{K\text{-}PKE.Decaps}(\mathsf{dk}, c)$ [Nat24b]

Input: decapsulation key $\mathsf{dk} \in \mathbb{B}^{768k+96}$.
Input: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.
Output: shared secret key $K \in \mathbb{B}^{32}$.

$\mathsf{dk_{PKE}} \leftarrow \mathsf{dk}[0 : 384k]$, $\mathsf{ek_{PKE}} \leftarrow \mathsf{dk}[384k : 768k + 32]$
$h \leftarrow \mathsf{dk}[768k + 32 : 768k + 64]$, $z \leftarrow \mathsf{dk}[768k + 64 : 768k + 96]$
$m' \leftarrow \mathsf{K\text{-}PKE.Decrypt}(\mathsf{dk_{PKE}}, c)$
$(K', r') \leftarrow \mathsf{G}(m'||h)$
$\bar{K} \leftarrow \mathsf{J}(z||c)$
$c' \leftarrow \mathsf{K\text{-}PKE.Encrypt}(\mathsf{ek_{PKE}}, m', r')$
if $c \neq c'$ **then**
    $K' \leftarrow \bar{K}$
end if
return K'

and MSIS problems. It achieves strong security through structured lattices and rejection sampling, avoiding the need for complex Gaussian sampling. ML-DSA employs parameters $n = 256$ and $q = 8380417$ over the ring $\mathcal{R}_{\mathsf{DSA},q} = \mathbb{Z}_q/(x^n + 1)$.

The signature generation procedure of ML-DSA is presented in Algorithm 4, and we refer to FIPS204 for the full specification of ML-DSA [Nat24a]. Again, the function H is based on SHA-3. The function $\mathsf{Decompose}(r, \alpha)$ can be defined as follows. Let r be an integer in $\mathbb{Z}_q$, then we can represent r as

$$r = r_1\alpha + r_0, \tag{2}$$

where $q - 1 = m \cdot \alpha$, $r_1 \in [0, m]$, and $r_0 \in (-\alpha/2, \alpha/2]$. $\mathsf{MakeHint}(z, r, \alpha)$ checks whether z generates a carry bit for $\mathsf{HighBits}(r, \alpha)$ or not. $\mathsf{UseHint}(h, r, \alpha)$ (used in signature verification) adds the carry to $\mathsf{HighBits}(r, \alpha)$ depending on h, which denotes the hint vector produced by $\mathsf{MakeHint}(z, r, \alpha)$.

2.4 Number Theoretic Transform (NTT)

NTT is a modular analogue of the DFT, commonly employed in PQC and FHE to accelerate polynomial multiplications in $\mathcal{R}_q$. Unlike the DFT, which operates over the complex numbers, NTT is performed over a finite field or ring, typically $\mathbb{Z}_q$, where q is a prime admitting a principal $2n$-th root of unity, ψ. For a polynomial $a(x)$ of degree $n - 1$, the NTT maps its coefficient vector to the evaluation domain by computing

$$\bar{a}_k = \sum_{j=0}^{n-1} a_j \psi^{j(2k+1)} \bmod q, \quad 0 \leq k < n, \tag{3}$$

where ψ is a primitive $2n$-th root of unity modulo q.

In the time domain, multiplying two degree-$(n-1)$ polynomials using schoolbook multiplication requires $\mathcal{O}(n^2)$ coefficient multiplications and additions. By applying NTT, the polynomials are transformed into the frequency domain, where multiplication reduces to element-wise multiplication, which is only $\mathcal{O}(n)$. The Fast Fourier Transform (FFT) algorithm, adapted to modular arithmetic, enables the efficient computation of the NTT and its inverse (INTT) in $\log n$ stages, each consisting of $n/2$ butterfly operations, resulting in an overall complexity of $\mathcal{O}(n \log n)$. Due to the convolution theorem, multiplication in the time domain (i.e., multiplication in $\mathcal{R}_q$) can be performed using element-wise multiplication in the NTT domain.

Algorithm 4 ML-DSA.Sign(sk, M', rnd)

Input: Private key $sk \in \mathbb{B}^{32+32+64+32((l+k)\text{bitlen}(2\eta)+dk)}$
Input: Formatted message $M' \in \{0,1\}^*$
Input: Random (or dummy) string $rnd \in \mathbb{B}^{32}$
Output: Signature $\sigma \in \mathbb{B}^{\lambda/4+32l(1+\text{bitlen}(\gamma_1-1))+\omega+k}$ ▷ $\lambda = 196, \gamma_1 = 2^{19}, \omega = 55$ for ML-DSA-65

1: $(\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0) \leftarrow \mathsf{skDecode}(sk)$
2: $\hat{\mathbf{s}}_1 \leftarrow \mathsf{NTT}(\mathbf{s}_1)$, $\hat{\mathbf{s}}_2 \leftarrow \mathsf{NTT}(\mathbf{s}_2)$, $\hat{\mathbf{t}}_0 \leftarrow \mathsf{NTT}(\mathbf{t}_0)$
3: $\hat{\mathbf{A}} \leftarrow \mathsf{ExpandA}(\rho)$
4: $\mu \leftarrow \mathsf{H}(\mathsf{BytesToBits}(tr)||M', 64)$
5: $\rho'' \leftarrow \mathsf{H}(K||rnd||\mu, 64)$, $\kappa \leftarrow 0$, $(\mathbf{z}, \mathbf{h}) \leftarrow \perp$
6: **while** $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
7: $\quad \mathbf{y} \in R_q^l \leftarrow \mathsf{ExpandMask}(\rho'', \kappa)$
8: $\quad \mathbf{w} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}} \circ \mathsf{NTT}(\mathbf{y})))$
9: $\quad \mathbf{w}_1 \leftarrow \mathsf{HighBits}(\mathbf{w})$
10: $\quad \tilde{c} \leftarrow \mathsf{H}(\mu||\mathsf{w1Encode}(\mathbf{w}_1), \lambda/4)$
11: $\quad c \in R_q \leftarrow \mathsf{SampleInBall}(\tilde{c})$
12: $\quad \hat{c} \leftarrow \mathsf{NTT}(c)$
13: $\quad \ll c\mathbf{s}_1 \gg \leftarrow \mathsf{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_1)$, $\ll c\mathbf{s}_2 \gg \leftarrow \mathsf{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{s}}_2)$
14: $\quad \mathbf{z} \leftarrow \mathbf{y} + \ll c\mathbf{s}_1 \gg$
15: $\quad \mathbf{r}_0 \leftarrow \mathsf{LowBits}(\mathbf{w} - \ll c\mathbf{s}_2 \gg)$
16: $\quad$ **if** $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ **or** $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ **then**
17: $\quad\quad (\mathbf{z}, \mathbf{h}) \leftarrow \perp$
18: $\quad$ **else**
19: $\quad\quad \ll c\mathbf{t}_0 \gg \leftarrow \mathsf{NTT}^{-1}(\hat{c} \circ \hat{\mathbf{t}}_0)$
20: $\quad\quad \mathbf{h} \leftarrow \mathsf{MakeHint}(-\ll c\mathbf{t}_0 \gg, \mathbf{w} - \ll c\mathbf{s}_2 \gg + \ll c\mathbf{t}_0 \gg)$
21: $\quad\quad$ **if** $|| \ll c\mathbf{t}_0 \gg ||_\infty \geq \gamma_2$ **or** #1's in $\mathbf{h} > \omega$ **then**
22: $\quad\quad\quad (\mathbf{z}, \mathbf{h}) \leftarrow \perp$
23: $\quad\quad$ **end if**
24: $\quad$ **end if**
25: $\quad \kappa \leftarrow \kappa + l$
26: **end while**
27: **return** $\sigma \leftarrow \mathsf{sigEncode}(\tilde{c}, \mathbf{z} \bmod^{\pm} q, \mathbf{h})$

The incomplete NTT is a variant of the standard NTT, which may be used to reduce computational overhead and memory usage in lattice-based cryptosystems. Unlike the full NTT, which performs all $\log_2 n$ butterfly stages in both forward and inverse directions, the incomplete version omits one or more of the stages when the algorithmic structure allows. Subsequent element-wise multiplication, then, becomes more involved and should be adapted to operate on this partially transformed data. Incomplete NTT is sometimes necessary due to the mathematical structure of $\mathcal{R}_q$, as in ML-KEM, where a $2n$-th root of unity does not exist. It may also be preferred for efficiency reasons in ML-DSA [AHKS22].

2.5 Barrett Reduction

Barrett reduction [Bar87] is an efficient algorithm for computing the remainder $r = a \bmod q$, where $a = q \cdot t_0 + r$ for two integers a and q. The algorithm first computes a quotient t_0, and then derives the remainder as $a - t_0 \cdot q$. We use two variants of Barrett reduction in Algorithm 5 and Algorithm 6. The first variant incurs higher cost, as it requires a $2\lambda \times 2\lambda$-bit multiplication (Line 1), whereas the second variant only involves a $\lambda \times \lambda$-bit multiplication (Line 2). However, the former supports reduction of inputs of arbitrary size

Algorithm 5 Barrett Reduction Variant 1

Input: word-size λ
Input: $a \in [0, 2^{2\lambda})$
Input: $q \in (0, 2^{\lambda})$, $\mu = \lfloor 2^{2\lambda}/q \rfloor$
Output: $r = a \mod q$
Output: $a = t_1 \cdot q + r$
1: $t_0 = (a \cdot \mu) \gg 2\lambda$
2: $t_1 = t_0 \cdot q \mod 2^{\lambda+1}$
3: $r = a - t_1 \mod 2^{\lambda+1}$
4: **if** $r \geq q$ **then**
5: $\quad r = r - q$, $t_0 = t_0 + 1$
6: **end if**
7: **return** r, t_0

Algorithm 6 Barrett Reduction Variant 2

Input: word-size λ
Input: $a \in [0, 2^{2\lambda})$
Input: $q \in [2^{\lambda-1}, 2^{\lambda})$, $\mu = \lfloor 2^{2\lambda}/q \rfloor$
Output: $r = a \mod q$
Output: $a = t_1 \cdot q + r$
1: $a_1 = a \gg \lambda$
2: $t_0 = (a_1 \cdot \mu) \gg \lambda$
3: $t_1 = t_0 \cdot q \mod 2^{\lambda+2}$
4: $r = a - t_1 \mod 2^{\lambda+2}$
5: **if** $r \geq q$ **then**
6: $\quad r = r - q$, $t_0 = t_0 + 1$
7: **end if**
8: **if** $r \geq q$ **then**
9: $\quad r = r - q$, $t_0 = t_0 + 1$
10: **end if**
11: **return** r, t_0

given the word size λ. In contrast, the second variant assumes that the input a is at most twice the bit-length of the modulus q (i.e., the divisor).

2.6 Arithmetic and Boolean Masking

Masking is a widely applied countermeasure designed to secure cryptographic hardware and software implementations against passive side-channel attacks, such as those exploiting electromagnetic (EM) emissions or power consumption. Based on the secret sharing principle introduced by Shamir [Sha79], a sensitive variable s is decomposed into $d+1$ uniformly random shares $s^{\{i\}}$, where $0 \leq i \leq d$, ensuring d-th order probing security [ISW03].

In order to prove the probing security of larger gadgets, which are small, composable circuits or algorithmic components that operate on masked shares, Barthe et al. [BBD+16] introduce security notions of (strong-) non-interference (t-(S-)NI) for proving the security of smaller gadgets and their composition.

Definition 1 (t-(Strong) Non-Interference [BBD+16,SPOG19])**.** A gadget with one output sharing and $m^{\{i\}}$ input shares is t-NI (resp. t-SNI) secure if any set of at most t_1 probes on its internal wires and t_2 probes on output sharings such that $t_1 + t_2 \leq t$ can be simulated with $t_1 + t_2$ (resp. t_1) shares of each of its $m^{\{i\}}$ input shares.

Both of the following masking techniques are employed in lattice-based PQC implementations to ensure side-channel resistance: arithmetic masking and Boolean masking. For arithmetic masking, the sensitive value $s \in \mathbb{Z}_q$ is divided into uniformly random $d+1$ shares $s^{\{i\}} \in \mathbb{Z}_q$, and s is the arithmetic sum of $d+1$ shares, i.e., $s = \sum_{i=0}^{d} s^{\{i\}_A} \mod q$. For Boolean masking, s is the result of the bitwise XOR of $d+1$ shares, i.e., $s = \oplus_{i=0}^{d} s^{\{i\}_B}$. In masking, each individual share is statistically independent of the original secret. Consequently, an attacker must have/gain knowledge of all $d+1$ shares to infer any meaningful information about the underlying secret s. As both arithmetic and Boolean masking are used in lattice-based schemes such as digital signature algorithms and key encapsulation mechanisms, it becomes necessary to transform shares between these masking domains (A2B and B2A), which incurs significant performance overhead.

Refresh is a frequently used gadget to satisfy specific security requirements (e.g., achieving t-SNI) by re-randomizing the shares of sensitive variables. Intuitively, this

operation prevents the propagation of dependencies on previously used randomness across different gadgets. Formally, Refresh combines its input with a fresh sharing of zero in the corresponding domain. That is, $s'^{\{0:d\}_B} \leftarrow s^{\{0:d\}_B} \oplus z^{\{0:d\}_B}$ in the Boolean domain (or $s'^{\{0:d\}_A} \leftarrow s^{\{0:d\}_A} + z^{\{0:d\}_A}$ in the arithmetic domain), where $z^{\{0:d\}_B}$ (or $z^{\{0:d\}_A}$) is freshly generated sharing of zero ($z = 0$).

3 Hardware Accelerators

In this section, we provide an overview of the architecture and cryptographic hardware accelerators used in RISQrypt. The design comprises three cryptographic accelerators, a polynomial arithmetic accelerator referred to as **NTT-Lite**, a **Keccak** accelerator, and an A2B/B2A mask conversion accelerator referred to as **X2X**.

The hardware accelerators are leveraged through a software layer (Section 5). Consequently, the proposed approach is independent of the overall system architecture, provided that the accelerators can be efficiently accessed by the processor and that the processor can be programmed to execute our PQC software. For clarity, we describe the architecture here for the first-order masking case to keep the presentation simple. However, our approach supports arbitrary-order masking, including both X2X and Keccak accelerators (NTT-Lite operates independently of the masking order, *i.e.,* share-wise processing, and therefore does not require the masking order to be specified at synthesis time), as well as the masked algorithms presented in Section 5. In practice, our hardware platform synthesized for d-th order masking can be configured at run time to operate at any masking order up to and including d.

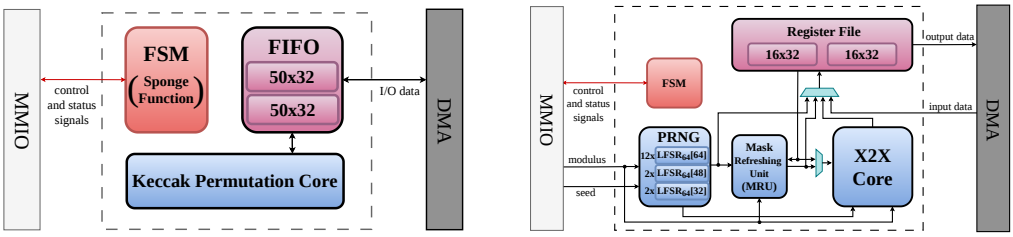
3.1 Memory Architecture

Cryptographic accelerators access the data RAM via a shared Direct Memory Access (DMA) interface to improve data read and write efficiency. Memory-Mapped I/O (MMIO) is a standard master–slave communication abstraction used to access peripherals. Although NTT-Lite also operates on masked data, it performs only linear operations and can therefore process each share independently using a single DMA interface. In contrast, Keccak and X2X include non-linear operations that require simultaneous access to multiple shares, necessitating separate DMA interfaces per share to prevent side-channel leakage that could result from multiplexing multiple shares on a single interface. For Keccak and X2X, multiple DMA interfaces are never active at the same time. In our implementation, although both MMIO interface and the DMA interface are realized using the Wishbone protocol, the design is not tied to this choice. For example, these interfaces could alternatively be implemented using AXI, also a widely adopted standard. Similarly, although the 32-bit HORNET RISC-V Core[1] [YTO21] is used as the main processor in our implementation, this choice is also not mandatory, and our work is orthogonal to the specific software core.

3.2 Keccak Accelerator

Figure 1a presents the simplified architecture of the Keccak Accelerator. We integrated the masked Keccak permutation core proposed in [GSM17], based on the so-called domain-oriented masking approach, into our architecture. We should note that the integrated Keccak core only provides the Keccak permutation. Implementing higher-level hash functions such as SHA-3 or SHAKE requires to implement the so-called sponge construction [Nat15]. In our design, the sponge construction is implemented directly within the accelerator to minimize software interaction and thereby improve overall performance. Furthermore, a FIFO mechanism is incorporated to overlap the Keccak permutation

[1] https://github.com/yavuz650/RISC-V **commit id:** 6c7af88



(a) Keccak (b) X2X

Figure 1: Keccak and X2X accelerators high-level architecture. Black lines denote data paths, red lines denote control and status signals. Internal control signals are omitted for clarity.

and data input and output. The level of parallelism in the Keccak permutation core is configurable, and we configured it such that it finishes a Keccak permutation in $4 \times 24 = 96$ clock cycles. Masking can be enabled or disabled at run time via control signals, allowing improved efficiency for non-sensitive hashing operations.

3.3 X2X Accelerator

Applying masking countermeasures to lattice-based cryptographic schemes necessitates the use of both arithmetic and Boolean masking [MGTF19, ÖY23, KDVB+22, CGTZ23]. In particular, core operations such as polynomial addition and multiplication are performed over arithmetic shares, while bitwise operations—such as those used in the Keccak hash function—are more naturally protected using Boolean masking. Consequently, a secure implementation must provide an efficient and reliable mechanism to convert between these two masking domains, specifically through A2B and B2A conversions.

To accelerate these operations, we integrate a dedicated mask conversion unit into our architecture. Specifically, we adopt the X2X core from [NDKV25], which also lends its name to our cryptographic accelerator. The X2X core enables mask conversion with high-throughput, such as 1 to 2 conversions per cycle, supporting both A2B and B2A conversions, for both a power-of-two or prime moduli. We would like to note that X2X core was originally designed for a word size of 13-bit in [NDKV25], and it is expanded to 32-bit for this work. The X2X core employs a 13-stage pipeline to support high-throughput A2B and B2A conversions. When the modulus is a power of two, the core supports a dual mode in which two conversions can be performed simultaneously, thereby doubling the throughput.

Table 2: Operations supported by X2X accelerator.

Opcode	Active Units			Modes
	PRNG	MRU	X2X Core	
`OP_PRNG`	✓	✗	✗	
`OP_X2X`	✓	✗	✓	A2B, B2A, $B2A_{1\text{-bit}}$
`OP_REF`	✓	✓	✗	A, B
`OP_MASK`	✓	✓	✗	A, B
`OP_REFX2X`	✓	✓	✓	A2B, B2A, $B2A_{1\text{-bit}}$

A: Arithmetic, B: Boolean

The simplified architecture of X2X accelerator is given in Figure 1b. As shown, in addition to the X2X core, we integrated a pseudo-random number generator (PRNG), and a mask refresher unit (MRU). To further improve performance, two 16 × 32-bit register

files are integrated as internal memory, one for each share. This enables 16 conversions to be completed in 29 clock cycles (excluding main memory access latency, non-dual mode), by leveraging pipelining in X2X core. The PRNG is primarily used to supply randomness to the X2X core, enabling it to operate at the desired throughput. It is implemented using 16 parallel linear feedback shift registers (LFSRs), each with a 64-bit state. The feedback polynomials are chosen to be primitive, ensuring non-repeating sequences of length $2^{64}-1$. 12 of these LFSRs generate 64-bit outputs in a single cycle through loop unrolling, while 2 of them generate 48-bit outputs and 2 of them generate 32-bit outputs. Overall, the PRNG is used to generate 736-bit uniformly random input used by the X2X core. Moreover, two 32-bit random integers are used to generate uniformly random elements in $\mathbb{Z}_q$ via rejection sampling. For each sample, three $\lceil \log q \rceil$-bit candidates are derived by masking the lower $\lceil \log q \rceil$ bits of 32-bit random values obtained from the LFSRs (reserved specifically for this purpose). The first candidate that is strictly less than q is accepted. One of these random numbers in $\mathbb{Z}_q$ is consumed by the X2X core in A2B mode while the other one is used by the MRU. Therefore, if no valid candidate is found, the X2X accelerator stalls. The probability of acceptance is given by

$$1 - \left(\frac{(2^{\lceil \log q \rceil} - q)}{2^{\lceil \log q \rceil}} \right)^3,$$

For example, for ML-KEM ($q = 3329$), this implies an acceptance rate of 0.993.

The operation modes supported by the X2X accelerator are summarized in Table 2. `OP_X2X` corresponds to performing A2B or B2A operations using the X2X core while `OP_REF` refers to performing mask refreshing in arithmetic or Boolean domain utilizing the MRU. While MRU and X2X core can operate independently, they are also connected in a cascaded manner, enabling efficient performing of mask refreshing followed by a mask conversion operation in a single pipeline execution (see Figure 1b), denoted by `OP_REFX2X`. In this configuration, the latency of A2B and B2A operations increases by 1 clock cycle due to the additional latency introduced by mask refreshing. This feature is important to achieve provable security in an efficient manner, which is further discussed in Section 5. The X2X accelerator also supports performing the initial masking of sensitive data through the MRU (denoted by `OP_MASK`), ensuring that sensitive values are never manipulated directly by software, where microarchitectural leakages can occur [GD23a]. Moreover, there is a dedicated command in the X2X accelerator, i.e., `OP_PRNG`, to read the PRNG output, so the software can use the X2X accelerator to produce pseudo-random numbers in the given modulus. Another important feature of X2X accelerator is the 1-bit mode specific to B2A, denoted by $\text{B2A}_{\text{1-bit}}$, where the input data is sent to the X2X core bit by bit, with a configurable stride between bits. This feature is particularly important for certain masked gadgets in ML-KEM, as also discussed in Section 5.2.1. We note that all operations also support arbitrary moduli, not necessarily powers of two.

4 NTT-Lite

In this section, we provide a detailed description of the architecture and sub-modules of our polynomial arithmetic accelerator NTT-Lite.

4.1 Scope

NTT-Lite is designed to accelerate not only the NTT operation but also other polynomial arithmetic and supporting functions used in PQC schemes, such as compression, encoding and pseudo-random sampling, which cannot be efficiently implemented using software. Although this work focuses on secure implementations of ML-KEM and ML-DSA, NTT-Lite is not restricted to the parameters of these schemes. It supports modulus q sizes up

to 32-bit and polynomial degrees n of up to 256, ensuring an agile design, flexibility and reusability across different LBC algorithms.

On the other hand, masked implementations of ML-KEM and ML-DSA require configurations that deviate from the standard parameter sets (e.g., $q = 3329$ or $q = 8380417$), such as the moduli of these schemes. This flexibility is particularly necessary for implementing the arithmetic involved in their non-linear operations. Our configurable NTT design satisfies these requirements. Additional discussion on the role of NTT-Lite in masked implementations is provided in Section 5. We employ unsigned arithmetic over $\mathbb{Z}_q$ as signed arithmetic makes differential SCA attacks significantly easier [TMS24].

4.2 Architecture

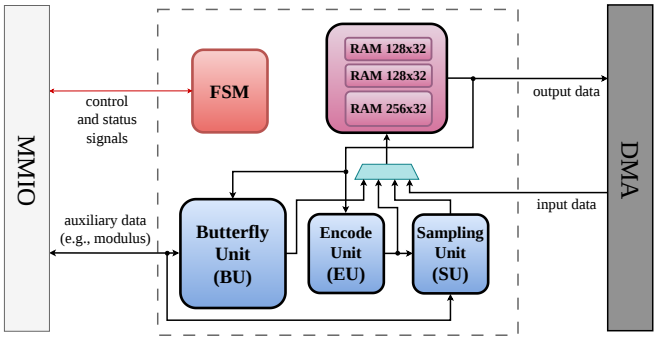


Figure 2: NTT-Lite high-level architecture. Internal control signals are omitted for clarity.

Figure 2 illustrates the high-level and simplified architecture for NTT-Lite. As shown, the NTT-Lite contains two 128×32 and one 256×32 RAM blocks. The core arithmetic unit in NTT-Lite is the Butterfly Unit (BU) as in almost all NTT designs in the literature. In addition, an Encode Unit (EU) is integrated into the design to accelerate the encoding and decoding operations used in PQC algorithms. Moreover, a Sampling Unit (SU) is implemented to perform operations related to pseudo-random number sampling used in PQC schemes. FSM performs address calculations and generates control signals for the internal memories as well as BU, EU and SU.

Since a butterfly circuit in the NTT computation has two inputs and two outputs, we use two independent 128×32 RAMs. In this manner, NTT-Lite can feed BU with a valid pair of inputs at each clock cycle. The 256×32 RAM is used for the storage of twiddle factors in NTT, as well as one of the operands in certain operations. We detail the internal RAM usage in the following sections. In addition to the main data path, an auxiliary 160-bit data path is employed. This path delivers constant values required by the provided functionalities, such as the modulus used in modular arithmetic. Further details on this auxiliary constant-data are provided in the next section.

4.2.1 Word Modes

The architecture provides two main operating modes, *single* and *dual*, selectable at run time. In single mode, the unit operates with a 32-bit word size, whereas in dual mode it is configured for 16-bit words. In dual mode, each 32-bit word stored in RAM is partitioned into its upper and lower halves, which are treated as two independent 16-bit words, enabling parallel processing and improved throughput. The dual-mode support in our design is similar to that of [AMI+23], but extended to support 16- and 32-bit operands. Additionally, *poly* mode is implemented as a variant of dual-mode, with all operations, except modular multiplication, executed identically. In this mode, the multiplication operation is realized

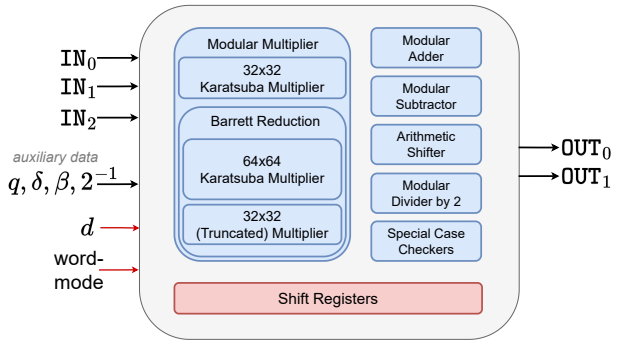


Figure 3: High-level architecture of the BU.

as degree-1 polynomial multiplication, supporting modulus sizes of up to 16-bit. The poly-mode is beneficial for implementing the base multiplication in incomplete NTT, such as the case of ML-KEM. The pseudo-code for modular multiplication in poly mode is given as follows:

$$C_H = A_L \cdot B_H + A_L \cdot B_H \qquad \text{mod } q \tag{4}$$
$$C'_L = A_L \cdot B_L \qquad \text{mod } q \tag{5}$$
$$T = A_H \cdot B_H \qquad \text{(not reduced)} \tag{6}$$

An additional output T is produced, which serves as an intermediate result together with C'_L. To complete the base multiplication in incomplete NTT domain, T is multiplied by ζ in single mode and accumulated to C'_L, again in single mode:

$$C_L = C'_L + T \cdot \zeta \qquad \text{mod } q \tag{7}$$
$$= A_L \cdot B_L + A_H \cdot B_H \cdot \zeta \qquad \text{mod } q \tag{8}$$

Note that ζ is the twiddle factor, a power of the primitive root ψ (see Section 4), and it is distinct for the indices of the element-wise multiplication. The two steps described above could be merged into a single step, as in [AMI+23]. This would cost an additional core multiplier (*e.g.*, a DSP block in FPGA implementations) and an additional independent RAM instance, since both $\{B_L, B_H\}$ and ζ must be read simultaneously. We acknowledge this as a viable alternative design choice but favor the proposed design for its lower hardware resource requirements.

4.2.2 Butterfly Unit (BU)

The architecture of the BU is illustrated in Figure 3. The unit comprises several submodules: a modular adder, a modular subtractor, a modular multiplier, a module that performs modular division by 2, an arithmetic shifter to the right, and shift (delay) registers. All of the mentioned submodules are designed such that they support single and dual modes (and poly mode for the modular multiplier). BU operates in a pipelined manner. The shift registers are used to synchronize the intermediate data for each operation. It takes three 32-bit operands, and outputs two 32-bit results. The modulus q, δ, β, and 2^{-1} are auxiliary data (which are not pipelined) used in related operations, as detailed in the following paragraphs.

The modular adder computes the sum of two operands and, if the result exceeds the modulus q, subtracts q to ensure that the output remains within the valid range. The modular subtractor operates in an analogous manner, performing subtraction followed by addition when necessary. The borrow bits output, b', in the modular subtractor are used as inequality flags in several operations.

Modular Multiplier. The modular multiplier module consists of a 32-bit integer multiplier and a modular reduction unit. The modular reduction is implemented using Barrett reduction method [Bar87]. The integer multiplier is responsible for implementing the poly mode and is realized using the Karatsuba multiplication algorithm [Kar63]. The design rationale behind choosing Barrett reduction is to recover the quotient from the modular reduction operation as well. In other words, the modular multiplier unit is also used as a divider. Computing the quotient is important for supporting functions of PQC such as Compress in ML-KEM and Decompose in ML-DSA. Another advantage is that, with Barrett reduction, the output of the operation is not multiplied with a factor as in Montgomery or Plantard. To support division with arbitrary 32-bit values, rather than restricting the dividend to a function of q^2, we implement the naïve Barrett reduction presented in Algorithm 5 (with $\lambda = 32$), which involves one 64-bit multiplication and one 32-bit multiplication. This is particularly needed when implementing a division needed in masked implementation of ML-KEM, as further discussed in Section 5. For efficient computation, the 64-bit multiplication is also implemented using a recursive Karatsuba algorithm. The pre-computed Barrett constant (denoted by δ in Figure 3) is supplied to the BU through the auxiliary data path. For the 32-bit multiplication in Barrett reduction, only the lower 32 bits of the product are required, resulting in a truncated multiplication. For both 32-bit and 64-bit Karatsuba multipliers, the core multiplication size is set to 16-bit, resulting in 3 and 9 core multiplications, respectively. As the truncated 32-bit multiplication is also achieved with 3 core multiplications of 16-bit, the modular multiplier requires 15 core multiplications in total. Without considering masking, the more efficient Barrett variant in Algorithm 6 could be used (again with $\lambda = 32$). Consequently, the Barrett reduction could be realized with only two 32-bit multiplications (one Karatsuba and one truncated multiplication), reducing the overall cost of modular multiplier to 9 core multiplications.

Next, we detail how different functions implemented by the BU architecture:

- $\mathsf{NTT}, \mathsf{NTT}^{-1}$: the butterfly circuits utilize the modular multiplier, modular adder, and modular subtractor. Specifically, Cooley–Tukey [CT65] (CT) butterflies are used for NTT and Gentleman–Sande [GS66] (GS) butterflies are used for INTT. Note that INTT requires post-processing, where both operands must be multiplied by 2^{-1} after each GS butterfly. The post-processing for one operand is embedded into multiplication by ψ^{-1} performed in the GS butterfly. For the other operand, it is handled using the modular divider by 2 sub-module. This operation does not require multiplication and is implemented as $(x \gg 1) + 2^{-1} \mod q$ using the pre-calculated $(2^{-1} \mod q)$ value.

- Compress, Decompress (ML-KEM, Eq. 1): the Compress operation requires both multiplication and division with rounding. $2^d \times x$ is calculated by the modular multiplier. Then, $q/2$ is added to the result using the modular adder, to simplify rounding using the adder. Finally, the division operation is done using Barrett Reduction. In Decompress, the divisor 2^d is a power-of-two. Therefore, the division is implemented using the arithmetic shifter. These primitives are also used for performing left or right shifts, or divisions. Rounding can be disabled by a control bit. In particular, by setting $q = 1$, left and right shifts can be realized using Compress and Decompress, respectively. Similarly, invoking Compress with $d = 0$ is equivalent to computing the division x/q.

- Decompose, UseHint (ML-DSA, Eq. 2): similar to Compress, Decompose requires division by rounding, which is also efficiently done by the Barrett reduction in the modular multiplier, $r_1 = (x')/\alpha$, $r'_0 = (x + \alpha_h) \mod \alpha$, where $x' = (x + \alpha_h)$ and $\alpha_h = (\alpha - 1)/2$. Note that r_1 and r'_0 are the division and the remainder from dividing x' to α. The added constant is subtracted afterwards using the modular

subtractor for computing the lower-order term, $r_0 = (r'_0 - (\alpha_h)) \mod \alpha$. Moreover, there is a dedicated logic responsible for checking and applying the special case of the Decompose operation, abstracted as special case checkers in Figure 3. The corner case required for this check (by re-purposing 2^{-1} input), as well as α (setting $\beta = \alpha$), is given through the auxiliary data. We should note that the Decompose operation flow can be used to perform Power2Round by disabling the post-processing. UseHint operation is built on top of the Decompose output. BU receives both parts of the output produced by Decompose as its input, namely r_1, r_0, in addition to the hint h. The sign check on r_1 is reformulated as a norm check, $r_1 \leq q/2$, and performed as described in below paragraph. Based on the outcome of this check, r_1 is either incremented or decremented by 1 using the modular adder. We remark that the Decompose operation can be considered as a generic operation that returns both the division and remainder.

- CheckNorm, MakeHint (ML-DSA): The infinity norm check $||\mathbf{x}||_\infty \geq \beta$ over a vector requires performing an inequality check for every element in it, $x \geq \beta$. First, $\beta - 1$ is added to the input data utilizing the modular adder unit, $x' = x + \beta - 1$. As a result, the check becomes $x' \geq (2\beta - 1)$. Then, we perform $x'' = (2\beta - 2) - x'$ using the modular subtractor (x'' is the value before modular reduction). $x \geq \beta$ if and only if x'' is negative, so we output the borrow bits from the modular subtractor. MakeHint uses the exact same steps, with an additional step for checking the special case as in Decompose. The bound $\beta' = \beta - 1$ is also provided through the auxiliary data, (the same signal used in Decompose), as well as the special case for MakeHint. Although these mainly target ML-DSA, inequality check is a generic feature.

4.2.3 Encode Unit (EU)

As discussed in Section 2.2, the decode function maps a sequence of bits to polynomial coefficients, while the encode function performs the inverse operation. In ML-KEM, where $\lceil \log q \rceil = 12$, there is no fixed alignment between the indices of 32-bit words and the 12-bit polynomial coefficients. The situation is the same for ML-DSA as the modulus is 23 bits. As a result, explicit bit shifts are needed to transform between bit strings and polynomial coefficients. As in BU, EU operates based on the word mode, which enables it to handle the word sizes of both ML-KEM and ML-DSA efficiently. The internal resources are shared between encode and decode modes. EU is implemented as a shift-register. Unlike BU, the EU is a stateful unit. It maintains a 64-bit register K, implemented as a shift register, and a counter tracking the number of valid bits stored in K. We intentionally sized K to 64 bits (rather than 32) to provide buffering and prevent stalling in the input/output stream.

4.2.4 Sampling Unit (SU)

The sampling unit is responsible for two main pseudo-random sampling functions, $\mathsf{SamplePolyCBD}_\eta$ and rejection sampling. The former is used exclusively by ML-KEM, while the latter is a core building block in PQC, employed by both ML-KEM and ML-DSA. For both operation modes, NTT-Lite takes a pseudo-random data string (possibly generated by an XOF) as input and applies the logic of these functions to that data. As shown in Figure 2, the SU takes its input from the EU, where the pseudo-random data is first decoded and split into chunks matching the bit-length processed by the SU. The SU contains two modular subtractors, and additional dedicated logic for both $\mathsf{SamplePolyCBD}_\eta$ and rejection sampling. The boundary for rejection sampling is supplied to NTT-Lite via the auxiliary data path. The SU also supports the dual-mode, consistent with the other units. We next describe the implementation details of these two sampling modes.

$\mathsf{SamplePolyCBD}_\eta$ takes $2 \times \eta$ input bits and outputs $(o_0 - o_1)$ where, $o_0 = (\sum_{i=0}^{\eta-1} b_i)$ and $o_1 = (\sum_{i=\eta}^{2\eta-1} b_i)$ with each b_i sampled uniformly from $\{0, 1\}$. Intuitively, o_0 and o_1

are the Hamming weights of two η-bit binary strings, and $\mathsf{SamplePolyCBD}_\eta$ computes the difference between these weights. In our implementation, EU decodes the pseudo-random input to data chunks of $2 \times \eta$ bits, each corresponding to the bits contributing to $(o_0||o_1)$, which are then processed by SU. In SU, each chunk is further decoded to o_0 and o_1 and the subtraction operation is performed.

The rejection sampling uses the first modular subtractor as an inequality checker, to decide $x < \beta$ by looking at the borrow bits. If the test passes, the data is forwarded to the output. In dual-mode, performing rejection sampling on 32-bit inputs requires a 16-bit buffer to handle scenarios in which only one of the two half-words is rejected. In some cases, the output of the rejection sampling must be centered, as in ML-DSA's small-coefficient sampler, whose output is uniform in $[-\eta, \eta]$. To achieve this, we compute $\beta/2 - x$ for centralization. The second modular subtractor performs this operation.

Table 3: Operations supported by NTT-Lite.

Opcode	Source Type	Used By	Active Units	Clock Cycles
OP_NTT	I	K, D	BU	$n'/2 \cdot \log n'$
OP_INTT	I	K, D	BU	$n'/2 \cdot \log n'$
OP_PWM	II	K, D	BU	n' / $2n'$
OP_ADD	II	K, D	BU	n'
OP_SUB	II	K, D	BU	n'
OP_SUM	III	K, D	BU	$n' - 1$
OP_MAC	VI	K, D	BU	n' / $2n'$
OP_COMPRESS	III	K	BU	n'
OP_DECOMPRESS	III	K, D[2]	BU	n'
OP_DECOMPOSE	V	D	BU	n'
OP_CHK_NORM	III	D	BU	$n'^{\star}$
OP_MAKE_HINT	III	D	BU	n'
OP_USE_HINT	VI	D	BU	n'
OP_ENCODE	III	K, D	EU	n'
OP_DECODE	IV	K, D	EU	n'
OP_CBD	IV	K	EU, SU	n'
OP_REJ_SAMP	IV	K, D	EU, SU	[1]

K: ML-KEM, D: ML-DSA.
n' denotes the number of 32-bit words in the data array, which is $n = 256$ for ML-DSA and $n/2 = 128$ for ML-KEM. * : operation is stopped immediately if operation returns failure, takes n' cycle if operation returns success.
[1] : Running time depends on the bound (β) and input data.
[2] : Used in masked ML-DSA implementation, see Section 5.

4.3 High-Level Operations

In this section, we detail how the NTT-Lite operates on vector inputs leveraging the BU, EU, and SU. Table 3 summarizes the set of operations implemented in NTT-Lite. Although the table indicates the primary scheme associated with each operation, this association is not mandatory; the configuration is never hard-coded, ensuring crypto agility. The parameter n', which defines the input array length, is not fixed, and $\log n'$ can be configured to any positive integer up to 8.

4.3.1 Operation Details and Latencies

For `OP_NTT` and `OP_INTT`, there are $\log n'$ stages and $n'/2$ butterfly operations are performed by BU at each stage, leading to roughly $n'/2 \log n'$ clock cycles. We note that each operation incurs a small additional latency due to the pipelined BU, SU, and EU units, as well as from data read and write operations; these latencies are omitted in Table 3 as well as the rest of the discussion for clarity. For example, `OP_NTT` and `OP_INTT` indeed take $n/2 \cdot \log n + \mathcal{L}$ where $\mathcal{L}$ denotes the summation of butterfly and internal memory access latency which is 12 in our implementation.

`OP_SUM` computes the sum of all indices in an input of size n', utilizing the modular addition in BU, taking $n' - 1$ clock cycles. `OP_ADD`, `OP_SUB`, `OP_PWM` (point-wise multiplication), `OP_COMPRESS`, `OP_DECOMPRESS`, `OP_DECOMPOSE`, `OP_MAKE_HINT` are element-wise operations utilizing the corresponding operations from BU. Recall from Section 4.2.1 that `OP_PWM` in poly-mode is handled differently. Since a double pass over the input operands is performed to handle the multiplications by twiddle factors (see Eq. 7), the latency is approximately $2n'$ cycles. `OP_MAC` is similar to `OP_PWM`, while it operates in a multiply-and-accumulate manner, particularly beneficial for matrix multiplications. `OP_CHK_NORM` is also applied element-wise, but terminates early if any element x satisfies $|x| \geq \beta$. This variable-time behavior is acceptable for ML-DSA, as the input is discarded and the process is restarted upon failure.

For `OP_ENCODE`, the n' input elements are streamed to the EU one-by-one, the operation completes once the final 32-bit encoded word is produced. Although the number of output words depends on the encoding parameter d, the total latency is determined by the input length n'. `OP_DECODE` and `OP_CBD` follow a similar pattern, the number of cycles is fixed by the n' output elements, while the required number of input words depends on d. For `OP_REJ_SAMP`, the user specifies the input length, and execution terminates once n' valid samples are generated; otherwise, all input is consumed. Thus, its runtime depends on the modulus, the input size, and the input values themselves.

4.3.2 Data Sources

The source and destination memories for each operation are classified under the *Source Type* column in Table 3. Type I operations require three input sources: two operands are fetched from the 128×32 RAMs and the third from the 256×32 RAM, such as `OP_NTT` and `OP_INTT`. In this case, the third operand corresponds to the twiddle factors. This class produces two outputs, written to the 128×32 RAMs. For the rest of the discussion, the concatenation of two 128×32 RAM is called RAM_0, and 256×32 RAM is called RAM_1. Type II operations require two operands, one is read from RAM_0 and the other is read from RAM_1. Type III operations require only a single input source, which is always read from the RAM_0. For Type IV such as `OP_DECODE`, the input is read from the RAM_1 to prevent overwriting its data prior to processing. For Types II, III, and IV, the output is written to the RAM_0. `OP_DECOMPOSE` produces two outputs: the primary output is written to the RAM_0, whereas the secondary output is stored in the RAM_1. This configuration is indicated as Type V. Finally, `OP_MAC` and `OP_USE_HINT` operations are classified under Type VI, as these have three source operands with a single output. In that case, while the first two operands come from RAM_0 and RAM_1, the third input source is the fresh data that is being transferred to NTT-Lite. That is directly processed without storing it internally.

In order to perform any of these opcodes provided by NTT-Lite, the input data must be transferred to NTT-Lite memory blocks RAM_0 and/or RAM_1, depending on the operation. While the data transfer seems as a performance drawback, in most operations, transferring the input to NTT-Lite memory, either to RAM_0 or RAM_1, and the operation is performed in parallel. Thanks to this overlapping mechanism, one of the transfer costs for the

inputs is eliminated. Moreover, input and output data transfers are not always performed, enabling efficient *chained* operations on data already present in NTT-Lite. This means that NTT-Lite can be programmed to reuse one of the inputs or the output of a previous operation as the input to the next operation. This feature is not hard-coded and fully programmed by the user. All operation codes can be chained.

NTT-Lite also allows the right-hand operand to be a constant, supplied as auxiliary data, which is advantageous in scenarios where one input remains fixed. We leave increasing the size of NTT-Lite memory blocks for further optimization as a future work.

Lastly, NTT-Lite provides a dedicated flag to clear its internal memory at the end of an operation. This is beneficial for avoiding transitional leakage when two shares are processed by sequentially NTT-Lite. This cleaning occurs without additional cost, as it is overlapped with ongoing operations.

4.4 Design Rationale

Many studies in the literature, such as [AMI+23, MCL+23, BNKU+26], propose separate hardware modules for auxiliary functions such as Compress/Decompress, Encode/Decode, Decompose. As a design choice, we do not adopt such an approach for several reasons. First, for area efficiency, we employ only three independent RAM instances (see Section 4.2), which serve as the left- and right-hand-side operand memories (see Section 4.3.2). Consequently, even if these auxiliary functions were implemented outside the BU, only one functionality could be active at any given time, resulting in underutilized hardware resources. Therefore, integrating the auxiliary functions into the BU incurs no performance penalty while reducing the overall area.

Moreover, all arithmetic operations required by these auxiliary functions (except Encode/Decode and pseudo-random sampling) are already supported by the BU. For example, our Barrett reduction unit supports division by an arbitrary divisor, naturally enabling the division operations required by Decompose and UseHint in ML-DSA, as well as Compress in ML-KEM. Recall that support for arbitrary prime moduli in the Barrett reduction stems from our design goal of achieving algorithm agility and supporting the arithmetic required by masked implementations. We also emphasize that all parameters associated with the functionality of NTT-Lite are configurable at run-time, *e.g.*, the parameter d in Compress/Decompress. As discussed in the next section, this run-time configurability also plays an important role in supporting masked implementations. Furthermore, as a feature of NTT-Lite, all operations provided by the BU, EU, and SU support dual mode. In contrast, although [AMI+23] also supports dual mode, only its modular arithmetic units operate in dual mode, whereas auxiliary functions do not.

5 Software and Masked Algorithms

This section describes how the unprotected and fully masked ML-KEM and ML-DSA implementations are realized in the proposed HW/SW co-design solution (Section 3). To this end, we introduce our approaches for masking the non-linear operations (CBD, norm checking, decompose, etc.), which are tweaked to leverage the existing hardware accelerators (NTT-Lite, Keccak and X2X) and maximize overall performance.

5.1 Software Stack and Drivers

Our software stack contains the interface with the RISQrypt hardware accelerators, namely NTT-Lite, Keccak, and X2X. In our proposed design, the software primarily serves an orchestration role, issuing commands to the hardware accelerators and providing memory address pointers for input and output data.

Our implementations cover all high-level APIs of ML-KEM and ML-DSA, using the integrated hardware accelerators that support nearly all underlying subroutines. Hashing and extendable-output functions are handled by the Keccak accelerator, which supports first-order masking. All polynomial arithmetic (which is linear under arithmetic masking) is executed by NTT-Lite using the operation codes described in Section 4, and in a share-wise fashion in the masked implementations. The only exception is the SampleInBall function in Dilithium, which is not computationally intensive and has therefore been left for future work. Between the processing of different shares, the memory-clearing feature of NTT-Lite is utilized (see Section 4.3.2). All remaining non-linear algorithms are efficiently mapped to NTT-Lite and X2X, enabled by the agile design of the architecture.

Programmability. In this work, programmability refers to software-level programmability rather than hardware reconfiguration. The hardware platform is assumed to be fixed (e.g., after tape-out or FPGA bitfile programming), while the accelerator parameters and operation commands are configured by software through device drivers at run-time. For example, the platform can execute ML-KEM by loading the corresponding software, which invokes the hardware accelerators via the driver interface. Without any modification to the underlying hardware, the same platform can instead execute ML-DSA by simply loading the appropriate software. Furthermore, multiple algorithms, such as ML-KEM and ML-DSA, can be included within a single software binary and selected at runtime.

5.2 Masked Algorithms for ML-KEM

Here, we introduce the masked gadgets employed in our masked ML-KEM implementation and present the masked decapsulation algorithm ML-KEM.MaskedDecaps. In order to protect ML-KEM and ML-DSA implementations against differential power analysis attacks, all components that process the secret key need to be masked.

5.2.1 Masked Binomial Sampling

The $\mathsf{MaskedCBD}_\eta$ gadget used in our work is presented in Algorithm 7. It follows the masked generation of a pseudo-random byte string from the sensitive r value in Lines 9-17 of K-PKE.Decrypt, using the Boolean masked PRF (Keccak). The $\mathsf{MaskedCBD}_\eta$ transforms uniformly distributed byte strings (input s in Algorithm 7) into polynomial coefficients (output z in Algorithm 7) that follow the centered binomial distribution over $[-\eta, \eta]$. Note that s refers to the 2η-bit substrings of the output of PRF, and $\mathsf{MaskedCBD}_\eta$ is applied to each substring to generate the required number of polynomial coefficients.

A first-order masked binomial sampler was proposed for the lattice-based NEWHOPE scheme in [OSPG18], and later extended in [SPOG19] to support arbitrary moduli q and masking orders d. In essence, this approach relies on first independently transforming each bit of the 2η-bit Boolean shared string to arithmetic shares (using 2η calls to $\mathrm{B2A}_q$) to compute the difference between their Hamming weights, by subtracting and adding coefficient-wise in the arithmetic domain.

Further, the authors of [SPOG19] propose an alternate high-order masked binomial sampler, which performs the Hamming weight computation directly in the Boolean domain and only requires a single B2A conversion, but uses explicit bitslicing operations for Boolean operations. As the proposed hardware architecture (Section 3) supports efficient mask conversion through the X2X accelerator, there is no need to include additional bitsliced arithmetic in the design (thus avoiding extra area overhead). Instead, we can rely on the existing X2X unit to perform multiple $\mathsf{B2A}_q$ operations efficiently.

To satisfy the required security notion, we introduce an explicit Refresh operation for the input s (Line 2). This is performed in a single command by the X2X accelerator (`OP_REFX2X`) combined with $\mathsf{B2A}_q$, thereby avoiding redundant processing in the pipeline.

Algorithm 7 MaskedCBD($s^{\{0:d\}_B}, \eta$) from [OSPG18, SPOG19]

Input: CBD parameter η
Input: Masking order d
Input: $s^{\{0:d\}_B} \in \mathbb{Z}_{2^{2\cdot\eta}}^{\{0:d\}_B}$
Output: $z^{\{0:d\}_A} \in \mathbb{Z}_q^{\{0:d\}_A}$ following CBD distribution

1: **for** $i = 0$ to $2 \cdot \eta - 1$ **do**
2: $\quad s'^{\{0:d\}_B}[i] = \mathsf{Refresh}(s^{\{0:d\}_B}[i])$ ▷ Using X2X
3: $\quad t^{\{0:d\}_A}[i] = \mathsf{B2A}_q(s'^{\{0:d\}_B}[i])$ ▷ Using X2X
4: **end for**
5: $z^{\{0:d\}_A} = 0$
6: **for** $i = 0$ to $\eta - 1$ **do**
7: $\quad z^{\{0:d\}_A} = z^{\{0:d\}_A} + t^{\{0:d\}_A}[i] - t^{\{0:d\}_A}[\eta + i] \mod q$ ▷ using NTT-Lite
8: **end for**
9: **return** $z^{\{0:d\}_A}$

This operation also leverages the 1-bit mode of X2X, as the $\mathsf{B2A}_q$ requires 1-bit input data. Here, Refresh takes 1-bit input shares and produces output shares in $\mathbb{Z}_q$ which is processed by $\mathsf{B2A}_q$. We emphasize that implementing the bit indexing in hardware is a key factor in reducing the execution latency of MaskedCBD, as the algorithm operates on vector elements independently. In particular, the bit indexing is performed in parallel with the Refresh and $\mathsf{B2A}_q$ operations. Without the dedicated 1-bit mode, bit indexing would have to be carried out explicitly in software, introducing an additional processing step and consequently increasing the overall execution latency. The arithmetic in Line 7 consists of classical chained additions and subtractions implemented in NTT-Lite, using `OP_ADD` and `OP_SUB`.

We now show Algorithm 7 to be t-SNI secure with $t + 1$ shares, providing resistance against a probing adversary with t probes. Additionally, it allows the gadget to be used in larger compositions, such as a full ML-KEM decapsulation.

Theorem 1. *The gadget* MaskedCBD*, presented in Algorithm 7, is t-SNI with $t \leq d$.*

Proof. The proof follows directly from the Refresh being t-SNI, as proposed in [SPOG19, Alg. 20], and the $\mathsf{B2A}_q$ being t-NI, as proven in [NDKV25]. Furthermore, the additions in Line 7 are performed share-wise, and thus can be modeled as a t-NI gadget. The Refresh operation stops the propagation of output/intermediate probes to the input s. □

5.2.2 Masked Compression with Subtraction

The MaskedSubCompress gadget, presented in Algorithm 8, allows to perform the masked compression used in K-PKE.Encrypt and K-PKE.Decrypt, and an optional subtraction. The latter is used to perform the final comparison in ML-KEM.Decaps, where the input b to Algorithm 8 corresponds to a compressed polynomial coefficient from the reference ciphertext c (repeated across all indices) in Algorithm 3.

The input is arithmetically shared, which enables efficient additions (Lines 2-6) using the NTT-Lite, after which the shared data is converted to the Boolean domain using the A2B conversion (X2X accelerator), during which an (optional) mask refresh is performed (Line 7). The conversion is required to perform the final shift (Line 9) in the Boolean domain. The division by q (Line 3) can be efficiently performed via Barrett reduction in NTT-Lite. It is worth noting that existing implementations using reduction mechanisms specific to ML-KEM's q (*e.g.*, [AMI+23]) cannot efficiently compute the quotient. On the other hand, some optimized Barrett reduction variants (such as Algorithm 6) are also

ineffective with their plain application since those assume that the dividend is at most twice the bit-length of q while $(a^{\{i\}_A} \cdot 2^{\delta+\alpha} + q/2) > q^2$.

Furthermore, this algorithm requires 32-bit (single-mode) processing because the dividend exceeds 16 bits, despite ML-KEM's general dual-mode operation. Switching between single-mode and half-mode is handled efficiently using `OP_ENCODE` and `OP_DECODE`, without involving software-level data processing. The shift in Line 9 is achieved via `OP_DECOMPRESS`.

Algorithm 8 MaskedSubCompress$(a^{\{0:d\}_A}, b, \delta)$ from [CGMZ23, DVBV22]

Input: $a^{\{0:d\}_A} \in \mathbb{Z}_q^{\{0:d\}_A}$
Input: $b \in \mathbb{Z}_{2^\delta}$, subtraction operand which is a compressed polynomial coefficient
Input: Compression factor δ
Output: $r^{\{0:d\}_B} \in \mathbb{Z}_{2^\delta}^{\{0:d\}_B}$ such that $r = \mathsf{Compress}_q(a, \delta) - b$

1: $\alpha = \lceil \log_2(q \cdot d) \rceil$
2: **for** $i = 0$ to d **do**
3: $\quad t^{\{i\}_A} = \lfloor (a^{\{i\}_A} \cdot 2^{\delta+\alpha} + q/2)/q \rfloor \mod 2^{\delta+\alpha+1}$ ▷ Using NTT-Lite
4: **end for**
5: $t^{\{0\}_A} = t^{\{0\}_A} + 2^{\alpha-1} \mod 2^{\delta+\alpha+1}$ ▷ Using NTT-Lite
6: $t^{\{0\}_A} = t^{\{0\}_A} - 2^\alpha \cdot b \mod 2^{\delta+\alpha+1}$ ▷ Using NTT-Lite
7: $t'^{\{0:d\}_A} = \mathsf{Refresh}(t^{\{0:d\}_A})$ ▷ Using X2X
8: $t^{\{0:d\}_B} = \mathsf{A2B}_{2^{\delta+\alpha+1}}(t'^{\{0:d\}_A})$ ▷ Using X2X
9: $r^{\{0:d\}_B} = (t^{\{0:d\}_B} \gg \alpha) \mod 2^\delta$ ▷ Using NTT-Lite
10: **return** $r^{\{0:d\}_B}$

Finally, we show that Algorithm 8 is t-SNI secure with $t+1$ shares, allowing it to be used in a larger composition.

Theorem 2. *The gadget* MaskedSubCompress*, presented in Algorithm 8, is t-SNI secure with $t \leq d$.*

Proof. The t-SNI property follows directly from the Refresh (Line 7) being t-SNI [SPOG19, Alg. 20]. Otherwise, the algorithm consists of a series of t-NI operations: all additions and the logical shift are performed in a share-wise fashion and the A2B operation is t-NI, as proven in [NDKV25]. □

We note that, without the t-SNI Refresh in Line 7, the algorithm achieves the t-NI notion. During the full ML-KEM decapsulation, we will use both these variants (and security notions) of Algorithm 8.

5.2.3 Masked Non-Zero Testing

The final step in the ML-KEM decapsulation is the ciphertext comparison, in which the re-computed ciphertext c' is compared to the original ciphertext c. We present our proposed masked gadget for masked non-zero testing in Algorithm 9. Our methodology relies on testing the equality of two integer coefficients u and u' being equivalent to testing if $u - u'$ being zero or not, which is computed by MaskedSubCompress (Algorithm 8). Note that MaskedSubCompress is executed for each element of the input vectors $\mathbf{u}$ and $\mathbf{v}$, prior to invoking Algorithm 9, and the elements of these vectors indeed corresponds to the MaskedSubCompress outputs. The comparison is performed in the masked domain, and on all coefficients at once, as it was shown in [BDH+21] that partial results of the comparison leaks secret information. As such, the comparison is condensed into a single bit, which is unmasked ($\top$ or $\bot$). As the objective of Algorithm 8 is to determine whether any element of the input vectors $\mathbf{u}$ or $\mathbf{v}$ is non-zero, our approach is to merge all coefficients into

a single value by summing them modulo a carefully chosen carrier prime q', for which the accumulated sum is guaranteed not to wrap around the modulus. Compared to the existing approach [CGMZ23, DVBV22, BPO+20], our proposed method for combining the zero-testing into a single masked variable (Lines 1–9) does not require additional randomness and is free from false positives.

The accumulations in Lines 5 and 8 are carried out using `OP_ADD`, *i.e.*, array-wise addition, followed by a final `OP_SUM` to perform accumulation across all indices in the NTT-Lite unit. The carrier prime q' is supported by NTT-Lite thanks to its configurability. Similarly, X2X allows the use of this modulus because it does not restrict the supported moduli to those of ML-KEM or ML-DSA. Random sampling in Line 11 and Refresh in Line 13 are performed using the X2X accelerator, using `OP_PRNG` and `OP_REF`, respectively.

Algorithm 9 MaskedZeroTestVec($\mathbf{u}^{\{0:d\}_B}, \mathbf{v}^{\{0:d\}_B}$)

Input: $\mathbf{u}^{\{0:d\}_B} \in (\mathbb{Z}_{2^{\delta_u}}^{\{0:d\}_B})^{k\times n}$
Input: $\mathbf{v}^{\{0:d\}_B} \in (\mathbb{Z}_{2^{\delta_v}}^{\{0:d\}_B})^{n}$
Input: A carrier prime q' that satisfies $q' > k \cdot n \cdot (2^{\delta_u} - 1) + n \cdot (2^{\delta_v} - 1)$
Output: $\perp$ if any of the elements of $\mathbf{u}$ or v are non-zero. Otherwise, $\top$

1: $t^{\{0:d\}_A} \in \mathbb{Z}_{q'}^{\{0:d\}_A} = 0$
2: **for** $j = 0$ to $n-1$ **do**
3: **for** $i = 0$ to $k-1$ **do**
4: $\mathbf{u}^{\{0:d\}_A}[i][j] = \mathsf{B2A}_{q'}(\mathbf{u}^{\{0:d\}_B}[i][j])$ ▷ Using X2X
5: $t^{\{0:d\}_A} = t^{\{0:d\}_A} + \mathbf{u}^{\{0:d\}_A}[i][j] \mod q'$ ▷ Using NTT-Lite
6: **end for**
7: $\mathbf{v}^{\{0:d\}_A}[j] = \mathsf{B2A}_{q'}(\mathbf{v}^{\{0:d\}_B}[j])$ ▷ Using X2X
8: $t^{\{0:d\}_A} = t^{\{0:d\}_A} + \mathbf{v}^{\{0:d\}_A}[j] \mod q'$ ▷ Using NTT-Lite
9: **end for**
▷ Below lines correspond to [CGMZ23, Alg. 1]
10: **for** $i = 0$ to d **do**
11: $r \leftarrow \mathbb{Z}_{q'}^{*}$ ▷ Using X2X
12: $t^{\{0:d\}_A} = t^{\{0:d\}_A} \cdot r \mod q'$ ▷ Using NTT-Lite
13: $t^{\{0:d\}_A} = \mathsf{Refresh}(\mathsf{t}^{\{0:d\}_A})$ ▷ Using X2X
14: **end for**
15: $t = t^{\{0\}} + t^{\{1\}} + \cdots + t^{\{d\}} \mod q'$ ▷ Unmasking, using NTT-Lite
16: **if** $t \neq 0$ **then return** $\top$
17: **else return** $\perp$
18: **end if**

Correctness For the false positive analysis, we consider whether an element of $\mathbf{u}$ or $\mathbf{v}$ can be non-zero while the algorithm outputs $\perp$. This scenario can only occur if a modular reduction takes place during the summation. However, the carrier modulus q' is chosen such that the accumulated sum never reaches the modulus boundary. Specifically, q' is selected so that even in the worst case, when all elements of $\mathbf{u}$ and $\mathbf{v}$ attain their maximum values, namely $(2^{\delta_u} - 1)$ and $(2^{\delta_v} - 1)$, the total sum remains strictly less than q'. Furthermore, the multiplication in Line 12 does not introduce a false positive, since $r \neq 0$ and q' is a prime, implying that $t^{\{0:d\}_A} \cdot r \equiv 0 \pmod{q'}$ if and only if $t^{\{0:d\}_A} = 0$. On the other hand, if all elements of $\mathbf{u}$ and $\mathbf{v}$ are zero, then $t^{\{0:d\}_A} = 0$, and the algorithm correctly outputs $\perp$, thereby eliminating false negatives.

Theorem 3. *The gadget* MaskedZeroTestVec*, presented in Algorithm 9, is t-NI secure with* $t \leq d$.

Proof. It is shown in [CGMZ23] that Line 10-14 (without final unmasking) is t-NI secure. As the B2A_q (Line 4 & 7) are t-NI and the additions in Line 5 & 8 are performed in each share domain, it follows trivially that the entire gadget is t-NI secure. □

5.2.4 Masked Decapsulation

Finally, all proposed gadgets are considered together to construct a fully masked ML-KEM decapsulation. For this, we consider all relevant steps in the algorithm in Algorithm 3 (and K-PKE subroutines), except the operations involving public values (e.g. public key, ciphertext, etc.), and perform a fine-grained analysis of its composed security. A summary of the proposed construction of the masked ML-KEM decapsulation is depicted in Figure 4.

First, we analyze K-PKE.Decrypt, as shown in Algorithm 2 and map its operations to gadgets as follows:

- G_{1-4}: $\mathsf{ByteDecode}_{12}$ (Line 3), $\hat{\mathbf{s}}^\top \circ \mathsf{NTT}(\mathbf{u}')$ (Line 4), NTT^{-1} (Line 6), subtraction with v' (Line 4). All are linear (polynomial) operations and are performed on each share separately.
- G_5: Compress (Line 5). We use the masked compression gadget in Algorithm 8 (t-NI), without the (optional) subtraction (b set to 0). We refer to this instantiation of Algorithm 8 by MaskedPolyToMsg.

Before the re-encryption in ML-KEM.Decaps, the decrypted message m' and the hash of the public key are hashed (using G hash function) to generate r and K' (mapped to gadget G_6), which are both sensitive. If the final ciphertext comparison is successful, the masked K' is unmasked to perform the final hash in the clear, as proposed in [BGR+21, BDK+21, KDVB+22]. It is critical that the short-term secret K' is only revealed if the comparison passes, and not in case of failure as then it could be used to recover the long-term secret key. The G operation, which relies on Keccak, is masked as proposed in prior art [GSM17] and integrated in our accelerator/peripheral.

Next, we analyze K-PKE.Encrypt (Algorithm 1), focusing on operations involving secret data:

- G_{7-9}: $\mathsf{PRF}(r, N)$ (Lines 10-15). This operation is also a symmetric primitive and implemented using the masked Keccak peripheral.
- G_{10-12}: SamplePolyCBD (Lines 10-15) to generate $\mathbf{y}, \mathbf{e}_1, e_2$. We utilize our proposed masked binomial sampler (Algorithm 7), which uses the X2X peripheral to accelerate the mask conversion.
- G_{13-16}: $\mathsf{NTT}(\mathbf{y})$ (Line 16), $\hat{\mathbf{A}}^\top \circ \hat{\mathbf{y}}$ (Line 17), NTT^{-1} (Line 17), addition with $\mathbf{e}_1$ (Line 19). All are linear operations and thus are masked by implementing them in a share-wise fashion.
- G_{17}: $\mathsf{Decompress_q}(\mathsf{ByteDecode}_1((m), 1))$ (Line 18). As before, the decoding operation is trivial to mask. Then, the Boolean shared message m' is converted to arithmetic shares with a B2A_q using the X2X peripheral, and multiplied with a constant. We refer to this gadget as MaskedPolyFromMsg. The MaskedPolyFromMsg algorithm (see G_{17} in Section 5.2.4) also leverages the 1-bit mode of X2X to perform the B2A efficiently on 1-bit input.
- G_{18-21}: $\hat{\mathbf{t}}^\top \circ \hat{\mathbf{y}}$, NTT^{-1}, addition with e_2, addition with decompressed message (Line 19). All are linear operations and are performed on each share.
- G_{22-23}: Compress (Line 20). The re-computed ciphertext components are compressed, and subtracted with the public (input) ciphertext $\mathbf{u}, v$, using the proposed

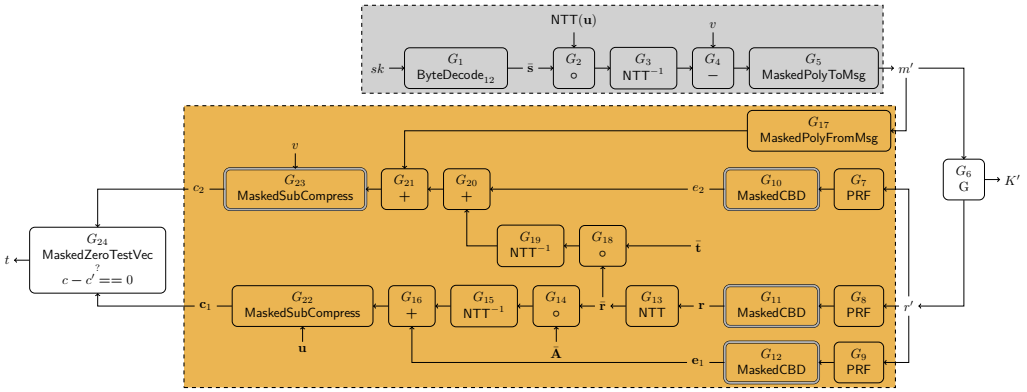


Figure 4: An abstract diagram of the ML-KEM.MaskedDecaps, consisting of the decryption (gray) and re-encryption (yellow). The t-NI gadgets are depicted with a single border, the t-SNI gadgets with a double border.

MaskedSubCompress gadget (Algorithm 8). To compute the ciphertext component c_2, the (optional) explicit mask refresh in Algorithm 8 is enabled to ensure the composition is secure.

Finally, the masked ciphertext comparison is performed on the (masked) compressed ciphertext ($\mathbf{c}_1$ and c_2) using the secure zero check in Algorithm 8, mapped to G_{24}. As shown in Section 5.2.3, t is unmasked as a final step to determine if the comparison is successful (or not). Note that Figure 4 excludes the unmasking of t as well as short-term secret K'.

We now show that our proposed masking of the ML-KEM decapsulation is t-NI secure with $t+1$ shares, and as a result also t-probing secure. In the proof, we iterate over all output and intermediate variables and show how each can be simulated using only a limited set of input shares, based on the t-(S)NI properties of the individual gadgets.

Theorem 4. *The gadgets in Figure 4 (ML-KEM.MaskedDecaps) are t-NI secure with $t \leq d$.*

Proof. All linear operations, in both the decryption and (re-) encryption, are modeled as t-NI gadgets as these are performed in a share-wise fashion (e.g. G_{1-4}, G_{13-16}, G_{18-21}). Additionally, all gadgets that perform symmetric key operations (G, PRF) are implemented as proposed in [GSM17] using t-NI DOM gates. As a result, the gadgets G_6 and G_{7-9} are modeled as t-NI gadgets. The Compress algorithm is implemented using the proposed Algorithm 8, which is proven to be t-SNI secure (G_{23}) or t-NI (G_5, G_{22}). Similarly, the Decompress operation consist of the t-NI B2A operation (X2X) and share-wise multiplication, and thus is modeled as the t-NI gadget G_{17}. The sampling operation (G_{10-12}) are implemented as the t-SNI gadget in Algorithm 7. The final comparison operation is implemented as proposed in Algorithm 9, and thus G_{24} is t-NI.

An adversary can probe any of these gadgets G_i internally (t_{G_i} probes) or their output shares (o_{G_i} probes), except the final outputs K' and t (due to t-NI property). In total, at most $t_{G_{\text{MaskedDecaps}}}$ internal probes can be placed:

$$t_{G_{\text{MaskedDecaps}}} = \sum_{i=1}^{24} t_{G_i} + \sum_{i=1}^{5} o_{G_i} + \sum_{i=7}^{23} o_{G_i}$$

The proof now consists of showing that any internal and output probes can be perfectly simulated with $\leq t_{G_{\text{MaskedDecaps}}}$ shares of the input sk, and that the required set of input shares $|I|$ is independent of $|O|$. To simulate the internal and output probes in the gadgets

of the decryption operation ($G_1 - G_5$, gray in Figure 4), a total of $t_{\text{gray}} = \sum_{i=1}^{5} t_{G_i} + o_{G_i}$ shares of the input sk are required due to the t-NI property of the gadgets. The total amount of shares of the inputs (m' and r') required to simulate all probes of the yellow zone (encryption) are:

$$t_{\text{yellow}} = t_{G_7} + o_{G_7} + t_{G_8} + o_{G_8} + t_{G_9} + o_{G_9} + t_{G_{10}} + t_{G_{11}} \\ + t_{G_{12}} + t_{G_{17}} + o_{G_{17}} + t_{G_{21}} + o_{G_{21}} + t_{G_{23}}$$

As we can see, the t-SNI property of $G_{10} - G_{12}$ and G_{23} stops the propagation of probes.

Finally, we sum all share contributions from the yellow and gray zone, to determine the set of input shares required for simulating the entire composition:

$$|I| = t_{\text{gray}} + t_{G_6} + t_{\text{yellow}}$$

As $|I| \leq t_{G_{\mathsf{MaskedDecaps}}}$, the composed gadget satisfies the t-NI notion. □

5.3 Masked Algorithms for ML-DSA

Here, we describe the masked gadgets employed in our masked ML-DSA implementation and present the masked signing algorithm ML-DSA.MaskedSign.

5.3.1 Masked Uniform Sampler

The masked uniform sampler, here referred to as MaskedUniform (Algorithm 10), is utilized to mask the function ExpandMask and generate $\mathbf{y}$ securely (Line 11, Algorithm 4). MaskedUniform takes as input the Boolean masked string s and the distribution parameter γ. Note that the string s indeed corresponds to substrings of the output of XOF, which takes as input the secret seed ρ'. Algorithm 10 is then executed across all substrings to generate the polynomial coefficients of $\mathbf{y}$. Each share of the input string s is of γ bits. This masked algorithm outputs arithmetic masked r, where each coefficient of each share belongs to $\mathbb{Z}_q$ and each coefficient of r follows the uniform distribution in $[-2^{\gamma-1}+1,\ 2^{\gamma-1}]$. All operations in this gadget can be directly implemented on our architecture using the NTT-Lite and X2X hardware accelerators. The consecutive Refresh and $\mathsf{B2A}_q$ operations are executed within a single pipeline invocation using the `OP_REFX2X` instruction (similar to Section 5.2.1). Furthermore, the subtraction operations performed by NTT-Lite in Lines 3 and 5 exploit its constant-operand feature, thereby eliminating the need to initialize and store a vector consisting entirely of zeros (and $2^{\gamma-1}$) in the NTT-Lite memory.

Algorithm 10 MaskedUniform$(s^{\{0:d\}_B}, \gamma)$

Input: Parameter γ
Input: $s^{\{0:d\}_B} \in \mathbb{Z}_{2^\gamma}^{\{0:d\}_B}$
Output: $r^{\{0:d\}_A} \in \mathbb{Z}_q^{\{0:d\}_A}$, where r follows uniform distribution in $(-2^{\gamma-1}, 2^{\gamma-1}]$
1: $s^{\{0:d\}_B} = \mathsf{Refresh}(s^{\{0:d\}_B})$ ▷ Using X2X
2: $r^{\{0:d\}_A} = \mathsf{B2A}_q(s^{\{0:d\}_B})$ ▷ Using X2X
3: $r^{\{0\}_A} = 2^{\gamma-1} - r^{\{0\}_A} \mod q$ ▷ Using NTT-Lite
4: **for** $i = 1$ to d **do**
5: $\quad r^{\{i\}_A} = 0 - r^{\{i\}_A} \mod q$ ▷ Using NTT-Lite
6: **end for**
7: **return** $r^{\{0:d\}_A}$

Theorem 5. *The gadget* MaskedUniform*, presented in Algorithm 10, is t-SNI with $t \leq d$.*

Proof. It follows an identical approach to the proof of Theorem 1. Refresh is a t-SNI gadget and B2A_q is a t-NI gadget. The subtractions in Lines 3 and 5 are performed in a share-wise manner, preserving the t-NI security. Therefore, the gadget MaskedUniform, a composition of t-SNI and t-NI gadgets, also achieves t-SNI security. □

5.3.2 Masked Norm Check

The masked norm check algorithm, referred to as MaskedCheckNorm_w (Algorithm 11), is used to securely perform both rejection checks $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ and $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ (Algorithm 4). MaskedCheckNorm_w takes arithmetic shares of a, the word size parameter w, and the bound β as input. It returns a masked 1-bit output r, which is 0 if the following equation holds: $q - \beta \geq a \geq \beta$ and otherwise 1. Our MaskedCheckNorm_w is adapted from [ABC+23], and can be mapped to both the NTT-Lite and X2X accelerators. Notably, the subtraction in Line 4 is performed modulo 2^{32} by NTT-Lite, once again exploiting its configurable modulus support. This operation further illustrates that an architecture restricted to the 12- and 24-bit moduli used in ML-KEM and ML-DSA (such as [AMI+23]) would be incapable of supporting the required computation. Moreover, the additions and subtractions in Lines 1 and 4 (implemented using `OP_ADD` and `OP_SUB`) leverage the constant-operand feature of NTT-Lite, as in the previous subsection. In Line 6, the right-shift is implemented using `OP_DECOMPRESS`, demonstrating the generality of this command, although it was originally designed for ML-KEM.

Algorithm 11 $\text{MaskedCheckNorm}_\omega(a, \beta)$ adapted from [ABC+23]

Input: Bound β
Input: Word-size parameter $\omega \geq \lceil \log q \rceil + 1$, *e.g.*, $\omega = 32$
Input: $a^{\{0:d\}_A} \in \mathbb{Z}_q^{\{0:d\}_A}$
Output: $r^{\{0:d\}_B} \in \mathcal{B}_1^{\{0:d\}_B}$ such that $r = 0$ if $(q - \beta \geq a \geq \beta)$. Otherwise, $r = 1$

1: $a^{\{0\}_A} = a^{\{0\}_A} + \beta - 1 \mod q$ ▷ Using NTT-Lite, the check becomes $(a \geq 2\beta - 1)$ ▷ Modulus switching
2: $a^{\{0:d\}_B} = \text{A2B}_q(a^{\{0:d\}_A})$ ▷ Using X2X
3: $t^{\{0:d\}_A} \in \mathbb{Z}_{2^\omega}^{\{0:d\}_A} = \text{B2A}_{2^\omega}(a^{\{0:d\}_B})$ ▷ Using X2X
4: $t^{\{0\}_A} = t^{\{0\}_A} - (2\beta - 1) \mod 2^\omega$ ▷ Using NTT-Lite, the check becomes $(t \geq 0)$
5: $t^{\{0:d\}_B} = \text{A2B}_{2^\omega}(t^{\{0:d\}_A})$ ▷ Using X2X
6: $r^{\{0:d\}_B} = t^{\{0:d\}_B}[\omega - 1]$ ▷ Using NTT-Lite, logical shift to right by $\omega - 1$-bit
7: **return** $r^{\{0:d\}_B}$

Now, we show that Algorithm 11 is t-NI secure.

Theorem 6. *The gadget* MaskedCheckNorm_w*, presented in Algorithm 11, is* t*-NI with* $t \leq d$*.*

Proof. In Algorithm 11, the B2A_q on Line 2, the B2A_{2^w} on Line 3, and the A2B_{2^w} on Line 5 are t-NI. The addition in Line 1, the subtraction in Line 4, and the logical shift operation in Line 6 are performed in a share-wise manner, thus preserving the t-NI security. Therefore, the gadget MaskedCheckNorm_w, a composition of t-NI gadgets, also achieves t-NI security overall. □

5.3.3 Masked Decompose Function

The masked decompose function, MaskedDecompose (Algorithm 12), is used to extract high and low bits of w securely (Lines 13 and 21, Algorithm 4). MaskedDecompose takes arithmetic shares of a and a decomposition parameter γ as input and returns unmasked high bits r_1 and arithmetic masked shares of low bits r_0, where unmasked $a = r_1 \cdot \delta + r_0$ and

$\delta = (q-1)/\gamma$. This algorithm can also be efficiently mapped onto both the NTT-Lite and X2X accelerators. The multiplications, additions, and subtractions in Lines 1, 2, and 13 are implemented using the `OP_PWM`, `OP_ADD`, and `OP_SUB` instructions, while exploiting the constant-operand feature for multiplications by δ and γ. Furthermore, the consecutive $\mathsf{Refresh}$ and $\mathsf{B2A}_\delta$ operations in Lines 9 and 10 are executed by the X2X accelerator in a single command using the `OP_REFX2X` instruction, operating using a modulus different from the standard ML-KEM or ML-DSA primes. The XOR ($\oplus$) operation required by Boolean unmasking in Line 7 is currently not implemented in NTT-Lite. However, it is trivial to add, and performing it in software does not introduce micro-architectural leakage concerns, as the data is no longer security-sensitive.

Our $\mathsf{MaskedDecompose}$ closely follows $\mathsf{SecDecomposeComp}$ from [ABC+23]. As proven in [ABC+23], the $\mathsf{MaskedDecompose}$ in Algorithm 12 is t-NI secure when r_1 is the public output.

Algorithm 12 $\mathsf{MaskedDecompose}(a^{\{0:d\}_A}, \gamma)$ from [CGTZ23]

Input: $a^{\{0:d\}_A} \in \mathbb{Z}_q^{\{0:d\}_A}$
Input: Decomposition parameter γ
Output: $r_1, r_0^{\{0:d\}_A}$ such that $(r_1, r_0) = \mathsf{Decompose}_q(a, \gamma)$

1: $\delta = (q-1)/\gamma$
2: $t^{\{0:d\}_A} = -a^{\{0:d\}_A} \cdot \delta \mod q$ ▷ Using NTT-Lite
3: $t^{\{0\}_A} = t^{\{0\}_A} + (q-1)/2 \mod q$ ▷ Using NTT-Lite
4: $t^{\{0:d\}_B} = \mathsf{A2B}_q(t^{\{0:d\}_A})$ ▷ Using X2X
5: **if** δ is a power-of-two **then**
6: $\quad r_1^{\{0:d\}_B} = t^{\{0:d\}_B}[\log \delta - 1 : 0]$
7: $\quad r_1 = r_1^{\{0\}_B} \oplus r_1^{\{1\}_B} \oplus \cdots \oplus r_1^{\{d\}_B}$ ▷ Unmasking
8: **else**
9: $\quad t^{\{0:d\}_B} = \mathsf{Refresh}(t^{\{0:d\}_B})$ ▷ Using X2X
10: $\quad r_1^{\{0:d\}_A} = \mathsf{B2A}_\delta(t^{\{0:d\}_B})$ ▷ Using X2X
11: $\quad r_1 = r_1^{\{0\}_A} + r_1^{\{1\}_A} + \cdots + r_1^{\{d\}_A} \mod \delta$ ▷ Unmasking, using NTT-Lite
12: **end if**
13: $r_0^{\{0:d\}_A} = a^{\{0:d\}_A} - \gamma \cdot r_1 \mod q$ ▷ Using NTT-Lite
14: **return** $r_1, r_0^{\{0:d\}_A}$

5.3.4 Masked Signing

We combine all masked gadgets to construct the fully masked ML-DSA signing operation: $\mathsf{ML\text{-}DSA.MaskedSign}$, shown in Figure 5. For our masked signing algorithm, we follow the sensitive analysis proposed in [ABC+23] and mask the same variables.

Now, we map $\mathsf{ML\text{-}DSA.Sign}$ to the masked domain:

- G_{1-2}: As part of the $\mathsf{ExpandMask}$ function (Line 7), masked Keccak is used as the gadget G_1 first, and after that $\mathsf{MaskedUniform}$ (Algorithm 10) is used as the gadget G_2 to sample $\mathbf{y}$ from $[-2^{\gamma-1}+1,\ 2^{\gamma-1}]$ following the uniform distribution.
- G_{3-5}: $\mathsf{NTT}(y)$, $\hat{\mathbf{A}} \circ \mathsf{NTT}(y)$, and $\mathsf{NTT}^{-1}(\mathsf{NTT}(\hat{\mathbf{A}} \circ \mathbf{y}))$ of Line 8 are all linear operations and are performed securely by applying them share-wise (G_3, G_4, and G_5).
- G_6: The decomposition function generates unmasked $\mathbf{w}_1$ used in $\mathsf{HighBits}(\mathbf{w})$ (Line 9) and the arithmetic shares of $\mathbf{w}$ used in $\mathsf{LowBits}(\mathbf{w} - c\mathbf{s_2})$ (Line 15). It is performed using the $\mathsf{MaskedDecompose}$ function in Algorithm 12 as the gadget G_6.

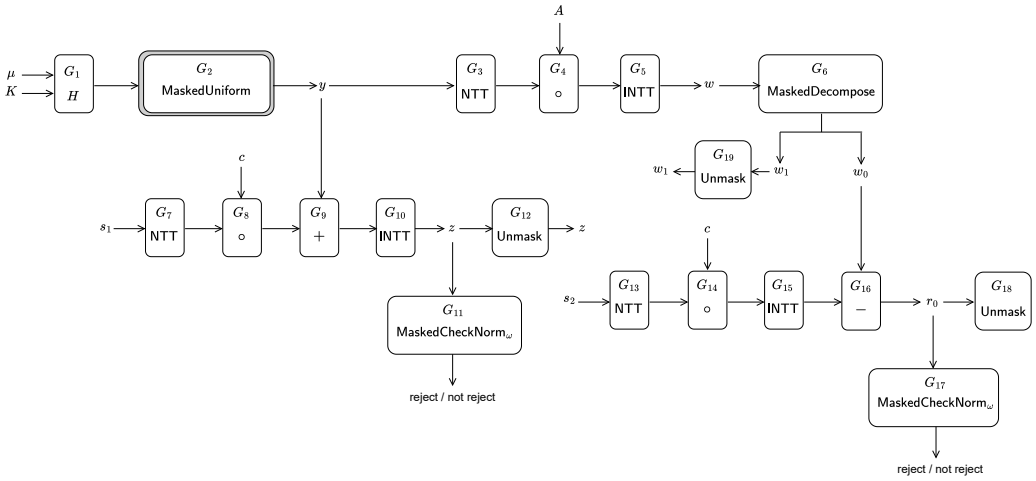


Figure 5: An abstract diagram of the ML-DSA.MaskedSign. All the gadgets are at least t-NI. The t-NI gadgets are depicted with a single border, while the t-SNI gadget MaskedUniform is depicted with a double border.

- G_{7-10}: $\mathsf{NTT}(\mathbf{s_1})$, $\mathsf{NTT}(c) \circ \mathsf{NTT}(\mathbf{s_1})$, $\mathsf{NTT}(\mathbf{y}) + (\mathsf{NTT}(c) \circ \mathsf{NTT}(\mathbf{s_1}))$, and $\mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{y}) + (\mathsf{NTT}(c) \circ \mathsf{NTT}(\mathbf{s_1})))$ are all linear and can be masked by applying them share-wise. These are implemented using G_7, G_8, G_9, and G_{10} gadgets, respectively.
- G_{13-16}: Similarly, $\mathsf{NTT}(\mathbf{s_2})$, $\mathsf{NTT}(c) \circ \mathsf{NTT}(\mathbf{s_2})$, $\mathsf{NTT}^{-1}(\mathsf{NTT}(c) \circ \mathsf{NTT}(\mathbf{s_2})$, and $\mathbf{w} - c\mathbf{s_2}$ are also linear and can be masked by applying them share-wise. These masked operations are carried out with gadgets G_{13}, G_{14}, G_{15}, and G_{16}, respectively.
- G_{11} and G_{17}: The gadget $\mathsf{MaskedCheckNorm}_\omega$, presented in Algorithm 11 is used as G_{11} and G_{17}. G_{11} and G_{17} are used to perform the first two rejection checks $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ and $||\mathbf{r}_0||_\infty \geq \gamma_2 - \beta$ (Line 16).
- G_{12}, G_{18}, and G_{19}: These gadgets are used to unmask $\mathbf{z}$, $\mathbf{r}_0$, and w_1 securely.

Similar to the masked signing algorithm of ML-DSA presented in [ABC+23], all gadgets of our masked signing algorithm are at least t-NI, and public variables are securely unmasked. Therefore, the masked signing algorithm achieves t-NI security, given the signature $\sigma = (\mathbf{z},\ \mathbf{h}, \bar{c})$ is a public output.

5.4 Randomness Assumptions

The t-NI and t-SNI proofs considered in this work assume that all masking gadgets receive fresh, independent, and uniformly random values. In our implementation, these values are generated by an LFSR-based PRNG seeded by a TRNG and periodically refreshed. This hybrid TRNG-PRNG construction is commonly used in secure hardware designs to reduce the cost of generating huge amounts of randomness while maintaining access to entropy. Our security proofs are stated under the assumption of independent and unpredictable randomness and therefore do not depend on any specific property of the implemented PRNG. Rather, the PRNG serves solely as a practical instantiation of the randomness source required by the masking gadgets. Prior work has shown that combining multiple output bits of an LFSR into a single masked implementation can cause violations of probing security in very specific cases [CMM+24, GSH+26]. However, our experimental evaluation does not indicate any degradation of masking security attributable to the

implemented PRNG (see Section 6.1). We acknowledge that the formal proofs do not cover potential weaknesses of the underlying randomness generator. Therefore, in high-assurance deployments, the LFSR-based PRNG may be replaced by a secure stream cipher or another secure pseudorandom generator. Such a substitution would not require any modification to the masking gadgets or the associated t-NI or t-SNI proofs, since these gadgets and proofs only depend on the availability of independent and unpredictable randomness.

6 Results

In this section, we evaluate the proposed solution in terms of security and performance, and compare it with existing works in the literature. All source code related to the RISQrypt project is publicly accessible at https://github.com/cisec-su/RISQrypt . This includes the source code of the hardware accelerators, the associated SDKs, software layers for PQC implementations, and SCA scripts including trace acquisition and testing. Since the finalized standards introduce minor modifications to the control flow only in the software layer, while leaving the underlying hardware architecture unchanged and having negligible impact on performance, we base our evaluation on the Round 3 submission versions of ML-KEM and ML-DSA.

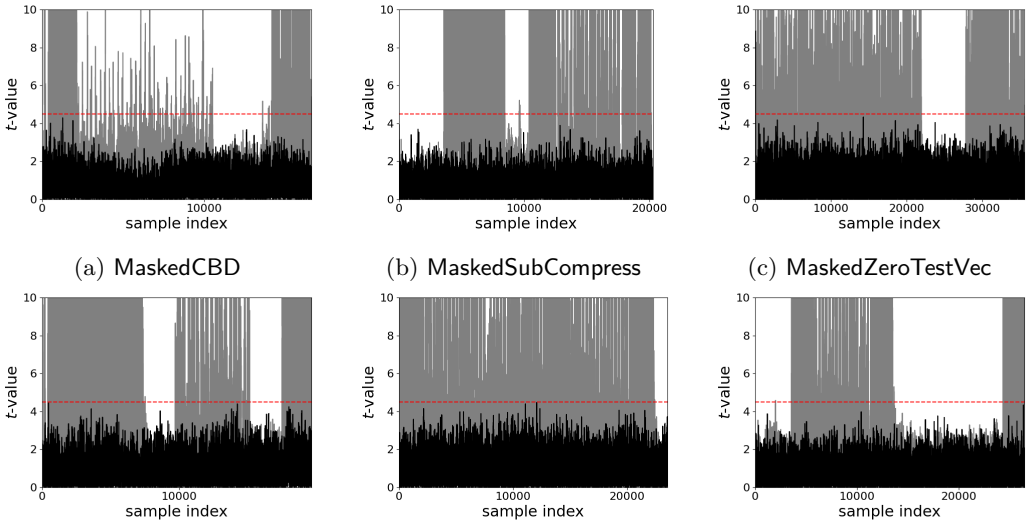
6.1 Side-Channel Leakage Evaluation

Although we give a formal security proof for our side-channel-protected design, a practical evaluation is still necessary to uncover implementation issues such as transitional leakage. To this end, we performed a Test Vector Leakage Assessment (TVLA), a standard procedure widely used in the side-channel analysis literature [GGJR11]. In TVLA, two sets of traces are captured from the evaluated implementation. A trace is a set of power leakages recorded during the execution of cryptographic function. One of the sets is captured with a random input, while the other is captured with a fixed input. The masked implementation is approved if the two trace sets are statistically not different from each other, which evidences that the power leakage does not depend on secret data. Welch's t-test is used as the statistical tool to compare the random-input and the fixed-input set, which computes a metric called t-value. The greater the t-value, the more different both sets are from each other. In the test, the null hypothesis proposes that the fixed-input and random-input trace sets follow the same statistics. A typical threshold for t-value is 4.5 in absolute value, which provides 99.999% confidence to reject the null hypothesis.

We used the ChipWhisperer framework for the experiments. CW Husky[2] is used for trace capturing while CW305[3] target board, which is equipped with an Artix-7 FPGA (XC7A100T), is used as the victim device running our implementation. The core clock of the victim device is set to 40 MHz. The trace collector facility is configured to collect four samples at each clock cycle. To validate our experimental setup, we also include the t-test results where the PRNG in the system is disabled. For this scenario, we collected 1K traces, whereas 100K/500K traces were used when PRNG was enabled. We captured the fixed-input and random-input sets by randomly interleaving them to mitigate environmental bias [TG16].

We first evaluated the masked non-linear algorithms discussed in Section 5. The results of corresponding t-test experiments are presented in Figure 6. Except for MaskedZeroTestVec, the algorithms were presented in single-word form in Section 5, while the t-test interval corresponds to the execution of these algorithms on the entire polynomial in $\mathcal{R}_q$. Moreover, we include the pseudo-random sampling performed by Keccak accelerator prior to execution of MaskedCBD and MaskedUniform, as these process the pseudo-random data produced

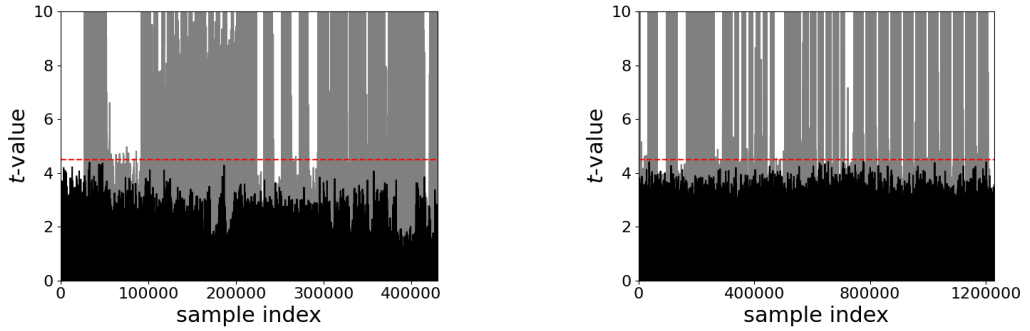
[2] https://rtfm.newae.com/Capture/ChipWhisperer-Husky/
[3] https://rtfm.newae.com/Targets/CW305%20Artix%20FPGA/



(a) MaskedCBD (b) MaskedSubCompress (c) MaskedZeroTestVec

(d) MaskedUniform (e) MaskedCheckNorm (f) MaskedDecompose

Figure 6: TVLA results for masked algorithms under two configurations: `PRNG_OFF` (gray) and `PRNG_ON` (black).



(a) ML-KEM.MaskedDecaps (b) ML-DSA.MaskedSign

Figure 7: TVLA results for high-level functions under two configurations: `PRNG_OFF` (gray) and `PRNG_ON` (black).

(see Figure 4 and Figure 5). For all evaluated algorithms, with the randomness disabled (`PRNG_OFF`), leakage is clearly observed after acquiring only 1K traces, as t-values are significantly higher than 4.5. On the other hand, when the PRNG is on (`PRNG_ON`), the t-values remain lower than 4.5, for both algorithms, with 500K traces.

Next, we evaluated the sensitive high-level functions, namely both ML-KEM.MaskedDecaps and ML-DSA.MaskedSign. In this analysis, the input refers to the secret key used by these functions. For ML-DSA.MaskedSign, the t-test is complex due to the algorithm being non-constant time (e.g., rejection sampling). We therefore analyze only a single iteration of the rejection sampling loop and ignore rejections within that iteration. The high-level analysis does not cover leakage across repeated rejection iterations, which is outside the scope of this work.

Figure 7 demonstrates the t-test results for both algorithms. In line with our initial analysis, leakage is clearly observed after acquiring only 1K traces for `PRNG_OFF` setting, while t-values do not exceed 4.5 for `PRNG_ON` setting, indicating that no leakage was

detected under the tested conditions. Owing to the considerably longer execution time of the high-level functions, the t-test experiments were conducted using 100K traces, limiting the statistical coverage of the analysis compared with the lower-level evaluations. We also note that since our t-test analysis is performed using a large number of samples, false positives (Type I error) are expected [DZD+17, BGG+14]. In [BGR+21], the authors use a more conservative threshold of 6.88 to reduce the probability of Type I error. Instead, we repeated the tests over the same intervals and consider effects that were not reproducible as false positives.

6.2 Performance Evaluation

To assess the effectiveness of the proposed hardware–software co-design, we conducted a comprehensive performance evaluation and compared it with existing implementations. By benchmarking against representative designs, we highlight the advantages of our approach in achieving lower latency while maintaining competitive hardware costs.

6.2.1 Hardware Overhead

Table 4: FPGA resource overhead of cryptographic modules and masking.

Configuration		LUT	FF	DSP	BRAM	Freq.
Crypto. Acc.	Mask.					(MHz)
✗	NA	5488	2703	0	80	63
✓	✗	16207	8869	9*	80.5	61
Crypto Overhead		10719	6166	9*	0.5	
✓	✓	30556	22203	15	80.5	61
Crypto Overhead		25068	19500	15	0.5	

* modular multiplier is not optimized for the unmasked scenario and theoretical number is reported. Maximum synthesisable frequencies are reported.

In Table 4, we report the resource overhead of cryptographic accelerators in the proposed design. We implemented our design on the Artix-7 XC7A100T-1CSG324C FPGA device using Xilinx Vivado 2023.2 with the balanced option. In the table, we report the FPGA resource consumption of three configurations. The first one is the base processor. Cryptographic accelerators are excluded, and the system contains only the RISC-V processor, memory, and peripherals. In the second one, cryptographic accelerators are included, but masking is disabled as a reference. The X2X accelerator is absent, and the Keccak accelerator does not support masking. This configuration serves as a performance and area reference. The third one is our proposed design, where cryptographic accelerators are included with full masking support. By comparing configurations with cryptographic acceleration, we quantify the overhead introduced by acceleration, both with and without masking. The reported results align with our design expectations. Overall, the reported hardware overhead represents a reasonable cost for achieving both security and high performance. The maximum frequency of the system decreases modestly when cryptographic acceleration is enabled. However, timing analysis shows that the critical path lies on the Wishbone bus. This is due to the increased overall design size and a more constrained place-and-route process, rather than within the internal logic of the cryptographic accelerators.

Table 5 presents a detailed breakdown of resource utilization for individual modules. For NTT-Lite, the BU consumes the majority of LUTs and all DSPs, as it implements the core polynomial arithmetic. The cost of EU and SU are relatively negligible. For Keccak and X2X, the core units also consume the largest portion of resources. In Keccak, FIFO and FSM also consume notably. The sponge function logic could be implemented in software to reduce FSM cost at the expense of performance (see Section 3.2). We note that disabling masking in Keccak approximately halves its resource consumption. Accordingly, the cost of the modular multiplier in NTT-Lite decreases by 6 DSPs when masking is not considered (see Section 4.2.2). In X2X, the PRNG also consumes a significant number of LUTs; however, A2B/B2A conversion methods with lower randomness requirements could reduce this overhead, which we leave for future work. We also report the maximum operating frequency achievable by each hardware accelerator. We remark that the accelerators do not operate at these frequencies in the architecture evaluated in this section. Rather, these values represent the maximum operating frequencies attainable when the individual accelerator designs are synthesized independently.

Next, we present the results of ASIC synthesis for the proposed design in Table 6. The design is synthesized using the TSMC 65 nm Low-Power (LP) standard cell library. We evaluated the design at two target frequencies: 100 MHz and 500 MHz. The latter incurs an area overhead of approximately 6% relative to the 100 MHz baseline, resulting from the timing-driven optimizations required to meet the stricter timing constraints. Notably, NTT-Lite consumes significantly more area than the others, primarily due to its memory (BRAM-equivalent) requirements. Further area optimization of NTT-Lite for ASIC implementations is left as future work. [AMI+23] reports an ASIC area of 1.107 mm^2 in 65 nm technology. Recall that [AMI+23] presents a unified pure hardware accelerator for ML-KEM and ML-DSA; however, its parameters are fixed, and therefore the architecture does not support additional algorithms as our platform does. Furthermore, [AMI+23] does not provide masking support. Nevertheless, its reported area is 2.06×/1.99× larger than the ASIC area of our design (with masking enabled) synthesized at 100/500 MHz. This difference primarily stems from the low memory requirements of NTT-Lite, which requires only three independent RAMs (see Section 4.2 and Section 4.4), whereas [AMI+23] requires 24 BRAMs in their FPGA implementation.

Table 5: FPGA resource utilization of individual modules.

Module	LUT	FF	DSP	BRAM	Freq. (MHz)
NTT-Lite	5547	2677	15	0.5	100
BU	2824	1389	15	0	
EU	968	254	0	0	
SU	379	160	0	0	
FSM + RAM	1376	874	0	0.5	
Keccak	9787	7626	0	0	111
Core	5514	4022	0	0	
FIFO	1024	3200	0	0	
FSM	3249	404	0	0	
X2X	9417	9163	0	0	71[1] / 100[2]
Core	3805	5084	0	0	
PRNG	3344	1090	0	0	
FSM + Reg. File	2268	2989	0	0	

[1] Half-cycle mode. [2] Full-cycle mode, which increases the LUT/FF usage by 140/1939.

Table 6: ASIC synthesis results of cryptographic modules.

Module	Area (mm^2) @ 100 MHz	Area (mm^2) @ 500 MHz
NTT-Lite	0.281	0.297
Keccak	0.132	0.132
X2X	0.123	0.126
Crypto total	0.536	0.555

6.2.2 Time Performance

Table 7: Time performance comparison with prior work.

Work	ML-KEM-768				ML-DSA-65			
	KeyGen	Enc	Dec	MaskedDec	KeyGen	Sign	Verify	MaskedSign
RISQrypt	**23**	**28**	**35**	**109**	**120**	**480**	**126**	**1230**
[FVR+22]	214	298	313	1235	–	–	–	–
[WZZ+24]	536	671	639	–	2087	5003	1986	–
[DMMM23]	663	856	1083	–	–	–	–	–
[AOP+25]	62	72	83	–	234	476	216	–
[NDMZ+21]	695[1]	732[1]	862[1]	–	2975	10212	2964	–
[MCL+23]	–	–	–	–	264[2]	546[2]	281[2]	–
[MBB+23]	–	–	–	–	479	3988	511	–
[KSFS24]	–	–	–	–	1068	3253	1127	–
[KNAH22][3]	–	–	–	226[4]	–	–	–	–
[RRCC26][3]	–	–	–	–	–	–	–	3862[4]

Reported numbers are in thousands of clock cycles. [1] Based on INDCPA timing. [2] Estimated based on the reported clock frequency. [3] Pure hardware implementation. [4] Approximated from ML-KEM-512 (ML-DSA-87) implementations.

Table 7 presents the time performance of our implementation in comparison with related work from the literature. Developed source codes for RISQrypt's software layer is compiled with the RISC-V toolchain using `-O3` option to optimize speed. Reported numbers are in thousands of clock cycles. For our solution, the mean number of clock cycles is reported as the average over 1024 experiments with uniformly random inputs. For the existing works, the reported figures are taken directly from the corresponding publications. For ML-KEM, the performance of the KEM functions are reported. Input message length is set to 32B for ML-DSA.

As can be observed in the table, our design significantly outperforms the literature across all functions of ML-KEM and ML-DSA. For ML-KEM.MaskedDecaps, there is only one prior work suitable for comparison, namely [FVR+22]. This study is the closest to ours in terms of scope and also reports the best performance among existing literature. The authors propose instruction set extensions for Keccak and mask conversions and a loosely coupled accelerator for polynomial arithmetic. However, since their mask conversion engine and Keccak accelerator are implemented as instruction set extensions, their flexibility and overall efficiency are inherently limited. As a result, our implementation outperforms their reported ML-KEM.MaskedDecaps performance by a factor of **11.3**×. A similar performance advantage is observed for the unmasked functions. In particular, for ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.Decaps, our implementation achieves speed-ups of **9.3**×, **10.64**×, and **8.94**×, respectively. On the other hand, compared to [AOP+25], which reports the closest timing results, our implementation still achieves speed-ups of **2.69**×, **2.57**×, and **2.37**× for ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.Decaps, respectively.

For ML-DSA.MaskedSign, there is **no directly comparable work in the literature**. Although [WZZ+24] also considers masking in their implementation, the applied counter-

measure is limited to polynomial arithmetic. Therefore, their reported results for masked functions are not directly comparable to ours. Nevertheless, we compare our performance against unprotected implementations reported in the literature. Compared to [AOP+25], our implementation achieves speedups of **1.95**×, and **1.71**×, for ML-KEM.KeyGen, and ML-DSA.Verify, respectively, while performance of ML-DSA.Sign remains comparable. Note that the acceleration strategy of [AOP+25] differs from ours in that they adopt instruction set extensions on a RISC-V–based vector processor with high area overhead, which operates as a co-processor loosely coupled to the main processor. In the next section, we further show that our design achieves significantly better area efficiency than [AOP+25], in addition to the improved time performance. Another competitive implementation is [MCL+23][4], against which we achieve speedups of **2.19**×, **1.14**×, and **2.23**×, for ML-DSA.KeyGen, ML-DSA.Sign, and ML-DSA.Verify. We note that [MCL+23] focuses exclusively on Dilithium and does not provide a masked implementation. However, despite the narrower scope, their acceleration strategy is conceptually similar to ours, as they also employ loosely coupled accelerators for both NTT and Keccak while leaving a small portion of the computation to software.

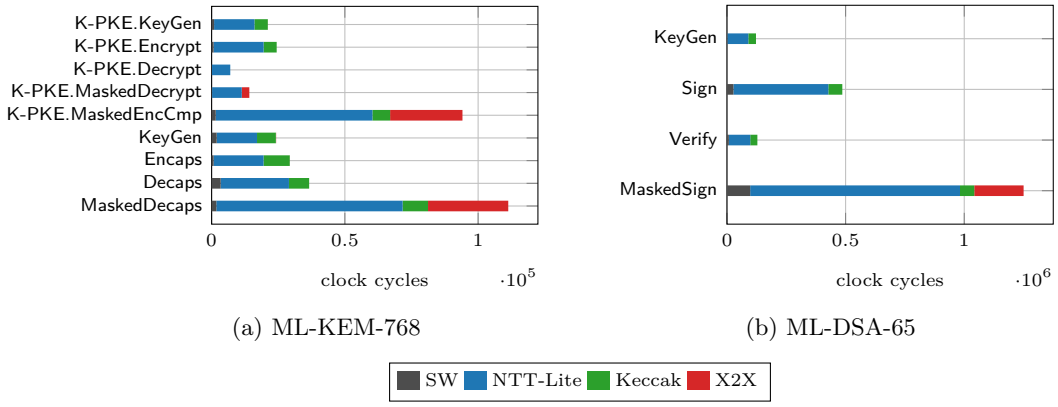
In general, the primary reason for the superiority of our work is that, with the exception of [AOP+25, MCL+23], none of the existing studies employ loosely coupled accelerators for all required primitives. The works [NDMZ+21] and [MBB+23] focus on tightly coupled accelerators and consequently achieve only a limited level of acceleration. Although [WZZ+24] does not rely on tightly coupled acceleration, it does not include a hardware accelerator for Keccak and instead applies software-level optimizations. In [KSFS24], similar to [FVR+22], the authors employ a tightly coupled accelerator for Keccak while using a loosely coupled accelerator for the NTT. As a result, their performance remains below ours. Furthermore, [DMMM23] accelerates only Keccak. Overall, our superior performance demonstrates the advantage of employing loosely coupled accelerators for all primary operations.

We also compare RISQrypt against the pure hardware solutions of [KNAH22, RRCC26], both of which provide complete masking schemes comparable to our work, in contrast to approaches of [WZZ+24, BNKU+26] that protect only selected components, such as polynomial arithmetic. Since [KNAH22] reports results for ML-KEM-512 and [RRCC26] reports results for ML-DSA-87, we normalize their execution times using scaling factors of 1.65 and 0.78, respectively, to estimate the corresponding performance for ML-KEM-768 and ML-DSA-65. These factors are obtained from the ratios of the corresponding software benchmark timings reported in [KRSS19][5], which is a widely used benchmark suite for PQC implementations in the literature. These factors are only approximate for hardware implementations, as hardware scaling across parameter sets may differ from software due to differences in parallelism and resource utilization. As shown in the results, RISQrypt achieves speedups of **2.07**× and **3.13**× over [KNAH22] and [RRCC26], respectively. We omit comparisons of the unprotected implementations with pure hardware solutions, as they are beyond the scope of this work.

In Figure 8, we analyze the time share of each accelerator in the system. Again, mean clock cycle values are reported over 1024 experiments with uniformly random inputs. Observe that NTT-Lite dominates the running time for all functions. This offers an opportunity to further speed-up the design by integrating more computation resources, i.e., multiple BUs, into NTT-Lite, which we leave for future work. It also confirms our design goal of supporting all arithmetic operations within NTT-Lite, in both masked and

[4]The authors of [MCL+23] do not report clock cycle counts directly. Instead, they provide execution times in milliseconds, the operating frequencies of the target platforms (666 MHz for hardware and 150 MHz for software) and the ratio of hardware-accelerated subroutines as 0.94. We estimate a combined effective frequency as $176 \approx 666 \cdot (1 - 0.94) + 150 \cdot 0.94$ and compute the corresponding clock cycles based on the reported timings and this estimated frequency.

[5]https://github.com/mupq/pqm4/blob/master/benchmarks.csv **Commit ID: 90bfb63**



(a) ML-KEM-768 (b) ML-DSA-65

Figure 8: Number of clock cycles spent by accelerators.

unmasked functions. The second most time-consuming accelerator for masked functions is X2X. In contrast, Keccak accounts for only a small fraction of the total execution time in our implementation, unlike the hashing bottlenecks reported in prior software-oriented studies [KRSS19]. The software execution contributes less than 8% of the total cycle count across all evaluated functions, highlighting the effectiveness and functional coverage of the proposed hardware acceleration. Particularly, it takes roughly 2% of the total cycles for ML-KEM.MaskedDecaps and 8% of the total cycles for ML-DSA.MaskedSign. The ratio of mask conversion operations done by X2X in ML-KEM.MaskedDecaps is greater compared to ML-DSA.MaskedSign, amounting to 27% and 16%, respectively. In ML-DSA.MaskedSign, NTT-Lite accounts for 70% of the operations.

To further study the efficiency of our design for masked implementations, we report the time performance of the non-linear algorithms in Table 8 for both ML-KEM and ML-DSA. As in Section 6.1, time performance reported here corresponds to the execution of these algorithms on the entire polynomial in $\mathcal{R}_q$. For MaskedCBD and MaskedUniform, we also report the time cost of pseudo-random sampling performed by Keccak accelerator, denoted by XOF. Observe that all of the utilized non-linear gadgets are realized highly efficiently, spending a few thousand clock cycles. Interestingly, the overhead of XOF to MaskedCBD is very small, around 4%. We note that in XOF + MaskedCBD, the Keccak overhead is limited to the FIFO read operation, as the block permutation is executed in parallel with the MaskedCBD logic. For MaskedUniform used in ML-DSA, the overhead of Keccak is around 30%, since Keccak processes multiple blocks in this case and therefore must wait for the completion of the block permutation until all the blocks are squeezed. Although MaskedSubCompress and MaskedPolyToMsg are based on the same algorithm (Algorithm 8), the first takes more time as it requires 32-bit arithmetic.

Table 8: Time performance of non-linear algorithms in clock cycles.

ML-KEM-768		**ML-DSA-65**	
MaskedCBD	3733	MaskedUniform	3639
XOF + MaskedCBD	3894	XOF + MaskedUniform	4714
MaskedSubCompress	5443	MaskedCheckNormω	6720
MaskedPolyFromMsg	1177	MaskedDecompose	7632
MaskedPolyToMsg	2037		
MaskedZeroTestVec	8765		

Code size is another crucial consideration in embedded systems. In Table 9, we report the code size of our implementation and compare it with existing works from the literature.

Table 9: Code size comparison with prior work.

ML-KEM-768			ML-DSA-65		
Work	**Masking**	**Size (B)**	**Work**	**Masking**	**Size (B)**
RISQrypt	✓	17612	**RISQrypt**	✓	18848
[FVR+22]	✓	28554			
RISQrypt	✗	10236	**RISQrypt**	✗	14324
[FVR+22]	✗	13028	[WZZ+24]	✗	32120*
[WZZ+24]	✗	32520*	[KSFS24]	✗	20052
[AOP+25]	✗	11800	[AOP+25]	✗	21008

* Reported for Kyber-1024 and Dilithium-87; no significant difference is expected with respect to the security level.

Table 10: Hardware utilization of our work compared to designs from the literature. Only the overhead of cryptographic modules are considered.

Work	Platform	Architecture	LUT/FF/DSP/BRAM	Area_{HW}	Scheme	$\text{Area}_{\text{SW/T}}$	ATP
Masked							
RISQrypt	▲	L	25068/19500/15/0.5	39.73	K D	2.42/49.0 2.6/49.16	11.26/13.71/53.4 59.0/604.79/61.95
[FVR+22]	▲	L, T	16979/8834/7/20.5	35.12	K	3.97/45.93	98.3/136.88/567.28
[KNAH22]	◗	HW	152860/NA/54/489.5	445.62	K	0/445.62	-/1007.11[1] /-
[RRCC26]	▮	HW	61629/33861/21/47	111.55	D	0/111.55	-/3862.08[1] /-
Unprotected							
RISQrypt	▲	L	10719/6166/9/0.5	16.86	K D	1.4/25.11 1.97/25.33	5.78/7.04/8.79 30.81/123.24/32.35
[FVR+22]	▲	L, T	7687/3515/7/4.5	14.15	K	1.79/22.78	48.75/67.88/71.3
[WZZ+24]	▼	L	3087/1050/4/3 6639/1660/16/3	6.54 14.11	K D	4.59/17.97 4.57/25.52	96.33/120.59/114.84 532.55/1276.63/506.77
[DMMM23]	▲	L	7128/3798/0/0	9.02	K	NA/18.42	122.16/157.73/199.56
[NDMZ+21]	◆	T	178/0/5/0.5 377/0/10/0.5	2.01 3.75	K D	NA/11.4 NA/14.28	79.26/83.48/98.31 424.71/1457.86/423.14
[MBB+23]	▼	T	459/0/2/0	1.08	D	NA/11.6	55.58/462.77/59.3
[MCL+23]	▼	L	13128/11556/4/14	28.04	D	NA/37.93	100.13/207.10/106.58
[KSFS24]	■	L, T	7219/3238/7/6	14.43	D	2.76/24.03	256.65/781.71/270.82
[AOP+25]	▲	L	48158/16641/112/0	91.08	K D	1.62/99.55 2.89/100.81	61.72/71.67/82.62 235.91/479.89/217.77

Design approaches: (L) Loosely coupled HW/SW co-design, (T) Tightly coupled HW/SW co-design, (HW) pure hardware. (Platforms: (▲) Artix7, (▼) Zynq7000, (◆) ZCU106, (■) Zynq Ultrascale+, (◗) VC707, (▮) Kintex-7. ✣ Limited SCA protection. Schemes: ML-KEM (K), ML-DSA(D). ATP: KeyGen/Enc/(Masked)Dec (K) or KeyGen/(Masked)Sign/Verify (D). $\text{Area}_{\text{SW/T}}$: Area_{SW} / Area_{T}. [1] Approximated from ML-KEM-512 (ML-DSA-87) implementations.

We present the code sizes separately for the masked and unmasked configurations, as different source files are compiled depending on the scenario. For example, in the masked configuration for ML-KEM, the compiled implementation includes ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.MaskedDecaps, whereas the unmasked configuration includes ML-KEM.KeyGen, ML-KEM.Encaps, and ML-KEM.Decaps. The same approach is followed for ML-DSA. The results show that our design is also competitive in terms of code size compared to prior work. The primary reason for this advantage is the reduced software workload, as also evidenced by the profiling results.

6.2.3 Area-Time Comparison

In this section, we perform an area-time evaluation of our architecture and demonstrate that the area overhead is well justified given the achieved performance improvements. Computing the Area–Time Product (ATP) for HW/SW co-designs is non-trivial; therefore, we adopt the following methodology. First, we estimate the hardware area overhead using $\text{Area}_{\text{HW}} = \text{LUT} + \text{FF}/2 + \text{DSP} \cdot 309 + \text{BRAM} \cdot 564$, which expresses all resources in LUT-equivalent units. The coefficients 309 and 564 were obtained by forcing Vivado to synthesize a 16×16-bit multiplier and a 1024×32 memory block using LUTs only, as previously done [PHGCGP22]. The multiplier size matches the core multiplication width of our architecture, while the memory configuration corresponds to the maximum depth of a single BRAM in Artix-7 with 32-bit data width. The estimated LUT-equivalents of DSPs and BRAMs may vary slightly depending on the target platform. Since we consider HW/SW co-designs, we also include the area cost of the processor core for the software stack (e.g., RISC-V). We employ the area of the HORNET core used in RISQrypt, as reported in Table 4, excluding BRAM usage, for all designs in this section. Specifically, $\text{Area}_{\text{Core}} = \text{LUT}_{\text{Core}} + \text{FF}_{\text{Core}}/2 = 6839.5$. We then estimate the area overhead of the software by converting the reported code size into an equivalent number of BRAM blocks as $\text{Area}_{\text{SW}} = \text{BRAM}_{\text{Code}} = \text{Code Size}/4096$. The division by 4096 captures the byte capacity of a single 1024×32 BRAM block. The total area is then computed as $\text{Area}_{\text{T}} = \text{Area}_{\text{HW}} + \text{Area}_{\text{Core}} + \text{Area}_{\text{SW}}$ and the Area–Time Product is defined as $\text{ATP} = (\text{Area} \times \text{Time})$. The only exception is that we set $\text{Area}_{\text{T}} = \text{Area}_{\text{HW}} + \text{Area}_{\text{SW}}$ for pure hardware solutions as they do not need a software core to operate. We exclude the clock frequency from the comparison, as it strongly depends on the target platform. For example, both our design and [FVR+22] use a soft RISC-V core on an Artix-7 FPGA, which results in relatively low operating frequencies; both implementations report clock frequencies of around 60 MHz. On the other hand, platforms such as Zynq-7000, which integrate an ARM processor, can reach up to 150 MHz, as reported in [MCL+23]. The ATP scores reported for studies targeting FPGA platforms different from ours [KNAH22,RRCC26,WZZ+24,NDMZ+21,MBB+23,MCL+23,KSFS24] should be considered estimates, as differences in FPGA resources, such as clock frequency and LUT-equivalents, may affect the reported results. A fully fair comparison would require porting all designs to a common platform and re-evaluating them under identical conditions; however, such an effort would be engineering-intensive and beyond the scope of this work. Moreover, our design can be synthesized at 500 MHz clock frequency in ASIC environment (see Table 6), which is sufficient for the clock frequencies typically used in low-power microcontrollers. We underline that our goal is not to rank prior works against one another, but rather to justify the hardware overhead of our design by demonstrating that it achieves superior ATP.

For the comparison, certain missing data points had to be estimated, as not all components required for the ATP computation are consistently reported in the literature. In particular, the studies [NDMZ+21, MCL+23, MBB+23, DMMM23] do not provide code size information. For [NDMZ+21] and [MBB+23], we used the code size of a ML-DSA implementation compiled with the RISC-V toolchain, as reported in [KSFS24].

For [MCL+23], we computed a code size by taking the average of the reported code sizes of works that support ML-DSA and employ loosely coupled acceleration approach, namely our implementation, [WZZ+24], and [KSFS24]. Similarly for [DMMM23] and [NDMZ+21], we took the average of the ML-KEM code size reported by us, [FVR+22], and [WZZ+24]. These decisions do not affect the conclusions drawn in this section, as our implementation has the smallest code size among the works that report it (see Table 9), and code size is a minor factor overall.

Table 10 presents the computed ATP results of our work compared to the related designs from the literature. The results show that RISQrypt provides superior ATP results compared to the existing work. Although our design is slightly larger in several cases, its high performance compensates for the area overhead, resulting in a favorable overall cost. [FVR+22] utilizes 31% fewer LUTs and 53% fewer FFs compared to our design. However, due to its significantly higher BRAM consumption, the total area estimates (Area_T) of our design are only 1.13× and 1.19× larger for the masked and unprotected configurations, respectively. Recall that the authors of [FVR+22] propose tightly coupled accelerators for Keccak and mask conversion. Despite their slightly lower area estimate, our solution achieves approximately **8.11**× to **10.62**× better ATP results, due to its significantly improved time performance. This further underscores the advantage of our loosely coupled accelerator architecture across all operations. In the unprotected setting, the tightly coupled designs of [MBB+23, NDMZ+21] also stand out due to their very low area overhead. However, despite their smaller area footprint, the performance gains achieved by our approach more than compensate, resulting in superior ATP overall.

Recall that [AOP+25] leverages vector instructions in the loosely coupled Big Number co-processor within the OpenTitan project to perform polynomial arithmetic. The high level of parallelism achieved through these vector instructions comes at a substantial area cost. Accelerator overhead of their design (Area_HW) is **5.4**× larger than ours. This area advantage, combined with the time-performance improvements discussed in the previous section, results in **3.89**× to **10.69**× better ATP values. Later, [NHP26] improves upon [AOP+25], achieving a 17% speedup. Nevertheless, its estimated ATP remains **3.23**× to **8.87**× higher than that of our solution, even when disregarding its additional area overhead, which is present in the reported implementation.

The total area results of [WZZ+24] for Dilithium are very close to ours, while for Kyber their design consumes approximately **0.39**× of Area_HW of our design. However, these results correspond to two separate, scheme-specific architectures, whereas our measurements are based on a unified architecture supporting both ML-KEM and ML-DSA. When their two dedicated designs are deployed together (Area_HW columns aggregated for Kyber and Dilithium) to provide support for both schemes, the total area cost becomes approximately **1.22**× that of ours. Although the total area overhead of [KSFS24] is slightly lower than ours, our design achieves a **6.34**× to **8.37**× improvement in ATP. Area_HW of [MCL+23] is **1.47**× that of our design, while supporting only Dilithium. Moreover, their modular reduction unit is specialized to the specific structure of Dilithium's modulus q, which limits generality. Specifically, our design achieves a **1.68**× to **3.29**× better ATP compared to [MCL+23].

Against pure hardware solutions providing comparable masked implementations [KNAH22, RRCC26], RISQrypt requires only **0.08**× and **0.35**× the Area_HW of these designs, respectively. Combined with its superior execution time (see Table 7), RISQrypt achieves **18.85**× and **6.38**× lower ATP, respectively.

6.3 Implementation Results for SLH-DSA and FN-DSA

In this subsection, we present implementation results for SLH-DSA [Nat24c] and FN-DSA [FHK+20] on the RISQrypt platform. The focus of this work is to demonstrate the versatility and applicability of RISQrypt in accelerating these algorithms, among others.

While a detailed comparison with existing implementations would certainly be of interest, such an evaluation falls outside the scope of the present paper and is therefore left for future work. For FN-DSA, we implement only the signature verification function, as FN-DSA is primarily intended for use cases involving fast verification of offline signatures especially on embedded systems. For both algorithms, we evaluate two configurations: one with hardware acceleration enabled and one with hardware acceleration disabled. The latter uses the unmodified reference implementations of both algorithms[6]. For a better evaluation of the effect of hardware acceleration, we integrated optimized assembly implementations of Keccak targeting RISC-V ISA into the mentioned reference implementations. For unprotected and masked Keccak calls, we used [ZYHK25] and [GD23b], respectively.

The hardware-accelerated SLH-DSA implementation is based on a modified version of the reference implementation, in which all Keccak function calls are redirected to RISQrypt's dedicated Keccak accelerator. The hardware-accelerated FN-DSA implementation leverages both the Keccak and NTT-Lite accelerators. Owing to its programmability, NTT-Lite supports the FN-DSA modulus $q = 12289$. Since NTT-Lite supports sizes of up to $n = 256$ whereas FN-DSA-512 requires $n = 512$, we adopt an incomplete NTT-based implementation. Specifically, NTT-Lite's dual mode is capable of executing two independent $n = 256$ NTTs in parallel, which corresponds to the incomplete NTT decomposition for an $n = 512$ transform. Our evaluation considers the compressed signature verification procedure of FN-DSA, in which signatures employ a compact encoding that stores polynomial coefficients using fewer bits. Consequently, signature verification requires an initial decompression step. As this functionality is not natively supported by NTT-Lite, we evaluate two implementation variants. In the first, decompression is performed entirely in software. In the second, decompression is integrated as an additional execution mode of NTT-Lite's EU. The additional hardware required for this mode incurs an area overhead of 119 LUTs and 13 FFs under our Vivado synthesis flow. The corresponding results are presented in Table 11. For the unprotected SLH-DSA operations, RISQrypt achieves speedups ranging from 54.91× to 57.10×, while reducing the code size by 0.16× that of the software-only implementation. More significantly, for masked signature generation, RISQrypt provides a 495.59× together with a code-size reduction by 0.38×. The larger speed-up observed for the masked case is expected, as software implementations incur substantially higher overhead when incorporating masking countermeasures. For FN-DSA, RISQrypt achieves speedups of 14.78×–34.19× for compressed signature verification, depending on the decompression implementation, and 0.78×–0.79× reduction in code-size.

Table 11: Performance of RISQrypt's SLH-DSA and FN-DSA implementations.

		SLH-DSA-SHAKE-128f				**FN-DSA-512**
	HW Acc.	KeyGen	Sign	Verify	MaskedSign	Verify
Time	✗	74702	1748429	102624	16745751	547
	✓	1338	31838	1797	33789	37 / 16[2]
	Speed-up	**55.83×**	**54.91×**	**57.10×**	**495.59×**	**14.78× / 34.19×**
Code-Size (B)	✗		13712		30164[1]	23356
	✓		11596		18768[1]	5112 / 4916[2]
	Reduction		**0.16×**		**0.38×**	**0.78× / 0.79×**

Reported numbers for time performance are in thousands of clock cycles. [1] Includes compiled code for KeyGen+MaskedSign+Verify functions. [2] Additional functionality for FN-DSA's decompression is added to EU.

[6] https://github.com/sphincs/sphincsplus **commit ID: 7ec789a** and https://falcon-sign.info/impl/falcon.c.html

7 Conclusion

We presented RISQrypt, a HW/SW co-design framework for secure, fast, and agile implementations of PQC. Our architecture integrates three dedicated hardware accelerators: NTT-Lite for polynomial arithmetic, Keccak, and X2X for A2B & B2A mask conversions. NTT-Lite is an efficient design, which is run-time configurable and optimized for both half- and full-word lengths needed by ML-KEM and ML-DSA. We showed that all subroutines in ML-KEM and ML-DSA are supported by the functionality of integrated hardware accelerators. Most importantly, we presented efficient implementations for both ML-KEM's masked decapsulation and ML-DSA's masked signature generation, and demonstrated that they are side-channel secure both theoretically and in practice. We also discussed that the required building blocks for the masked implementations are also fully supported by X2X and NTT-Lite, avoiding data processing in the software core, which could otherwise introduce microarchitectural leakage. We showed that having a dedicated accelerator for each major computational primitive is essential to achieving high performance. As a result, our implementation achieves an order-of-magnitude speedup over the state of the art while also delivering a superior area–time product performance.

An additional advantage of employing a programmable polynomial arithmetic accelerator is the resulting agility of the design. We have demonstrated that the proposed hardware platform can be used to efficiently implement additional PQC algorithms built on similar computational primitives by implementing FN-DSA and SLH-DSA. Moreover, it enables efficient and side-channel-secure implementations of client-side functions in Fully Homomorphic Encryption (FHE) schemes, including encryption and decryption for BFV [Bra12, FV12] and CKKS [CKKS17], as well as the transciphering algorithm PASTA [DGH+23]. We leave the detailed implementation and evaluation of these additional algorithms for future work.

8 Acknowledgments

This work is partially supported by the European Union's Horizon Europe research and innovation program under grant agreement No: 101079319, TUBITAK under Grant Number 122E222, the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and the CyberSecurity Research Flanders with reference number VOEWICS02.

References

[AB75] R.C. Agarwal and C.S. Burrus. Number theoretic transforms to implement fast digital convolution. *Proceedings of the IEEE*, 63(4):550–560, 1975.

[ABC+23] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting Dilithium against leakage: Revisited sensitivity analysis and improved implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4):58–79, 2023.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J Kannwischer, and Amber Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In *International Conference on Applied Cryptography and Network Security*, pages 853–871. Springer, 2022.

[Ajt96] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of*

Computing, STOC '96, page 99–108, New York, NY, USA, 1996. Association for Computing Machinery.

[AMI+23] Aikata Aikata, Ahmet Can Mert, Malik Imran, Samuel Pagliarini, and Sujoy Sinha Roy. KaLi: A crystal for post-quantum security using Kyber and Dilithium. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70(2):747–758, 2023.

[AOP+25] Amin Abdulrahman, Felix Oberhansl, Hoang Nguyen Hien Pham, Jade Philipoom, Peter Schwabe, Tobias Stelzer, and Andreas Zankl. Towards ML-KEM & ML-DSA on OpenTitan. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2025.

[BAA23] Luke Beckwith, Abubakr Abdulgadir, and Reza Azarderakhsh. A flexible shared hardware accelerator for NIST-recommended algorithms CRYSTALS-Kyber and CRYSTALS-Dilithium with SCA protection. In *Cryptographers' Track at the RSA Conference*, pages 469–490. Springer, 2023.

[Bar87] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 311–323. Springer, Berlin, Heidelberg, August 1987.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 116–129, New York, NY, USA, 2016. Association for Computing Machinery.

[BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit: with application to lattice-based KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):553–588, Aug. 2022.

[BDH+21] Shivam Bhasin, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. Attacking and defending masked polynomial comparison. *IACR TCHES*, 2021(3):334–359, 2021.

[BDK+17] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS – Kyber: a CCA-secure module-lattice-based KEM. Cryptology ePrint Archive, Report 2017/634, 2017. https://eprint.iacr.org/2017/634.

[BDK+21] Michiel Van Beirendonck, Jan-Pieter D'anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. A side-channel-resistant implementation of SABER. *J. Emerg. Technol. Comput. Syst.*, 17(2), apr 2021.

[BDK+22] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber algorithm specifications and supporting documentation. Technical report, National Institute of Standards and Technology, 2022. Round 3 Submission.

[BDPA12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak SHA-3 Submission. In *Proceedings of the Round 3 SHA-3 Workshop*, 2012. Retrieved: 2025-12-07.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *International Conference on Smart Card Research and Advanced Applications*, pages 64–81. Springer, 2014.

[BGR+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. Masking Kyber: First- and higher-order implementations. *IACR TCHES*, 2021(4):173–214, 2021.

[BNG21] Luke Beckwith, Duc Tri Nguyen, and Kris Gaj. High-performance hardware implementation of CRYSTALS-Dilithium. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10. IEEE, 2021.

[BNKU+26] Mojtaba Bisheh-Niasar, Emre Karabulut, Kiran Upadhyayula, Michael Norris, and Bharat Pillilli. Adams bridge accelerator: Bridging the post-quantum transition. *Cryptology ePrint Archive*, 2026.

[BPO+20] Florian Bache, Clara Paglialonga, Tobias Oder, Tobias Schneider, and Tim Güneysu. High-speed masking for polynomial comparison in lattice-based kems. *IACR TCHES*, 2020(3):483–507, 2020.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Berlin, Heidelberg, August 2012.

[CGL+24] Jean-Sébastien Coron, François Gérard, Tancrède Lepoint, Matthias Trannoy, and Rina Zeitoun. Improved high-order masked generation of masking vector and rejection sampling in Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):335–354, Sep. 2024.

[CGMZ23] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. High-order polynomial comparison and masking lattice-based encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 153–192, 2023.

[CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. Improved gadgets for the high-order masking of Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(4), 2023.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Cham, December 2017.

[CKRG+24] Xavier Carril, Charalampos Kardaris, Jordi Ribes-GonzáLez, Oriol Farràs, Carles Hernandez, Vatistas Kostalabros, Joel Ulises González-Jiménez, and Miquel Moreto. Hardware acceleration for high-volume operations of CRYSTALS-Kyber and CRYSTALS-Dilithium. *ACM Transactions on Reconfigurable Technology and Systems*, 17(3):1–26, 2024.

[CMM+24] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking – unrolled trivium to the rescue. *IACR Communications in Cryptology*, 1(2), 2024.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[DGH+23] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Pasta: A case for hybrid homomorphic encryption. *IACR TCHES*, 2023(3):30–73, 2023.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, Feb. 2018.

[DKL+22] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium algorithm specifications and supporting documentation. Technical report, National Institute of Standards and Technology, 2022. NIST PQC Selected Algorithm Submission, Version 3.02.

[DMMM23] Alessandra Dolmeta, Mattia Mirigaldi, Maurizio Martina, and Guido Masera. Implementation and integration of Keccak accelerator on RISC-V for CRYSTALS-Kyber. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, CF '23, page 381–382, New York, NY, USA, 2023. Association for Computing Machinery.

[DVBV22] Jan-Pieter D'Anvers, Michiel Van Beirendonck, and Ingrid Verbauwhede. Revisiting higher-order masked comparison for lattice-based cryptography: Algorithms and bit-sliced implementations. *IEEE Transactions on Computers*, 72(2):321–332, 2022.

[DZD+17] A Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In *International Conference on Smart Card Research and Advanced Applications*, pages 105–122. Springer, 2017.

[FHK+20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru, Oct 2020.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive, Report 2012/144, 2012.

[FVR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR TCHES*, 2022(1):414–460, 2022.

[GD23a] John Gaspoz and Siemen Dhooghe. Threshold implementations in software: Micro-architectural leakages in algorithms. *IACR TCHES*, 2023(2):155–179, 2023.

[GD23b] John Gaspoz and Siemen Dhooghe. Threshold implementations in software: Micro-architectural leakages in algorithms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(2):155–179, 2023.

[GGJR11] Benjamin Jun Gilbert Goodwill, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.

[GLK21] Wenbo Guo, Shuguo Li, and Liang Kong. An efficient implementation of Kyber. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(3):1562–1566, 2021.

[GS66] W. M. Gentleman and G. Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the 1966 Fall Joint Computer Conference*, 1966.

[GSH+26] Anna Guinet, Jan Schoone, Niklas Höher, Dina Hesse, and Tim Güneysu. LFSRs and Boolean Masking: An In-depth Security Analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(3):426–464, Jul. 2026.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of KECCAK, 2017.

[HHLW20] Yiming Huang, Miaoqing Huang, Zhongkui Lei, and Jiaxuan Wu. A pure hardware implementation of CRYSTALS-Kyber PQC algorithm through resource reuse. *IEICE Electronics Express*, 17(17):20200234–20200234, 2020.

[HKL+22] Daniel Heinz, Matthias J Kannwischer, Georg Land, Thomas Pöppelmann, Peter Schwabe, and Daan Sprenkels. First-order masked Kyber on ARM Cortex-M4. *Cryptology ePrint Archive*, 2022.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[JGCS24] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. A configurable CRYSTALS-Kyber hardware implementation with side-channel protection. *ACM transactions on embedded computing systems*, 23(2):1–25, 2024.

[Kar63] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[KDVB+22] Suparna Kundu, Jan-Pieter D'Anvers, Michiel Van Beirendonck, Angshuman Karmakar, and Ingrid Verbauwhede. Higher-order masked Saber. In *International Conference on Security and Cryptography for Networks*, pages 93–116. Springer, 2022.

[KNAH22] Tendayi Kamucheka, Alexander Nelson, David Andrews, and Miaoqing Huang. A masked pure-hardware implementation of Kyber cryptographic algorithm. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–1. IEEE, 2022.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.

[KRSS19] Matthias J Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4. 2019.

[KSFS24] Patrick Karl, Jonas Schupp, Tim Fritzmann, and Georg Sigl. Post-quantum signatures on RISC-V with hardware acceleration. *ACM Transactions on Embedded Computing Systems*, 23(2):1–23, 2024.

[MBB+23] Konstantina Miteloudi, Joppe W Bos, Olivier Bronchain, Björn Fay, and Joost Renes. Pq. v. alu. e: Post-quantum RISC-v custom alu extensions on Dilithium and Kyber. In *International Conference on Smart Card Research and Advanced Applications*, pages 190–209. Springer, 2023.

[MCL+23] Gaoyu Mao, Donglong Chen, Guangyan Li, Wangchen Dai, Abdurrashid Ibrahim Sanka, Çetin Kaya Koç, and Ray C. C. Cheung. High-performance and configurable sw/hw co-design of post-quantum signature CRYSTALS-Dilithium. *ACM Trans. Reconfigurable Technol. Syst.*, 16(3), June 2023.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking Dilithium: Efficient implementation and side-channel evaluation. In *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*, pages 344–362. Springer, 2019.

[Nat15] National Institute of Standards and Technology (NIST). SHA-3 standard: Permutation-based hash and extendable-output functions. 2015.

[Nat24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard. Federal Information Processing Standards Publication (FIPS) NIST FIPS 204, Department of Commerce, Washington, DC, 2024.

[Nat24b] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Technical report, U.S. Department of Commerce, Washington, D.C., 2024.

[Nat24c] National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Standard. Federal Information Processing Standards Publication (FIPS) NIST FIPS 205, Department of Commerce, Washington, DC, 2024.

[NDKV25] Quinten Norga, Jan-Pieter D'Anvers, Suparna Kundu, and Ingrid Verbauwhede. X2X: Low-randomness and high-throughput A2B and B2A conversions for d+1 shares in hardware. In *Constructive Approaches for Security Analysis and Design of Embedded Systems: First International Conference, CASCADE 2025, Saint-Etienne, France, April 2–4, 2025, Proceedings*, page 119–158, Berlin, Heidelberg, 2025. Springer-Verlag.

[NDMZ+21] Pietro Nannipieri, Stefano Di Matteo, Luca Zulberti, Francesco Albicocchi, Sergio Saponara, and Luca Fanucci. A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms. *IEEE Access*, 9:150798–150808, 2021.

[NHP26] Ruben Niederhagen and Hoang Nguyen Hien Pham. Improving ml-kem and ml-dsa on opentitan: Efficient multiplication vector instructions for otbn. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):495–519, Apr. 2026.

[OSPG18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. Practical CCA2-secure and masked ring-LWE implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):142–174, Feb. 2018.

[ÖY23] Sıla Özeren and Oğuz Yayla. Methods for masking CRYSTALS-Kyber against side-channel attacks. In *2023 16th International Conference on Information Security and Cryptology (ISCTürkiye)*, pages 1–6. IEEE, 2023.

[PHGCGP22] Francisco Pajuelo-Holguera, José M. Granado-Criado, and Juan A. Gómez-Pulido. Fast montgomery modular multiplier using FPGAs. *IEEE Embedded Systems Letters*, 14(1):19–22, 2022.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM STOC*, pages 84–93, Baltimore, MD, USA, 2005.

[RRCC26] Kamal Raj, Prasanna Ravi, Tee Kiah Chia, and Anupam Chattopadhyay. Improved ml-dsa hardware implementation with first order masking countermeasure. *Journal of Hardware and Systems Security*, 10(1):9, 2026.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[Sch90] C. P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[SPOG19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In *Public-Key Cryptography – PKC 2019: 22nd IACR International Conference on Practice and Theory of Public-Key Cryptography, Beijing, China, April 14-17, 2019, Proceedings, Part II*, page 534–564, Berlin, Heidelberg, 2019. Springer-Verlag.

[TG16] Michael Tunstall and Gilbert Goodwill. Applying TVLA to public key cryptographic algorithms. *Cryptology ePrint Archive*, 2016.

[TMS24] Tolun Tosun, Amir Moradi, and Erkay Savas. Exploiting the central reduction in lattice-based cryptography. *IEEE Access*, 12:166814–166833, 2024.

[WWL+24] Nuo Wang, Liji Wu, Lei Li, Munkhbaatar Chinbat, and Xiangmin Zhang. Design and implementation of a NTT accelerator based on RISC-V for PQC Kyber. In *2024 IEEE 18th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, pages 24–27, 2024.

[WZCG22] Tengfei Wang, Chi Zhang, Pei Cao, and Dawu Gu. Efficient implementation of Dilithium signature scheme on FPGA SoC platform. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(9):1158–1171, 2022.

[WZZ+24] Tengfei Wang, Chi Zhang, Xiaolin Zhang, Dawu Gu, and Pei Cao. Optimized hardware-software co-design for Kyber and Dilithium on RISC-V SoC FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):99–135, 2024.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of CCA-secure key exchange mechanism CRYSTALS-Kyber on FPGA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 328–356, 2021.

[YTO21] Yasin Yilmaz, Yavuz Selim Tozlu, and Berna Örs. Design and implementation of a 32-bit RISC-V core. page 460 – 464, 2021.

[ZSS+21] Sara Zarei, Aein Rezaei Shahmirzadi, Hadi Soleimany, Raziyeh Salarifard, and Amir Moradi. Low-latency Keccak at any arbitrary order. *IACR TCHES*, 2021(4):388–411, 2021.

[ZYHK25] Jipeng Zhang, Yuxing Yan, Junhao Huang, and Çetin Kaya Koç. Optimized software implementation of keccak, kyber, and dilithium on rv {32, 64} im {B}{V}. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):632–655, 2025.

[ZZW+22] Cankun Zhao, Neng Zhang, Hanning Wang, Bohan Yang, Wenping Zhu, Zhengdong Li, Min Zhu, Shouyi Yin, Shaojun Wei, and Leibo Liu. A compact and high-performance hardware architecture for CRYSTALS-Dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 270–295, 2022.