A Flexible Shared Hardware Accelerator for NIST-Recommended Algorithms CRYSTALS-Kyber and CRYSTALS-Dilithium with SCA Protection

Luke Beckwith , Abubakr Abdulgadir , and Reza Azarderakhsh

PQSecure Technologies

September 16, 2022

Abstract

NIST recently decided upon a set of cryptographic algorithms for future standardization. These algorithms are built upon hard problems which are believed to be resistant to quantum cryptanalysis, unlike RSA and ECC which are trivially broken by a quantum computer running Shor's algorithm. Cryptographic operations are computationally intense, and therefore are often offloaded to dedicated hardware in order to improve performance and reduce energy usage. However, different applications have different needs for performance and cost trade-offs, so it is beneficial to have a variety of performance options for hardware acceleration. In this work we present a flexible hardware architecture for selected algorithms, Kyber and Dilithium. Our architecture includes separate instances optimized for either Kyber and Dilithium as well as a combined architecture which support both algorithms in one design. Further, the design can be instantiated at three levels of performance: lightweight, mid-range, and high performance. We also present a masked implementation for the Kyber-only implementation which protects against first order differential power analysis attacks and timing attacks. The masked implementation requires 2.5× more LUTs and 6.5× more clock cycles for decapsulation.

1 Introduction

The current public key encryption algorithms, RSA and ECC, are built upon the difficulty of integer factorization and the elliptic curve discrete logarithm problem. These problems are difficult to solve on classical computers, and thus were able to be used for secure encryption. However, it is known that if a large quantum computer is created these problems become trivial to solve by applying Shor's algorithm [1]. Due to this upcoming threat, NIST is in the process of standardizing new public key encryption schemes which are built on hard problems that are not known to be vulnerable to quantum attacks. In particular, NIST is working to standardize Key Encapsulation Mechanisms (KEM), which are used to establish shared secret keys for symmetric encryption, and digital signatures, which are used to verify the authenticity and integrity of messages. The families of algorithms that were submitted to the competition were lattice-based, code-based, hashed-based, multivariate, and super isogeny-based. Multivariate and isogeny-based algorithms have recently received serious classical attacks and are not under serious consideration in their current form [2],[3]. Lattice-based algorithms have the best combination of performance and key sizes. Code-based and symmetric-based have lower performance and larger keys but also have the longest history of cryptanalysis, making them a more conservative option.

NIST recently selected the first set of algorithms that will be standardized: the lattice based KEM CRYSTALS-Kyber, the two lattice based digital signature schemes Dilithium and Falcon, and the symmetric-based digital signature scheme SPHINCS$^+$ [4]. Kyber and Dilithium are the recommended algorithms, as they have high performance for all operations, small keys, and high confidence in their security. Falcon was also standardized as it has smaller signature and public keys sizes than Dilithium making it more suitable for some applications. However, Falcon key and signature generation are much slower than Dilithium. SPHINCS$^+$ has slow key and signature generation as well as large signatures,

but was standardized to provide diversity to the portfolio of algorithms standardized. Additionally, it has very small public keys which makes it well suited for some niche applications.

The NSA also recently released the Commercial National Security Algorithm Suite (CNSA) 2.0, which provides recommendations for which cryptographic algorithms and parameters should be used for specific applications [5]. CRYSTALS-Kyber and CRYSTALS-Dilithium configured for security level 5 were the recommend public key algorithms.

1.1 Previous Works

There have been several works related to Kyber and Dilithium in hardware. Several designs of note for Kyber are the compact design presented by Xing and Li [6] and the high performance design presented by Dang et al [7]. These works focus exclusively on optimized designs for Kyber. The work by Xing and Li was the first to create a highly optimized and interlaced scheduling of sampling and polynomial multiplication which allow impressive performance with a low number of BRAMs and LUTs. The work by Dang et al also made efforts to optimize the scheduling of operations with a focus on performance. The polynomial multiplication unit was duplicated to match the length of vectors for the specific security level which allowed very high parallelization of operations at the cost of some flexibility.

For Dilithium, the recent designs of note are the mid-range implementation by Land et al [8], the high performance implementation by Beckwith et al [9], and the compact and high performance implementation by Zhao et al [10]. The work by Land et al focused on making a compact design with reasonable area and performance. The design used a single polynomial multiplier with two butterfly units to perform the Number Theoretic Transform (NTT). The design can be instantiated to support either individual operations at a specified security level, or all operations at a single security level. The high performance design by Beckwith et al utilized several instances of the polynomial multiplier and hash functions to improve performance as well as a optimized scheduling to improve performance. That work was improved upon by Zhao et al which achieved slightly improved performance for key generation and verification in more compact design which required fewer resources.

Another interesting design is the implementation combining Saber and Dilithium into a single core by Akaita et al [11]. Saber is not designed to efficiently support polynomial multiplication using the NTT, however the authors determined that due to the small modulus of Saber the NTT parameters used by Dilithium could be applied to Saber with only a small increase in the chance of decryption failure. This allowed both designs to share a single polynomial multiplier. Other auxiliary units, such as the hash function, are also able to be shared between the algorithms. However, many units such as those used for sampling and encoding polynomial were not able to be shared between Saber and Dilithium. This work was expanded on by the same author in another work that combined Kyber and Dilithium into one hardware architecture [12]. This was the first public work to implement a combined core for both algorithms in hardware.

1.2 Contribution

In this work, we present a flexible architecture for Kyber and Dilithium. The discussed architecture can be configured for lightweight, mid-range, and high performance and can be instantiated as independent modules or as a combined architecture for both algorithms. All security levels are supported and are selected at runtime making it compatible with the NSA's recent guidelines [5]. The lightweight configuration achieves the lowest area for both the individual algorithms and the combined architecture. The mid-range designs provide an excellent trade-off for lightweight devices that still require low latency key exchanges and digital signatures, and our high performance design achieves competitive performance while requiring lower resources and providing more flexibility than other high performance designs. We also present results for a masked implementation of Kyber, providing a first look into the cost of protecting these new standards against power analysis attacks.

2 Background

2.1 Notation

The notation will follow that of the specifications of Kyber [13] and Dilithium [14]. To summarize: the polynomial ring $\mathbb{Z}[X]/(X^n + 1)$ is denoted by R_q. Matrices of polynomials are bold and upper-case,

for example $\boldsymbol{A} \in R_q^{n \times k}$. Vectors in R_q are lower case and bold, polynomials in R_q are lower case. The hat symbol is used to show that an element is in the NTT domain, for example $\hat{e} = NTT(e)$.

2.2 CRYSTALS-Kyber

CRYSTALS-Kyber is a lattice-based cryptosystem built on the difficulty of the Module Learning with Errors (MLWE) problem [13]. The MLWE problem can be summarized as follows: choose a random matrix $\boldsymbol{A} \in R_q^{n \times k}$, a random small vector $\boldsymbol{s} \in R_q^k$, and a random small error $\boldsymbol{e} \in R_q^n$, and define $\boldsymbol{b} = \boldsymbol{A} \times \boldsymbol{s} + \boldsymbol{e}$. Then there are two versions of the LWE problem: the search version where the challenge is to recover $\mathbf{s}$ from the pair ($\mathbf{A}$,$\mathbf{b}$), and the decision version where the challenge is to distinguish between ($\mathbf{A}$,$\mathbf{b}$) and a uniform sample. This is used as follows to create the Chosen Plaintext Attack (CPA) secure PKE Kyber, which can then be converted to a Chosen Ciphertext Attack (CCA) secure KEM.

Key Generation: Key generation is used to generate a public and private key pair from a random 32-byte seed. All polynomials are in R_q. The polynomials in the public matrix $\mathbf{A}$ are sampled uniformly using rejection sampling, the polynomials of the secret vector $\mathbf{s}$ are sampled from the centered binomial distribution and thus have small coefficients. The public key is the pair ($\mathbf{A}$,$\mathbf{t}$=$\mathbf{A}$×$\mathbf{s}$+$\mathbf{e}$), however to reduce the transmission bandwidth only the seed used to generate $\mathbf{A}$ is transmitted and the polynomial $\mathbf{t}$ is encoded into an array of bytes. The matrix $\mathbf{A}$ is also assumed to be sampled in its NTT-domain form. The secret key is the polynomial vector $\mathbf{s}$ which was used to generate the public key. The polynomials of the keys are stored in the NTT domain to reduce the computation time of encapsulation and decapsulation. The pseudocode for key generation is shown in Alg. 1. The relation to the MLWE problem is clear in that the secret value $\mathbf{s}$ cannot be recovered from the public pair ($\mathbf{A}$,$\mathbf{t}$).

Algorithm 1 Kyber CPA Key Generation

Input: Random $d \in \{0,1\}^{256}$
$(\rho, \sigma) \leftarrow$SHA3-512$(d)$
$\hat{\boldsymbol{A}} \in R_q^{k \times k} \leftarrow$RejectionSampler$(\rho)$
$\boldsymbol{s} \in R_q^k \leftarrow$CBDSampler$_{\eta_1}(\sigma, 0)$
$\boldsymbol{e} \in R_q^k \leftarrow$CBDSampler$_{\eta_1}(\sigma, k)$
$\hat{\boldsymbol{s}} \leftarrow$NTT$(\boldsymbol{s})$
$\hat{\boldsymbol{e}} \leftarrow$NTT$(\boldsymbol{e})$
$\hat{\boldsymbol{t}} \leftarrow \hat{\boldsymbol{A}} \circ \hat{\boldsymbol{s}} + \hat{\boldsymbol{e}}$
return (pk=(ρ,Encode$_{12}(\hat{\boldsymbol{t}})$), sk=Encode$_{12}(\hat{\boldsymbol{s}})$)

Algorithm 2 Kyber CPA Encryption

Input: $pk = (\rho, t_{enc})$, message $m \in \{0,1\}^{256}$, random $r \in \{0,1\}^{256}$
$\hat{\boldsymbol{t}} \leftarrow$Decode$_{12}(t_{enc})$
$\hat{\boldsymbol{A}} \in R_q^{k \times k} \leftarrow$RejectionSampler$(\rho)$
$\boldsymbol{r} \in R_q^k \leftarrow$CBDSampler$_{\eta_1}(r, 0)$
$\boldsymbol{e_1} \in R_q^k \leftarrow$CBDSampler$_{\eta_2}(r, k)$
$e_2 \in R_q \leftarrow$CBDSampler$_{\eta_2}(r, 2k)$
$\hat{\boldsymbol{r}} \leftarrow$NTT$(\boldsymbol{r})$
$\boldsymbol{u} \leftarrow$NTT$^{-1}(\hat{\boldsymbol{A}}^T \circ \hat{\boldsymbol{r}}) + \boldsymbol{e_1}$
$v \leftarrow$NTT$^{-1}(\hat{\boldsymbol{t}}^T \circ \hat{\boldsymbol{r}}) + e_2 +$Decompress$_q($Decode$_1(m), 1)$
return $c =$ (Encode$_{d_u}($Compress$_q(u, d_u))$, Encode$_{d_v}($Compress$_q(v, d_v))$)

Encryption: Encryption generates the ciphertext of the message using the public key and a random seed r. The public matrix is regenerated from the seed ρ, multiplied with a random small vector $\boldsymbol{r}$, and masked with a random small error polynomial vector $\boldsymbol{e_1}$. This results forms $\boldsymbol{u}$, the first component of the signature. The vector $\boldsymbol{r}$ is then multiplied by the other component of the public key $\boldsymbol{t}$. The result is masked with a small error vector $\boldsymbol{e_2}$, and a polynomial generated from the message is added to it. The message is converted to a polynomial by mapping 0 bits of the 32-byte message as coefficients with the value 0, and the 1 bits as coefficients with value $\frac{q-1}{2}$. The resulting polynomial v

Algorithm 3 Kyber CPA Decryption

Input: $sk = (\hat{\boldsymbol{s}})$, ciphertext $c = (c_1, c_2)$
$\boldsymbol{u} \leftarrow \text{Decompress}_q(\text{Decode}_{d_u}(c_1), d_u)$
$v \leftarrow \text{Decompress}_q(\text{Decode}_{d_v}(c_2), d_v)$
$\hat{\boldsymbol{s}} \leftarrow \text{Decode}_{12}(sk)$
$m \in \{0,1\}^{256} \leftarrow \text{Encode}_1(\text{Compress}_q(v - \text{NTT}^{-1}(\hat{\boldsymbol{s}}^T \circ \text{NTT}(\boldsymbol{u})), 1))$
return m

Table 1: Parameters for round 3 Kyber submission

	Sec. Level	n	k	q	(η_1, η_2)	(d_u, d_v)	pk (B)	sk (B)	ct (B)
Kyber512	1	256	2	3329	(3,2)	(10,4)	800	1632	768
Kyber786	3	256	3	3329	(2,2)	(10,4)	1184	2400	1088
Kyber1024	5	256	4	3329	(2,2)	(11,5)	1568	3168	1568

is the second component of the ciphertext. Both u and v are compressed and encoded into an array of bytes. The pseudocode is shown in Alg. 2.

Decryption: Decryption first decodes the encoded secret key and ciphertext polynomials. These polynomials are used to calculate $v - \boldsymbol{s}^T \times \boldsymbol{u}$, which is compressed and encoded into an array of 32-bytes. If the secret key and ciphertext are valid, this will recover the message. This succeeds because:

$$\begin{aligned} & v - \boldsymbol{s}^T \times \boldsymbol{u} \\ &= (\boldsymbol{t}^T \times \boldsymbol{r} + e_2 + m') - (\boldsymbol{s}^T \times (\boldsymbol{A}^T \times \boldsymbol{r} + e_1)) \\ &= ((\boldsymbol{A} \times \boldsymbol{s} + \boldsymbol{e})^T \times \boldsymbol{r} + e_2 + m') - (\boldsymbol{s}^T \times (\boldsymbol{A}^T \times \boldsymbol{r} + e_1)) \\ &= (\boldsymbol{A} \times \boldsymbol{s})^T \times \boldsymbol{r} + \boldsymbol{e}^T \times \boldsymbol{r} + e_2 + m' - \boldsymbol{s}^T \times \boldsymbol{A}^T \times \boldsymbol{r} - \boldsymbol{s}^T \times e_1 \\ &= m' + (\boldsymbol{e}^T \times \boldsymbol{r} - \boldsymbol{s}^T \times e_1 + e_2) \end{aligned}$$

Since r, e, e_1, e_2 are all small vectors, the result will be very close to m' and thus when m' is compressed and encoded the original message will be recovered. The pseudocode is shown in Alg. 3.

Fujisaki-Okamoto Transform: This CPA secure encryption scheme is transformed into a CCA secure KEM using the Fujisaki-Okamoto (FO) transformation [15]. The FO transform involves re-encrypting the message during decapsulation and comparing it to the received ciphertext. If the ciphertexts do not match, then a random value is returned in place of the shared secret. This prevents attackers from gaining information about the secret key using malicious ciphertexts. The only change to key generation is in the secret key. Rather than including only the encoded polynomial $\mathbf{s}$, a random seed z is included as well as the public key and its hash. In encapsulation, the random coin r is generated psuedorandomly from the hash of the public key hash and the message, and the shared secret is generated based upon this hash and the hash of the ciphertext. This is also done in decapsulation if the ciphertext is valid.

The parameters for the third round submission of Kyber are shown in Table 1. The modulus and size of polynomials is the same for all security levels. The size of vectors and matrices, parameters for CBD sampling, and parameters for ciphertext compression vary depending on the security level.

2.3 CRYSTALS-Dilithium

Like Kyber, Dilithium is a lattice-based algorithm. Its security is built upon the MLWE and the Module Short Integer Solution (MSIS) problems. Like many other signature schemes, it is constructed using a version of the Fiat-Shamir transformation which can convert an interactive identification protocol into a non-interactive signature. To begin, a simplified version of Dilithium will be discussed. Similarly Kyber, key generation begins with a calculation in the form of the MLWE problem where the public matrix and the calculated vector $\boldsymbol{t} = \boldsymbol{A} \times \boldsymbol{s_1} + \boldsymbol{s_2}$ form the public key and $\boldsymbol{s_1}, \boldsymbol{s_2}$ are kept secret. During signing a random vector $\boldsymbol{y}$ is generated to mask the calculation $c \times \boldsymbol{s_1}$, where c is a short polynomial calculated based on the hash of the upper bits of the multiplication $\boldsymbol{A}$ and $\boldsymbol{y}$ and the message. Since c is also a part of the signature, only the addition of $\boldsymbol{y}$ hides the secret $\boldsymbol{s_1}$. To prevent the signature from leaking information about the secret, the signature is verified to have coefficients within certain bounds that properly hide the secret polynomial, if not, the signature is rejected, and a new attempt

is made with a different $\boldsymbol{y}$ vector. To verify the signature, the verifier calculates $\boldsymbol{A} \times \boldsymbol{z} - c \times \boldsymbol{t}$ and hashes the upper bits of the result with the message. If the result matches c, then the signature is accepted.

Algorithm 4 Dilithium Key Generation

Input: Random $\zeta \in \{0,1\}^{256}$
$(\rho, \rho', K) \leftarrow$SHAKE256$(\zeta)$
$\hat{\boldsymbol{A}} \in R_q^{k \times l} \leftarrow$ExpandA$(\rho)$
$\boldsymbol{s_1} \in S_\eta^l \leftarrow$ExpandS$(\sigma, 0)$
$\boldsymbol{s_2} \in S_\eta^k \leftarrow$ExpandS$(\sigma, l)$
$\hat{\boldsymbol{s_1}} \leftarrow$NTT$(\boldsymbol{s_1})$
$\hat{\boldsymbol{s_2}} \leftarrow$NTT$(\boldsymbol{s_2})$
$\boldsymbol{t} \leftarrow$NTT$^{-1}(\hat{\boldsymbol{A}} \circ \hat{\boldsymbol{s_1}} + \hat{\boldsymbol{s_2}})$
$(\boldsymbol{t_1}, \boldsymbol{t_0}) \leftarrow$Power2Round$_q(\boldsymbol{t}, d)$
$tr \leftarrow$SHAKE256$(\rho || t_1)$
return (pk=(ρ, Encode($\boldsymbol{t_1}$)), sk=(ρ, K, tr, Encode($\boldsymbol{s_1}$), Encode($\boldsymbol{s_2}$), Encode($\boldsymbol{t_0}$)))

Algorithm 5 Dilithium Sign

Input: $sk = (\rho, K, tr, s_{1_{enc}}, s_{2_{enc}}, t_{0_{enc}})$, $M \in \{0,1\}^*$
$\hat{\boldsymbol{A}} \in R_q^{k \times l} \leftarrow$ExpandA$(\rho)$
$\hat{\boldsymbol{s_1}} \leftarrow$NTT(Decode$(s_{1_{enc}})$)
$\hat{\boldsymbol{s_2}} \leftarrow$NTT(Decode$(s_{2_{enc}})$)
$\hat{\boldsymbol{t_0}} \leftarrow$NTT(Decode$(t_{0_{enc}})$)
$\mu \leftarrow$SHAKE256$(tr || M)$
$\rho' \leftarrow$SHAKE256$(K || \mu)$
$k \leftarrow 0; done \leftarrow 0$
while $done == 0$ **do**
  $\boldsymbol{y} \in S_{\gamma_1}^l \leftarrow$ExpandMask$(\rho', k)$
  $\hat{\boldsymbol{y}} \leftarrow$NTT$(\boldsymbol{y})$
  $\boldsymbol{w} \leftarrow$NTT$^{-1}(\hat{\boldsymbol{A}} \circ \hat{\boldsymbol{y}})$
  $w_1 \leftarrow$HighBits$_q(w, 2\gamma_2)$
  $\tilde{c} \leftarrow$SHAKE256$(\mu || w_1)$
  $c \leftarrow$SampleInBall$(\tilde{c})$
  $\hat{c} \leftarrow$NTT(c)
  $\boldsymbol{z} \leftarrow$NTT$^{-1}(\hat{\boldsymbol{y}} + \hat{c} \circ \hat{\boldsymbol{s_1}})$
  $r_0 \leftarrow$LowBits$_q(\boldsymbol{w} -$NTT$^{-1}(\hat{c} \circ \hat{\boldsymbol{s_2}}), 2\gamma_2)$
  $h \leftarrow$MakeHint$($NTT$^{-1}(-\hat{c} \circ \hat{\boldsymbol{t_0}}), \boldsymbol{w} -NTT^{-1}(\hat{c} \circ \hat{\boldsymbol{s_2}} - \hat{c} \circ \hat{\boldsymbol{t_0}}), 2\gamma_2)$
  if $||\boldsymbol{z}||_\infty \geq \gamma_1 - \beta$ or $||\boldsymbol{r_0}||_\infty \geq \gamma_2 - \beta$ or $||c\boldsymbol{t_0}||_\infty \geq \gamma_2$ or $wt(h) > \omega$ **then**
    $k \leftarrow k + l$
  else
    $done \leftarrow 1$
  end if
end while
$z_{enc} \leftarrow$Encode$(\boldsymbol{z})$
return $\sigma = (\tilde{c}, z_{enc}, h)$

Key Generation: Like in Kyber, all polynomials are sampled from the output of SHAKE128 or SHAKE256. However, all samples are generated using uniform sampling with the public matrix $\mathbf{A}$ being uniformly sampled in the range $[0, q-1]$ and the secret polynomials $\boldsymbol{s_1}, \boldsymbol{s_2}$ being sampled in the range $[-\eta, \eta]$. In order to reduce the transmission cost of the public key, only the upper bits of $\boldsymbol{t}$ are included in the public key. In order to account for the missing lower bits, the signature will require a hint based on the lower bits of $\boldsymbol{t}$ and the secret polynomials. The pseudocode for key generation is shown in Alg. 4.

Sign: Signature generation is the most complex operation of Dilithium. The goal is to generate a polynomial pair (z, c). However, as previously stated, a hint is also needed for the verifier since

Algorithm 6 Dilithium Verify

Input: $pk = (\rho, t_{1_{enc}})$, $M \in \{0,1\}^*$, $\sigma = (\tilde{c}, z_{enc}, h)$
$\hat{\boldsymbol{A}} \in R_q^{k \times l} \leftarrow$ExpandA($\rho$))
$\boldsymbol{z} \leftarrow$Decode($z_{enc}$)
$\boldsymbol{t} \leftarrow$Decode($t_{enc}$)
$\mu \leftarrow$SHAKE256(SHAKE256($\rho||t_1$)$||M$)
$c \leftarrow$SampleInBall($\tilde{c}$)
$\hat{\boldsymbol{t_1}} \leftarrow$NTT($\boldsymbol{t_1}$)
$\hat{\boldsymbol{z}} \leftarrow$NTT($\boldsymbol{z}$)
$w_1' \leftarrow$UseHint$_q$(h, NTT$^-$1($\hat{\boldsymbol{A}} \times \hat{\boldsymbol{z}} - c \times \hat{\boldsymbol{t_1}}$))
if $||\boldsymbol{z}||_\infty < \gamma_1 - \beta$ and $\tilde{c}$ ==SHAKE256($\mu||w_1'$) and $wt(h) \leq \omega$ **then**
  return Accept
else
  return Reject
end if

Table 2: Parameters for round 3 Dilithium submission

	Sec. Level	n	(k,l)	q	η	γ_1	γ_2	pk (B)	sk (B)	sig (B)
Dilithium2	2	256	(4,4)	8380417	2	2^{17}	$(q-1)/88$	1312	2528	2420
Dilithium3	3	256	(6,5)	8380417	4	2^{19}	$(q-1)/32$	1952	4000	3293
Dilithium5	5	256	(8,7)	8380417	2	2^{19}	$(q-1)/32$	2592	4864	4595

only the upper bits of $\boldsymbol{t}$ are included in the public key. The signer begins by decoding the secret key and converting the polynomials back into the NTT domain. They then hash the message and byte arrays from the secret key into the seed rho' which is used to generate candidate signatures. The uniformly sampled $\boldsymbol{y}$ vector is used both to calculate $\hat{c}$ and to hide value of the secret polynomial in the calculation of $\boldsymbol{z}$. The hint is then calculated based on $c, \boldsymbol{t_0}, \boldsymbol{s_2}$, and $\boldsymbol{w}$. If the coefficients of $\boldsymbol{z}$, $\boldsymbol{r_0}$, or $c\boldsymbol{t_0}$ exceed the defined boundaries, or if the hint exceeds the maximum size, then the signature is rejected. A new attempt is generated by incrementing the nonce that is appended to ρ' before hashing.

Verification: Verification attempts to recreate the $\hat{c}$ seed using the signature and public key. It begins by decoding the signature and public polynomials and by hashing the message and public key to recreate μ. It then calculates $\boldsymbol{A} \times \boldsymbol{z} - c \times \boldsymbol{t_1}$, applies the hint, and hashes the upper bits. If the signature was valued and the hash matches $\hat{c}$, then it is accepted. This will succeed for a valid signature and public key because of the following equivalence:

$$\boldsymbol{A} \times \boldsymbol{z} - c \times \boldsymbol{t}$$
$$\boldsymbol{A} \times (\boldsymbol{y} + c \times \boldsymbol{s_1}) - c \times (\boldsymbol{A} \times \boldsymbol{s_1} + \boldsymbol{s_2})$$
$$\boldsymbol{A} \times c \times \boldsymbol{s_1} + \boldsymbol{A} \times \boldsymbol{y} - c \times \boldsymbol{A} \times \boldsymbol{s_1} - c \times \boldsymbol{s_2}$$
$$\boldsymbol{A} \times \boldsymbol{y} - c \times \boldsymbol{s_2} \approx \boldsymbol{A} \times \boldsymbol{y}$$

Since c is a short polynomial and $\boldsymbol{s_2}$ has small coefficients, the effect of its subtraction will not impact the upper bits of the result. Additionally, the carry bits from the lower bits of $\boldsymbol{t}$ are accounted for by the hint and so W_1' will equal W_1 and c will be correctly recreated.

2.4 Common Operations

Kyber and Dilithium have many low level operations in common. The most costly operations for both algorithms are polynomial sampling using Keccak and polynomial arithmetic using NTT multiplication. Both algorithms use the two operations in similar but slightly differing ways. Both sample polynomials from the output of the SHAKE functions, however only Kyber uses CBD sampling. Both use the NTT to accelerate polynomial multiplication, however Kyber uses an incomplete NTT due to its small modulus size which requires a more complex operation for point-wise multiplication. Both algorithms also use different coefficient moduli and have different length vectors and matrices. These differences must be accounted for when designing a shared implementation.

2.5 Hybrid Mode Operation

During the the transition period to quantum-secure cryptography, some applications may want to deploy both classical and quantum-resistant key exchanges in a hybrid operation mode. This configuration increase the performance cost of the key exchange, but provides security in the case that either cryptosystem is broken. Thus the system is still secure against quantum attacks, but if a classical attack is found that weakens the new standard, the connection is still secured by the classical algorithm. An example configuration is shown in Fig. 1, where the classical and quantum-secure algorithms are configured in parallel and the output of both is used as input for a Key Derivation Function (KDF) which generates the shared secret key.

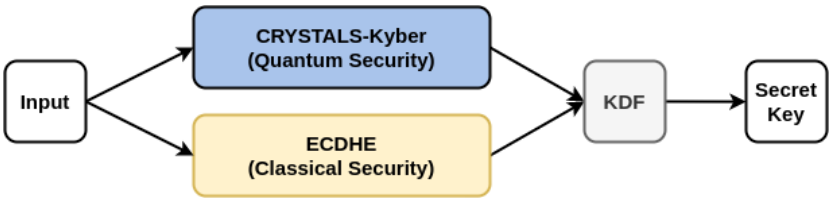

Figure 1: Example configuration for hybrid cryptographic deployment.

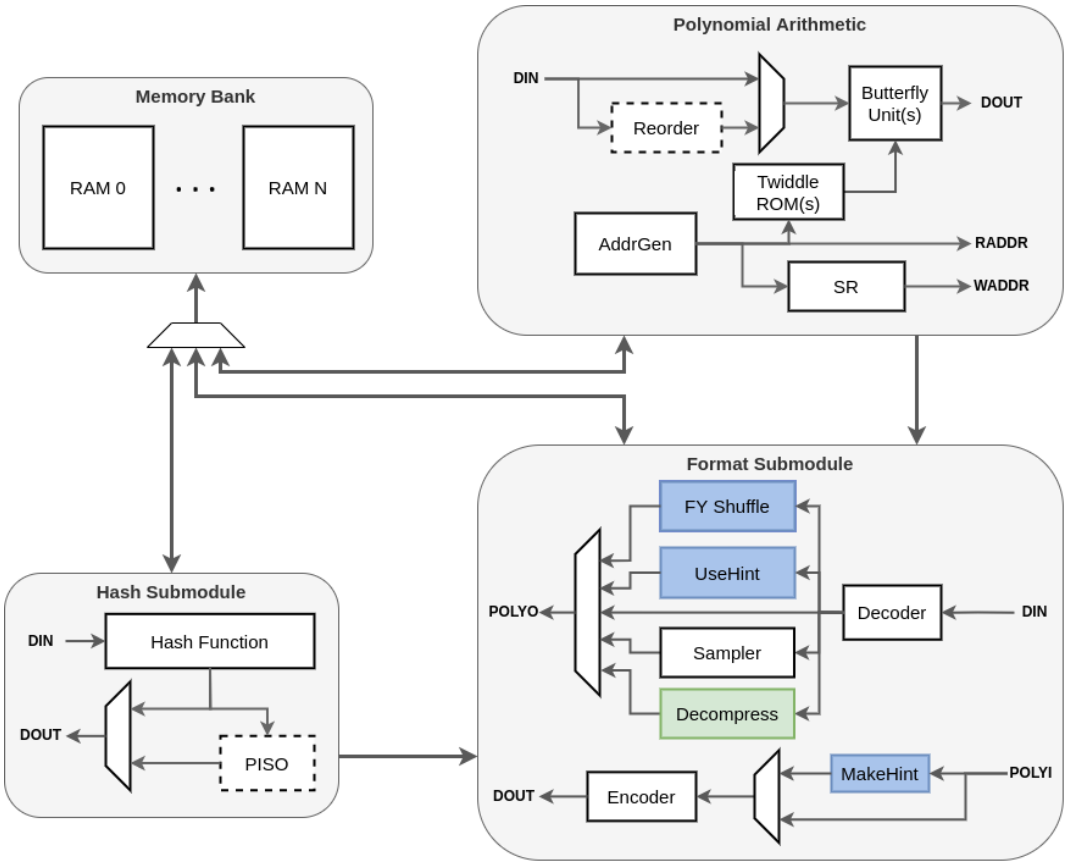

Figure 2: Top level block diagram. Blue modules are Dilithium-only, green are Kyber-only, dashed lines are used in higher performance designs.

3 Methodology

In this section we provide an overview of the design of our three levels of performance for Kyber, Dilithium, and the combined design. The first design focuses on low resource consumption and is

designated lightweight or LW. The second is the mid-range design, also designated by MR, which seeks to provide strong performance with reasonable area. The third is high speed, designated by HS, which prioritizes low latency. The high level view of the architecture is shown in Fig. 2. All designs use the same high level architecture, with performance trade-offs made by changing the submodule designs and top level databus widths.

3.1 Polynomial Arithmetic

Polynomial arithmetic is one of the most costly operations in lattice-based cryptosystems, particularly NTT based polynomial multiplication. Thus having optimized NTT and point-wise multiplication is important for compact and high performance designs. Our polynomial arithmetic unit is optimized for each of our three design levels. The polynomial operations are performed by units called butterflies, named after the NTT sub-operations. Each butterfly can perform all of the following operations: Cooley-Tukey butterfly for the NTT, Gentlemen-Sande butterfly for the inverse NTT, point-wise multiplication, point-wise addition, and point-wise subtraction. Further, if the design is instantiated to support Kyber, Karatsuba multiplication is supported as well as compression which can be performed after addition and subtraction. If the instantiation is configured to only support Dilithium, these capabilities are disabled to reduce area. If the design is configured to support Dilithium, the butterfly is also capable of performing the decomposition operation. All operations are fully pipelined, meaning a result is produced every cycle once the pipeline is filled, with the exception of Karatsuba multiplication. Since the Karatsuba multiplication on the polynomial pairs requires four multiplications and each butterfly has only one multiplier, it can accept an input every four clock cycles.

Most of the polynomial arithmetic architecture can be easily shared between Kyber and Dilithium. Within the butterfly, the datapath width is 23-bit for the Dilithium only and combined instances, and 12-bit for Kyber only. The modular multiplier can be configured to support one or both moduli as well as to support the compression operation of Kyber and the decomposition operation of Dilithium. The NTT control logic can be reused for both with the only modification needed being support for an early abort signal to skip the final layer of the Kyber NTT. The control logic of the algorithm specific operations can be selectively enabled as needed based on the configuration of the instance.

The lightweight design uses a single pipelined butterfly as shown in the bottom right of Fig. 3. The controller uses the "ping-pong" method for the NTT where coefficients are read and written back and forth between two memories to allow full utilization of the butterfly. Coefficients are stored in 1×256 arrays in dual port memory, so each address maps to one coefficient. Since Kyber uses a partial NTT, the latency is $7 \times 128 + d = 896 + d$ cycles where d is the pipeline length, the latency for Karatsuba multiplication is $128 \times 4 + d = 512 + d$ cycles, and the latency for all other operations is $256 + d$.

The mid-range design uses two butterflies operating in parallel. In order to increase throughput without needing additional memories, the dimensions of the BRAM are reconfigured to store polynomials as a 2×128 array, so each address gives access to two coefficients. This allows nearly identical control logic to the lightweight design while doubling the performance. The Kyber NTT requires only $7 \times 64 + d = 448 + d$ cycles, the latency for Karatsuba multiplication is $64 \times 4 + d = 256 + d$ cycles, and the latency for all other operations is $128 + d$. This design is shown in the bottom left of Fig. 3.

The high performance design requires more substantial changes to increase performance. We use a 2×2 butterfly similar to the approach described in [9],[16]. However, to reduce the need for additional resources we continue to use the ping-pong method for memory access rather then the coefficient reordering described in [9]. This design continues to store polynomials as an 2×128 array, using a reorder buffer to properly align the coefficients before loading them into the butterflies. Since Kyber uses an odd number of layers, the last two butterfly units can be bypassed. The NTT requires only $4 \times 64 + d = 256 + d$ cycles, the latency for Karatsuba multiplication is $32 \times 4 + d = 128 + d$ cycles, and the latency for all other operations is $64 + d$. This design is shown in the top of Fig. 3.

3.2 Hashing and Sampling

Another potential bottleneck in CRYSTALS is hashing and the sampling of polynomials using the SHA3 SHAKE function. Each polynomial contains 256 coefficients, for Kyber the uniform polynomials of the $\mathbf{A}$ matrix are in the range $[0, q]$ and require 12-bits each to sample and the CBD polynomials are in the range $[-\eta, \eta]$ and require 4 or 6 bits depending on the value of η. On average, each polynomial of $\mathbf{A}$ requires 3 permutations of SHAKE128, and each CBD polynomial requires 1 and 2

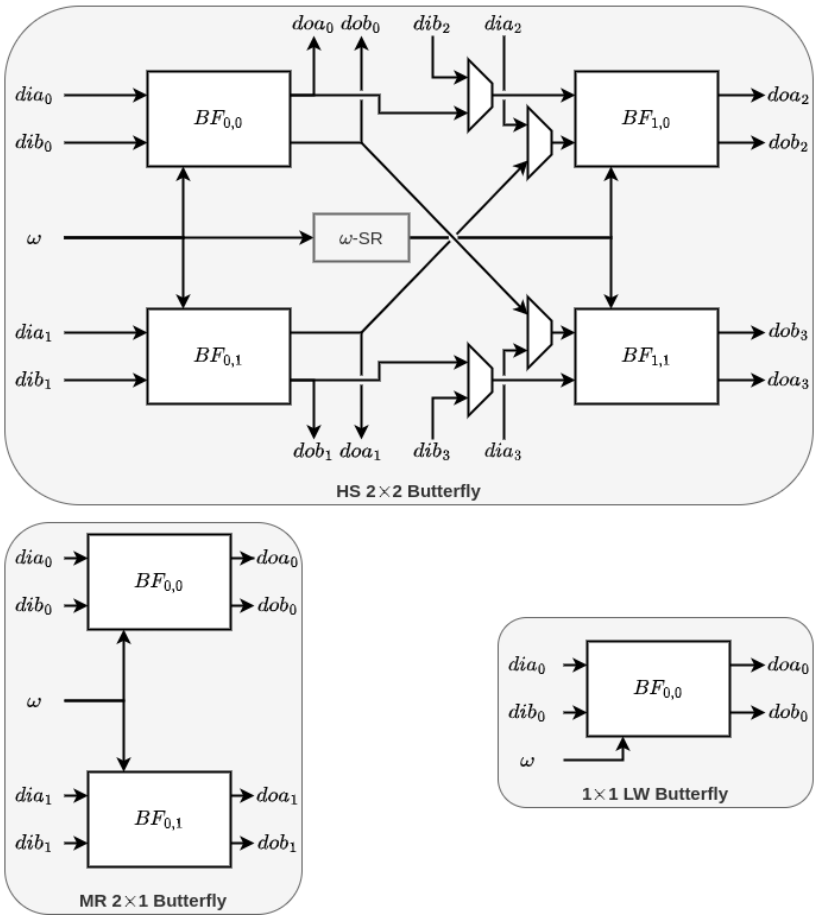

Figure 3: Butterfly configurations for high speed, mid-range, and lightweight designs

permutations of SHAKE256 depending on the value of η. For Dilithium, the modulus is 23-bits so each coefficient of $\mathbf{A}$ requires 24-bits of randomness. The $\mathbf{y}$ coefficients require 18 or 20 bits, and the secret coefficients require 4 bits. So, on average each polynomials of $\mathbf{A}$ requires 5 permutations of SHAKE128, each polynomial of $\mathbf{y}$ requires 5 permutations of SHAKE256, and each small polynomial requires 2 permutations of SHAKE256. In order to achieve a reasonable level of performance, all designs use a high performance Keccak core which completes the permutation in 24 clock cycles.

The lightweight design use the Keccak output directly for sampling polynomial coefficients. For the mid-range and high performance designs, a PISO is instantiated which unloads the entire Keccak state in one cycle and then feeds it to the sampler as the next permutation runs in parallel. For the mid-range design two coefficients. are sampled per cycle from the PISO, for the high performance design four coefficients are sampled per cycle. This approach ensures that a polynomial is sampled in fewer cycles than the NTT operation, which allows polynomials to be sampled "on the fly" just before they are needed, reducing the amount of memory needed.

3.3 Polynomial Encoding and Decoding

For both Kyber and Dilithium, all output polynomials need to be converted from unpacked coefficient form to an encoded byte-array as well as several intermediate polynomials which are hashed. This involves converting centered coefficients to positive only values by mapping $(-a, a) \rightarrow (0, 2a)$. For a coefficient $x \in (-a, a)$, this is done by $x' = a - x$. Once all coefficients are converted to positive integers, encoding is a simple bus width conversion. So if the coefficients of a polynomial of degree N require b bits to represent in binary, the encoded results will be $N \times b/8$ bytes. In our design, all encoding is performed using a single module with a single shift register based bus width converter. The lightweight design accepts one coefficient input at a time, the mid-range accepts two, and the high

performance accepts four.

A similar approach is used for the decoding module. A single shift register based decoder is used for unpacking polynomials. This module is also used as as a bus width converter for all of the sampling modules. For example in the lightweight design, when performing rejection sampling for Kyber, the decoder receives input from the hash function 32-bits at a time and converts it to the 24-bits at a time needed for the sample. This module is also used for the hint decoding and the Fisher-Yates shuffle used to sample the challenge polynomial in Dilithium.

The number of coefficients processed in parallel scales with the design. Polynomials are decoded in parallel with arithmetic operations and are encoded directly from the output of the final arithmetic operations, thus they must be scaled to prevent encoding and decoding from becoming the bottleneck. For the lightweight design one coefficient is processed per cycle, for the mid range two are, and for the high performance design four are.

3.4 Side Channel Protection of Kyber

Side-channel attacks pose a serious threat to cryptographic implementations. Several power analysis attacks have been reported on lattice-based algorithms, which can lead to the recovery of the private key or the shared secret key in key encapsulation.

There have been several previous works on side-channel-resistant lattice-based implementations in the literature. For example, Fritzmann et al. presented masked hardware accelerators for Saber and Kyber that speed up hashing, sampling, and compression, among other tasks [17] in their HW/SW codesign implementation. The accelerators were designed to work with a RISC-V processor. In this implementation, Kyber-768 decapsulation was reported to require 1.23 million cycles. Bos et al. described first and high order masked Kyber in software that needs 3.1 million cycles for first order protected implementation [18]. Heinz et al. described a masked Kyber implementation on ARM Cortex-M4 that required around 3 million cycles to perform decapsulation for Kyber-768 [19]. To the best of our knowledge, there has not been a publicly reported masked full hardware implementation of Kyber.

We developed a full hardware masked implementation of Kyber-KEM designed to resist first-order differential power analysis (DPA) and timing attacks. Our design focuses on protecting the long term private key used during decapsulation as well as the shared symmetric session key. As shown in previous literature [17] all intermediate values derived from the private key may be targeted by side-channel attacks and consequently, must not be leaked.

We utilized masking to split all sensitive values into two shares and process them such that we ensure first-order security even in the presence of glitches that occur in hardware implementations. Additionally, we utilize shuffling in the NTT unit to provide further protection against power side-channel attacks.

The top-level block diagram is shown in Fig. 4. Inputs and outputs are received and sent in two shares. Internally, data that belong to the two shares are stored in two separate sets of two memory banks, RAM bank 1 and 2. Non-sensitive information such as ciphertext and the public key are saved in RAM bank 3. In Fig. 4, we use green, blue and black arrows to represent the first data share, the second data share and non-sensitive data, respectively. We use interconnect units to connect memories to processing units. All units use decoupled I/O and have a configuration interface to allow simple control logic. Our architecture uses a polynomial arithmetic unit capable of processing two shares in parallel employing two NTT butterflies. The hash-sampling units performs all SHA3 operations, rejection sampling, and CBD sampling.

The auxiliary unit performs share type conversion, ciphertext compression, and message decoding. These operations are bundled together to allow resource sharing among the components.

Since the hash-sampling and the auxiliary units mix two shares in non-linear operations, they utilizes randomness generated from the pseudo-random number generator to refresh the shares.

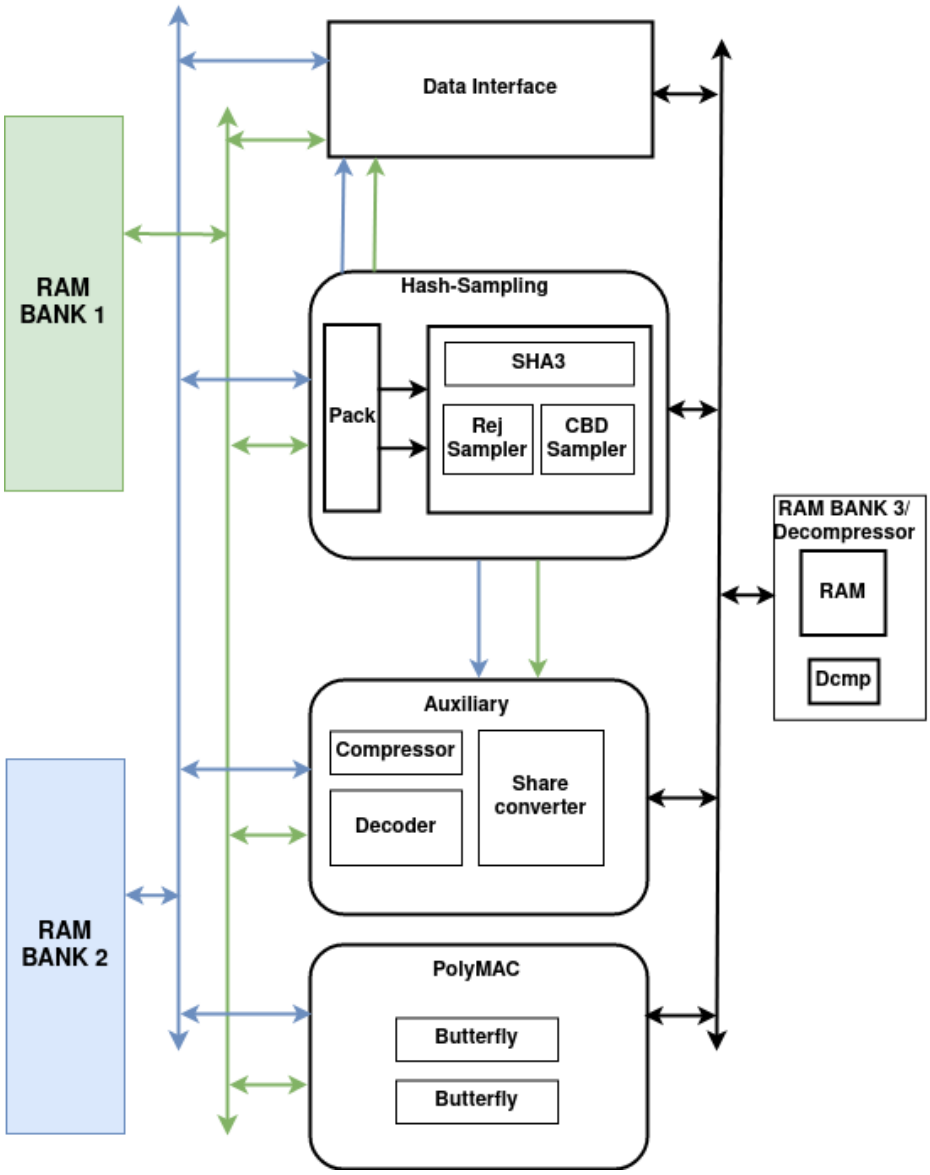

Figure 4: Simplified top-level architecture of masked hardware Kyber. The Green, blue and black arrows represent the first data share, the second data share and non-sensitive data, respectively.

4 Results

In this section we will will discuss the results of our design and compare them with other works on Kyber and Dilithium. As there have been many works on these algorithms, we will focus only on the state of the art designs. In particular, the lightweight Kyber implementations by Xing et al [6], the high performance Kyber design by Dang et al [7], the high performance Dilithium design by Zhao et al [10], and the combined architecture for both algorithms by Akaita et al [12]. The performance comparisons of cycle counts for Kyber and Dilithium are show in Figs. 6-9. For the sake of clarity, the performance comparison for Dilithium signing is split into a separate figure as it is substantially slower than key generation and verify. For all polar charts, a smaller area represent a more efficient design for the relevant metric - i.e. lower latency or lower area.

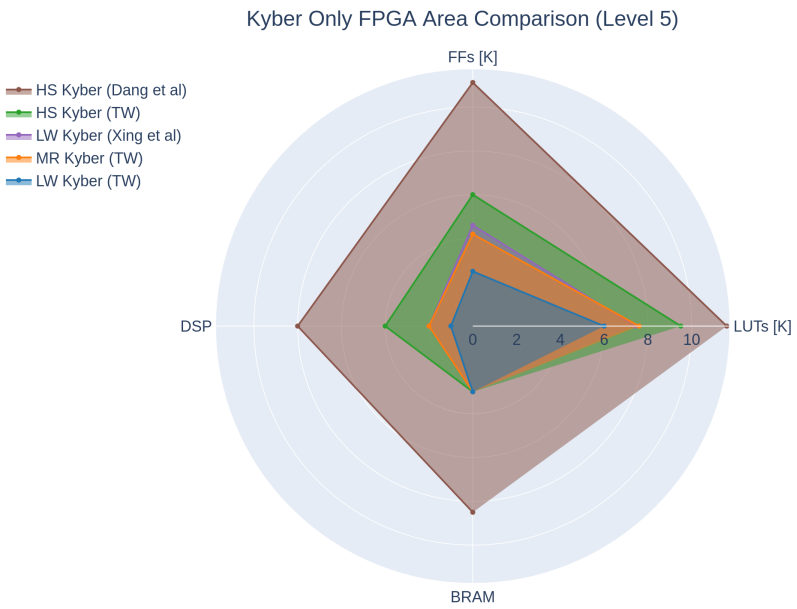

Figure 5: Comparison of resource utilization for Kyber only implementations

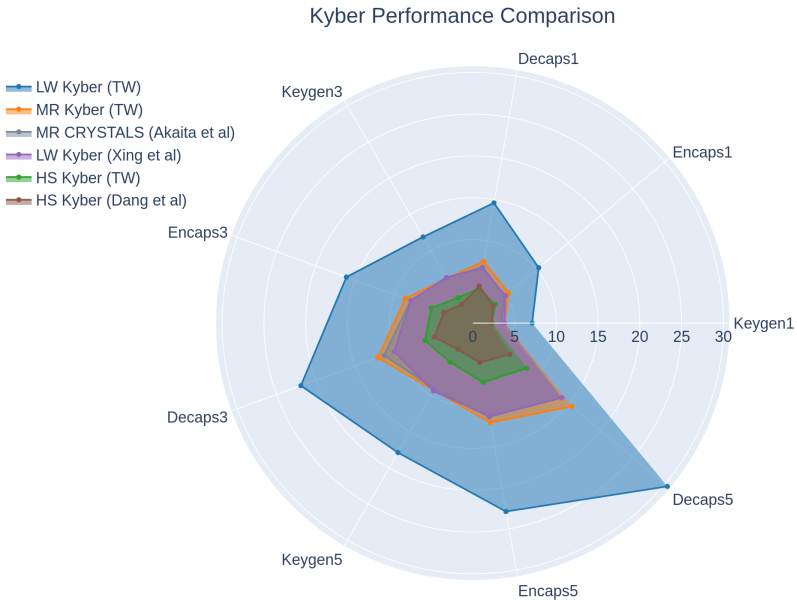

Figure 6: Comparison of Kyber performance by cycle count at all security levels

4.1 Kyber Comparison

The direct comparison of area for the Kyber only designs is shown in Fig. 5 and the performance is compared in Fig. 6. Compared to the previous work by Xing et al [6], our mid-range design which supports all operations and security levels is able to achieve similar performance and area to the server-side implementation which only supports key generation and decapsulation. The design by Xing et al also presume that some operations like hashing the public key may be performed in advance to improve performance, whereas our design achieves this performance performing all operations on demand.

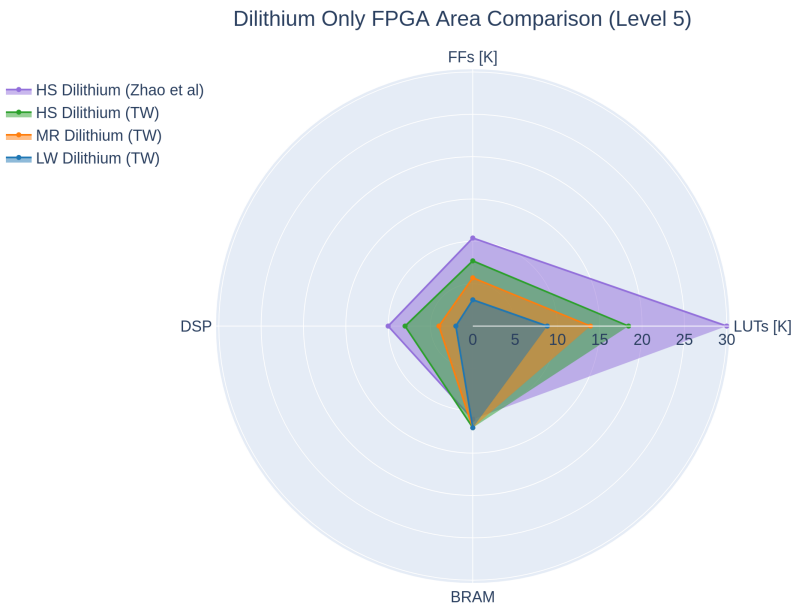

Figure 7: Comparison of resource utilization for Dilithium only implementations

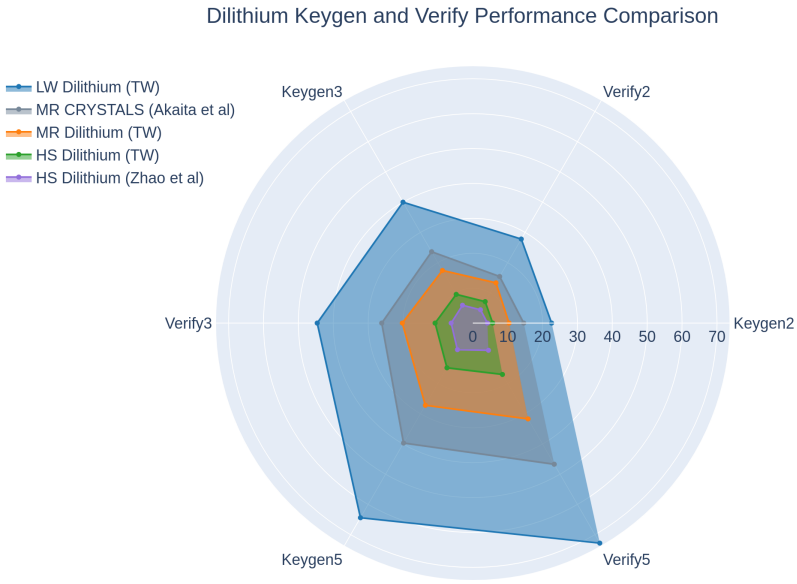

Figure 8: Comparison of Dilithium keygen and verify performance by cycle count at all security levels

Additionally, since each individual butterfly unit in our design can perform Karatsuba multiplication, we are able to scale down our design further than the design proposed by Xing et al which uses the two butterflies in series to performance Karatsuba multiplication. This allows our lightweight design to achieve the lowest area to date for a Kyber implementation, to the best of our knowledge.

The high performance design by Dang et al [7] achieves better performance at higher security levels, however it does so by having design optimizations for each specific security level. In particular, the polynomial arithmetic unit is instantiated k times where k is the dimension of the vectors for a particular security level of Kyber. This is efficient for optimizing performance for a particular security

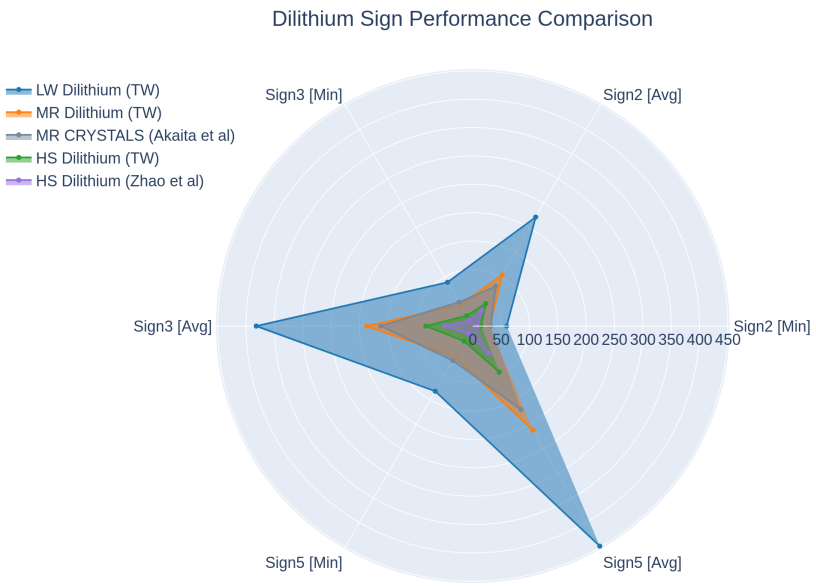

Figure 9: Comparison of Dilithium sign performance by cycle count at all security levels

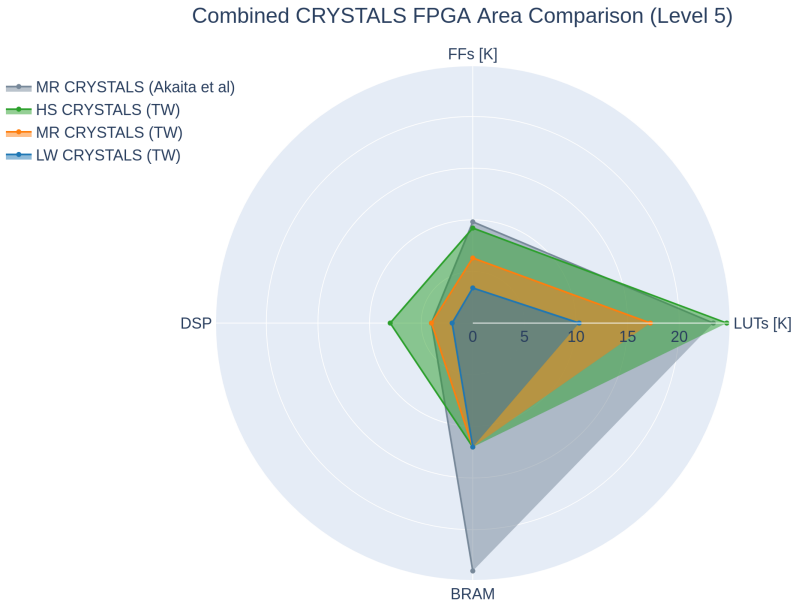

Figure 10: Comparison of resource utilization for combined CRYSTALS implementations

level, however it makes the design less flexible. Our approach of using the 2×2 butterfly means that our design can efficiently perform all security levels in a single instance. The benefit of this approach is shown in Fig. 5, our high performance design uses substantially fewer resources than their Kyber1024 implementation while supporting all security levels and having competitive performance.

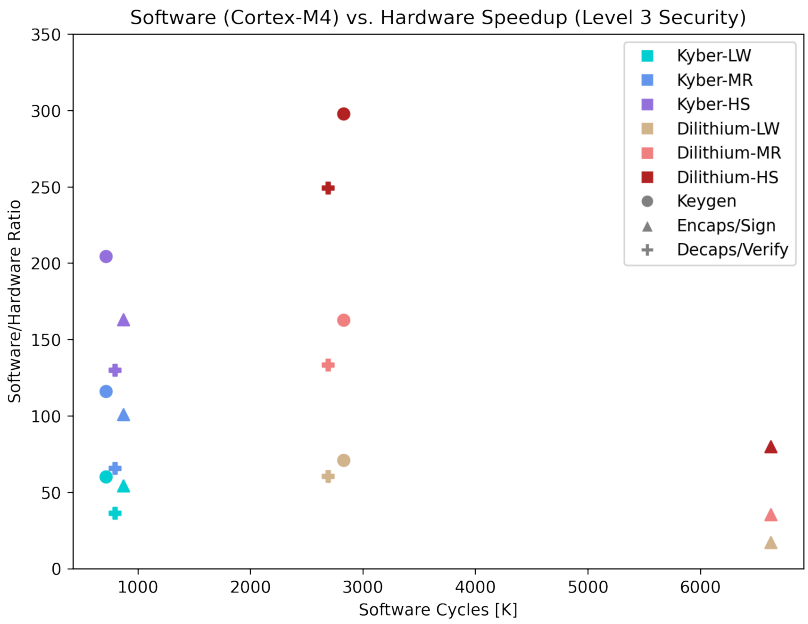

Figure 11: Performance improvement from hardware acceleration of software

4.2 Dilithium Comparison

The direct comparison of area for the Dilithium only designs is shown in Fig. 7 and the performance is compared in Figs. 8 and 9. The design by Zhao et al [10] has the best area and performance to date and thus will be the focus of our comparison. The design performs substantially faster than our mid-range and lightweight designs, however compared to our mid range design, the design by Zhao et al uses over 2× the number of LUTs and registers, 2.5× the number of DSP, but one fewer BRAM. Compared to our lightweight design it uses 3× the number of LUTs and registers and 5× the number of DSP. Our high performance design achieves competitive performance with substantially lower LUT and FF utilization.

4.3 Combined Comparison

The direct comparison of area for the combined designs is shown in Fig. 10. The performance numbers are shown in the Kyber and Dilithium performance figures. In terms of performance, the combined design by Akaita et al [12] is similar to the performance of our mid-range implementation. For Kyber the design by Akaita is 10% higher performance, for Dilithium the design is 25% slower for key generation and verification but 20% faster for average case signing. However, as shown in Fig. 10, substantially more resources are required for this slight performance benefit with their design using 30% more LUTs, 55% more registers, and 100% more BRAM than our design. Thus this work presents the most compact combined architecture thus far for Kyber and Dilithium. Additionally, our high performance design achieves substantially lower latency for similar area costs.

4.4 Comparison with Software

We also compare our work with optimized software on embedded devices. Fig. 11 shows the performance improvement when comparing the hardware implementations to an assembly optimized software running on a Cortex-M4 processor for level 3 security in both algorithms. For Kyber encapsulation and decapsulation, our lightweight design is over 50× faster the optimized software, the mid-range design is over 100×, and our high performance design is over 150× and 200× faster respectively. For Dilithium the key generation and verification follow a similar trend, though signing is only $20 - 80\times$

faster.

4.5 Side Channel Protection Results

Our masked implementation utilizes 18K LUTs, 8 DSPs and 7 block RAM units when instantiated on Xilinx Artix 7 FPGAs, and the key decapsulation for Kyber-512 requires 51K cycles.

To verify side-channel resistance, we used fixed-vs-random Test vector Leakage Assessment (TVLA) using 10,000 traces. The fixed vectors use a fixed private key, while the random vectors use a random private key. In all cases, the ciphertext and the public key are kept fixed. The core was instantiated in Chipwhisperer CW305 Artix7-based board, which was clocked at 10 MHz. A Picoscope3000 oscilloscope was used to collect power measurements at 125 M Samples/sec. The power was measured from the CW305's onboard amplifier, which amplifies the voltage drop over a 0.1Ω shunt resistor.

We performed a first TVLA test on Kyber decapsulation with no randomness provided; hence masking is disabled. This test is expected to show spikes above the TVLA threshold. The result of this test, shown in Fig. 12, is used as a baseline to assess improvement when randomness is added and proves the ability of the test setup to detect leakage. Another test was performed with randomness provided to enable countermeasures. The result of this test is shown in Fig. 13. In this test, all t-values are below the threshold, confirming the effectiveness of the side-channel protection.

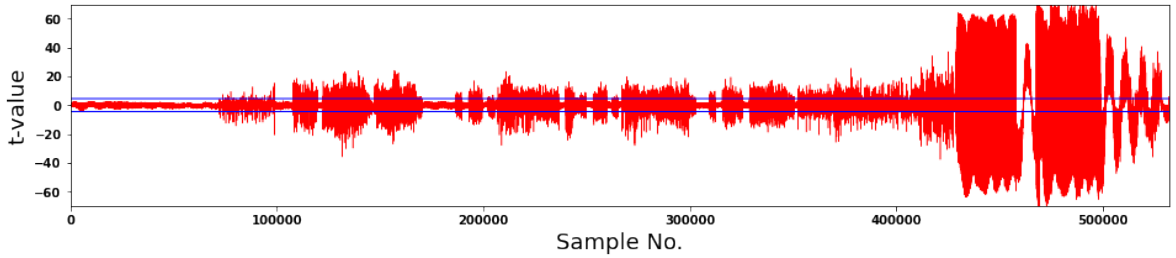

Figure 12: Kyber TVLA with randomness disabled (i.e. disabling countermeasures) showing leakage

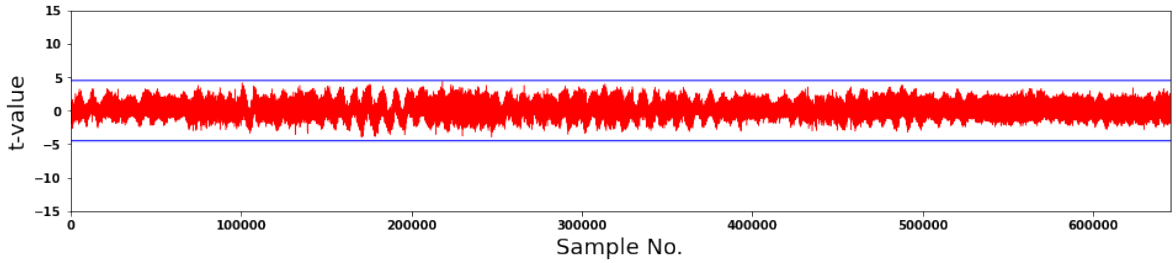

Figure 13: Kyber TVLA with randomness enable (i.e. enabling countermeasures) showing no leakage

5 Conclusions

In this work we have presented a flexible and combined architecture for the future cryptographic standards, Kyber and Dilithium. The design presented fits many applications including embedded devices that prioritize low area and energy as well as high performance applications that require low latency. Further, we have presented the first full hardware masked implementations that is secure against first order power analysis attacks for Kyber. This effort will be continued with a masked implementation of Dilithium and the combined CRYSTALS architecture.

IP Statement

This architecture described in this work is the property of PQSecure Technologies LLC and is currently patent pending.

References

[1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," en, *SIAM Journal on Computing*, vol. 26, no. 5, Oct. 1997, arXiv:quant-ph/9508027. [Online]. Available: http://arxiv.org/abs/quant-ph/9508027.

[2] W. Castryck and T. Decru, "AN EFFICIENT KEY RECOVERY ATTACK ON SIDH (PRELIMINARY VERSION)," *Cryptology ePrint Archive 2022/975*,

[3] W. Beullens, "Breaking Rainbow Takes a Weekend on a Laptop," *Cryptology ePrint Archive 2022/214*,

[4] D. Moody, "Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. NIST IR 8413, 2022. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf.

[5] NSA, *Cybersecurity Advisory Announcing the Commercial National Security Algorithm Suite 2.0*, Sep. 2022. [Online]. Available: https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF.

[6] Y. Xing and S. Li, "A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Feb. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8797.

[7] V. Dang, K. Mohajerani, and K. Gaj, "High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber," *Cryptology ePrint Archive 2021/1508*,

[8] G. Land, P. Sasdrich, and T. Güneysu, "A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware," in *Smart Card Research and Advanced Applications*, V. Grosso and T. Pöppelmann, Eds., Cham: Springer International Publishing, 2022, pp. 210–230.

[9] L. Beckwith, D. T. Nguyen, and K. Gaj, "High-Performance Hardware Implementation of CRYSTALS-Dilithium," *2021 International Conference on Field-Programmable Technology (ICFPT)*, 2021.

[10] C. Zhao, N. Zhang, H. Wang, *et al.*, "A Compact and High-Performance Hardware Architecture for CRYSTALS-Dilithium," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9297.

[11] A. C. Mert, D. Jacquemin, A. Das, D. Matthews, S. Ghosh, and S. S. Roy, "A Unified Cryptoprocessor for Lattice-based Signature and Key-exchange," *Cryptology ePrint Archive 2021/1461*,

[12] A. Aikata, A. C. Mert, M. Imran, S. Pagliarini, and S. S. Roy, "KaLi: A Crystal for Post-Quantum Security," *Cryptology ePrint Archive 2022/1086*,

[13] R. Avanzi, J. Bos, L. Ducas, *et al.*, "Kyber - Algorithm Specifications and Supporting Documentation," [Online]. Available: https://pq-crystals.org/kyber/resources.shtml.

[14] S. Bai, L. Ducas, E. Kiltz, *et al.*, "CRYSTALS-Dilithium," [Online]. Available: https://pq-crystals.org/dilithium/resources.shtml.

[15] G. Goos, J. Hartmanis, J. van Leeuwen, E. Fujisaki, and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed., vol. 1666, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. [Online]. Available: http://link.springer.com/10.1007/3-540-48405-1_34.

[16] D. T. Nguyen, V. B. Dang, and K. Gaj, "High-Level Synthesis in Implementing and Benchmarking Number Theoretic Transform in Lattice-Based Post-Quantum Cryptography Using Software/Hardware Codesign," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, F. Rincón, J. Barba, H. K. H. So, P. Diniz, and J. Caba, Eds., Cham: Springer International Publishing, 2020, pp. 247–257.

[17] T. Fritzmann, M. Van Beirendonck, D. Basu Roy, *et al.*, "Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography," en, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Nov. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/9303.

[18] J. W. Bos, M. Gourjon, J. Renes, and T. Schneider, "Masking Kyber: First- and Higher-Order Implementations," *Cryptology ePrint Archive 2021/483*,

[19] D. Heinz, M. J. Kannwischer, G. Land, P. Schwabe, and D. Sprenkels, "First-Order Masked Kyber on ARM Cortex-M4," *Cryptology ePrint Archive 2022/058*,