High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber

Viet Ba Dang, Kamyar Mohajerani and Kris Gaj

Cryptographic Engineering Research Group,
George Mason University
Fairfax, VA, U.S.A.

Abstract. Performance in hardware has typically played a significant role in differentiating among leading candidates in cryptographic standardization efforts. Winners of two past NIST cryptographic contests (Rijndael in case of AES and Keccak in case of SHA-3) were ranked consistently among the two fastest candidates when implemented using FPGAs and ASICs. Hardware implementations of cryptographic operations may quite easily outperform software implementations for at least a subset of major performance metrics, such as latency, number of operations per second, power consumption, and energy usage, as well as in terms of security against physical attacks, including side-channel analysis. Using hardware also permits much higher flexibility in trading one subset of these properties for another. This paper presents high-speed hardware architectures for four lattice-based CCA-secure Key Encapsulation Mechanisms (KEMs), representing three NIST PQC finalists: CRYSTALS-Kyber, NTRU (with two distinct variants, NTRU-HPS and NTRU-HRSS), and Saber. We rank these candidates among each other and compare them with all other Round 3 KEMs based on the data from the previously reported work.

Keywords: Post-Quantum Cryptography · lattice-based · Key Encapsulation Mechanism · hardware · FPGA

1 Introduction

Post-Quantum Cryptography (PQC) refers to a class of cryptographic algorithms that are resistant against all known attacks using quantum computers and can be implemented on traditional non-quantum computing platforms. These platforms include microprocessors, microcontrollers, graphics processing units (GPUs), Field Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs), and many others. The main goal of PQC is to replace the existing public-key cryptography standards based on RSA and Elliptic Curve Cryptography. These standards seem to be the most vulnerable to quantum computing and impossible to defend using traditional approaches such as *gradually* increasing key sizes [86, 14, 90, 39].

To initiate a timely transition to a new class of cryptographic schemes, in December 2016, NIST launched its PQC standardization process with the release of a "Call for Proposals and Request for Nominations for Public-Key Post-Quantum Cryptographic Algorithms" [72]. Sixty-nine submissions were judged complete and accepted for Round 1, which started in December 2017 [78, 1]. In January 2019, based on the initial security analysis and preliminary software benchmarking results, 26 submissions were qualified by NIST to Round 2. On July 22, 2020, NIST announced 15 candidates qualified for Round 3 of the standardization process. These candidates are summarized in Fig. 1. All

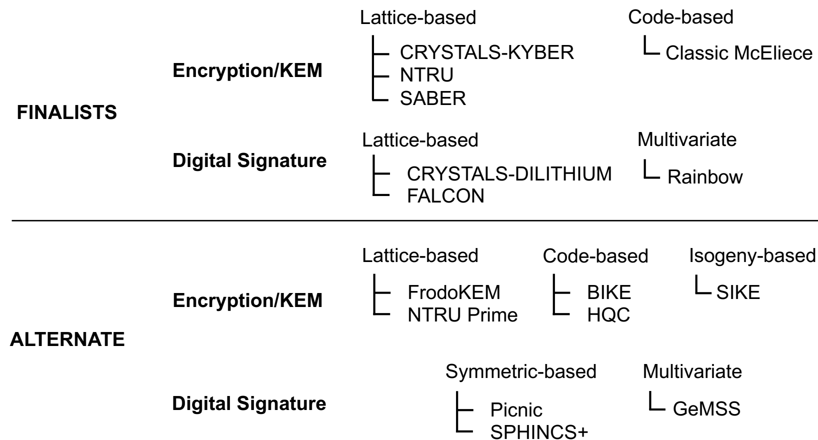

Figure 1: Finalists and alternate candidates qualified to Round 3 of the NIST PQC Standardization Process

Round 3 candidates represent five diverse families: lattice-based, code-based, multivariate, symmetric-based, and isogeny-based. Seven finalists are expected to be given priority in the standardization process. One encryption/KEM scheme and one digital signature scheme from this group may be selected as a PQC standard as early as 2022. Alternate candidates are treated as backup candidates. In Round 2, alternate candidates were judged to be either insufficiently investigated from the security point of view or were believed to lack some desired properties related to their performance (such as small public keys, small signatures, short execution time in software, etc.).

Hardware benchmarking has played a major role in all recent cryptographic standardization efforts, such as the AES, eSTREAM, SHA-3 [8, 37, 55, 57], and CAESAR contests [17, 20]. With the emergence of commonly-accepted hardware application programming interfaces (APIs) [43], development packages [42, 46], specialized optimization tools [38, 28], new design methodologies based on High-Level Synthesis (HLS) [44, 45], and mandatory hardware implementations in the final round of the CAESAR contest [17], the percentage of initial submissions implemented in hardware grew from 27.5% in the SHA-3 contest [36] to 49.1% in the CAESAR competition [20, 35]. Unfortunately, this trend could not be sustained in the NIST PQC standardization process. In many respects, PQC schemes are diametrically different and at least an order of magnitude more complex to implement compared to those evaluated in previous cryptographic contests.

Choice of Algorithms to Implement.

There are only four KEM PQC finalists. Since the NIST announcement in July 2020, it is urgent to compare them against each other. In particular, NIST is tentatively planning to choose only one of these candidates for the first round of standardization at the end of 2021 or at the beginning of 2022. The excellent implementation of Classic McEliece was reported in 2017-2018. Thus, the efficient implementations of the remaining three KEM finalists are of utmost importance at this point. Consequently, in this paper, we aim at evaluating and contrasting the hardware efficiency of CRYSTALS-KYBER, NTRU, and Saber.

In terms of algorithm types, we focus on KEMs with indistinguishability under a chosen-ciphertext attack (IND-CCA). Our primary goal was to implement all lattice-based IND-CCA secure KEMs described in the specifications of PQC finalists. The submission package of NTRU describes two substantially different KEMs : NTRU-HRSS and NTRU-HPS. As a result, we have implemented four KEMs representing three PQC finalists. For each implemented KEM, we generated results for all supported security levels.

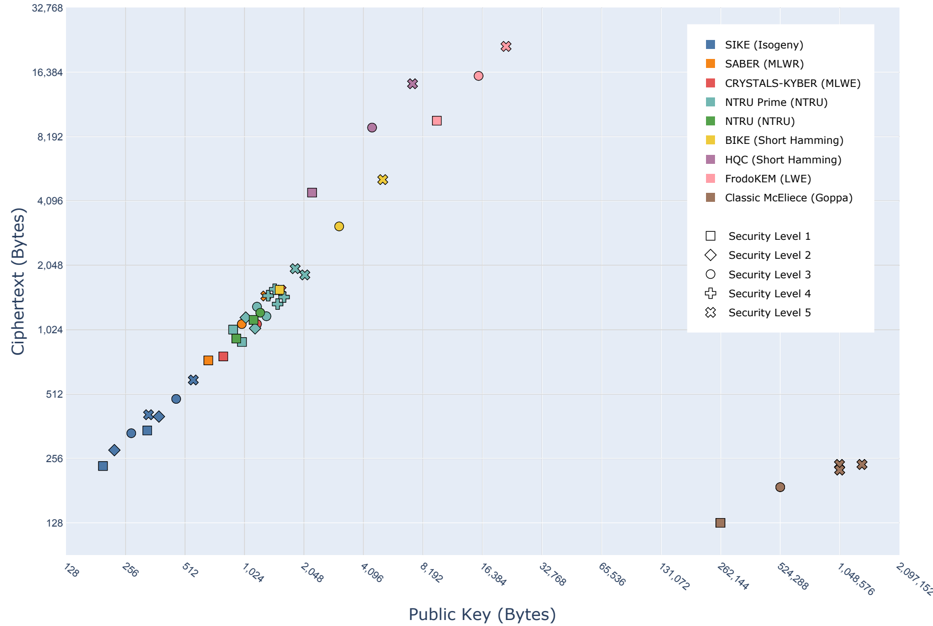

Figure 2: Relation between the ciphertext and public-key sizes for Round 3 PQC Key Encapsulation Mechanisms

In Fig. 2, we show the relationship between the ciphertext and public-key sizes of all Round 3 candidates. All schemes based on structured lattices - Saber, CRYSTALS-KYBER, NTRU Prime, and NTRU - have their ciphertext and public key sizes in the range between 512 and 2048 bytes. The only candidate better than them is an isogeny-based SIKE, which is still considered relatively recent and not sufficiently scrutinized from the security point of view. As a result, this scheme was qualified for Round 3 only as an alternate candidate. The only other PKE/KEM finalist, Classic McEliece, has public-key lengths between 0.25 and 2 Megabytes. So large public key sizes may significantly impact the sizes of data exchanged between two parties in the key establishment phase of any modern secure communication protocol, such as TLS, IPSec, SSH, etc. The sizes of keys and ciphertexts used by the selected lattice-based schemes are significantly smaller than those of the alternate code-based schemes, BIKE and HQC, and the unstructured-lattice scheme FrodoKEM. As a result, the key establishment time and the amount of memory required to store public-key certificates are also the most practical among all Round 3 candidates other than SIKE.

High-speed vs. lightweight. Assuming comparable technology, hardware implementations outperform software implementations using at least one, and typically multiple, metrics, such as latency, number of operations per second, power consumption, energy usage, and security against physical attacks. They also allow much higher flexibility in trading one subset of these metrics for another. From the point of view of benchmarking and ranking of candidates, such flexibility may become a curse, especially considering that no two metrics are likely to have a simple linear dependence on each other. A practical solution to this problem is to focus during the evaluation process on two major types of implementations: high-speed and lightweight.

In high-speed implementations, the primary target is speed, understood as either minimum latency (a.k.a. execution time) or the number of operations per second. For PQC schemes, this target amounts to optimizing the implementations of major operations involving the public and private key, respectively. For Key Encapsulation Mechanisms

(KEMs), these operations are encapsulation and decapsulation; for digital signature schemes, signature verification and generation; for public-key encryption (PKE), encryption and decryption. The time of key generation may also play a major role in the case when a public-private key pair cannot be reused for security reasons. The resource utilization is secondary. Still, hardware designers typically aim at achieving the Pareto optimality, in which any further speed improvement comes at a disproportionate cost in terms of resource utilization.

In lightweight implementations, the primary targets are typically minimum resource utilization and minimum power consumption, assuming that the execution time does not exceed a predefined maximum. Another way of formulating the goal is to achieve minimum execution time, assuming a given maximum budget in terms of resource utilization, power consumption, or energy usage. The maximum budget on resource utilization is related to the cost of implementation; the budget on power assures correct operation without overheating or devoting additional resources to cooling. The maximum energy usage affects how long a battery-operated device can function before the next battery recharge. In the context of the standardization process for cryptographic algorithms, the mentioned above maximum budgets are very hard to select. Any change in these thresholds may favor a different subset of candidates. With new standards remaining in use for decades, the timing, cost, and power requirements of new and emerging applications are very challenging to predict. Additionally, lightweight hardware implementations can outperform only software implementations targeting specific low-cost, low-power embedded processors, such as Cortex-M4. As a result, in this paper, we focus on developing, benchmarking, and ranking high-speed implementations.

Design Methodology. Hardware design methodologies are developed by the industry over the period of decades. The Register-Transfer Level (RTL) methodology is the most popular design methodology codified by academic textbooks and supported by most industry-grade computer-aided design tools. This methodology assumes designing/coding at a level that is manageable for humans and easy for tools to turn into efficient hardware. The entire system is divided into the Datapath and Controller. The Datapath is described using a hierarchical block diagram using medium-scale components (e.g., adders, multipliers, multiplexers, registers, and memories). The Controller is described using hierarchical algorithmic state machine (ASM) charts or state diagrams. Indirectly, the designer specifies what happens in the circuit in every clock cycle. Thus, the latency (the execution time of a particular major operation) in clock cycles is an inherent feature of the design. The tools determine the maximum clock frequency at which the circuit can operate and the amount of hardware resources used.

Any other approaches to hardware design are often mistrusted. In some cases, justifiably so. For example, recent attempts at replacing RTL with High-Level Synthesis resulted in PQC designs 2-4 orders of magnitude less efficient [13, 21]. Similarly, the use of the software/hardware co-design for PQC led to inconclusive results disregarded by NIST at the end of Round 2 [21, 2]. Therefore, the development of hardware implementations described in this paper follows the traditional RTL methodology.

Our Contributions. The main contributions of this paper are summarized below:

1. we have proposed, documented, and designed the first complete hardware implementations of two variants of NTRU (NTRU-HRSS and NTRU-HPS), as defined in the submissions to Rounds 2 and 3 of the NIST PQC standardization process

2. we have developed a new hardware implementation of CRYSTALS-KYBER outperforming the best previous design in terms of latency, number of operations per second, and the product of latency × #LUTs.

3. we have developed four new implementations of Saber. For the security level 3, two of them outperform the best previous design in terms of resource utilization. The

Table 1: Reported Hardware and Software/Hardware Implementations of KEMs qualified to Round 3

Algorithms	Hardware	Software/Hardware
Lattice-based		
CRYSTALS-KYBER	[13]^H, [49], [95]	[9], [10]*, [33], [3], [94]
FrodoKEM	[47], [13]^H,	[22], [9], [10]*
NTRU	[13]^H	–
NTRU Prime	[66], [76]	–
Saber	[13]^H, [87], [98]	[22], [68], [33]
Isogeny-based		
SIKE	[59], [26]	[67]
Code-based		
BIKE	[48], [79], [82], [81]	–
Classic McEliece	[92], [91], [13]^H	–
HQC	[83]^H	–

^{H} design developed using the High-Level Synthesis (HLS) approach
* extended version of [9]

other two have higher resource utilization but are significantly faster.

4. we benchmarked all mentioned above designs using two FPGA families and compared them with the earlier reported designs for all remaining Round 3 KEMs in terms of latency, the number of operations per second, and the number of LUTs.

All new designs reported in this paper are fully reproducible, and their source code will be released as open-source after the acceptance of this paper to a journal or a conference with proceedings.

2 Previous Work

Hardware and software/hardware implementations of all KEMs qualified to Round 3 of the NIST PQC Standardization Process are summarized in Table 1. The PQC candidates are grouped by family.

HLS-based implementations are distinguished with the superscript^H. These implementations were reported in only one paper [13]. They have been shown to give substantially different results than implementations developed using traditional Register-Transfer Level (RTL) methodology, in which HDL code is developed manually. Therefore, in this paper, we focus on implementations in which a hardware part of the design was developed using traditional RTL methodology. NTRU (as specified in Rounds 2 and 3 of the NIST process) and HQC are the only candidates with no RTL implementations reported to date.

In particular relevance to this paper, we are unaware of any hardware implementation of either NTRU-HPS or NTRU-HRSS as defined during Rounds 2 and 3 of the PQC standardization process. Earlier versions of NTRU were significantly different. Therefore, all major building blocks, top-level block diagram, scheduling scheme, and the corresponding control unit had to be designed from scratch. Nevertheless, we would like to acknowledge earlier work on the implementation of NTRUEncrypt Short Vector Encryption Scheme (SVES), as defined in the IEEE 1363.1 Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices [51]. The most complete high-speed constant-time hardware implementation of this scheme is reported in [29]. Some of the

major differences include the fact that NTRUEncrypt SVES is an encryption scheme rather than KEM. The underlying hash function is SHA-256 rather than SHA3-256. NTRUEncrypt SVES has a non-zero decryption failure rate. Decryption does not require polynomial multiplication in which both operands have only "large" coefficients. There is no notion of packing, unpacking, or lifting. All major parameters are substantially different. Additionally, the implementation reported in [29] does not support key generation in hardware. Another implementation of NTRUEncrypt SVES, reported in [62, 63], supports only encryption. All earlier hardware implementations of NTRU, such as those reported in [7, 74, 54, 5, 53], concerned variants that had even more differences as compared to the most recent specifications of NTRU-HPS and NTRU-HRSS.

The most similar hardware implementation of CRYSTALS-Kyber is described in [95]. Our project started before [95] was published. Major design decisions were made in 2020 based only on earlier available literature. These decisions differed in many aspects from those described in [95]. Some of the most important decisions included the use of k NTT multipliers vs. one used in [95]. Additionally, our design uses a different modular reduction unit, a much faster Keccak module, efficient NTT memory access, faster and smaller encoding and decoding units, and a more efficient rejection sampler. Unlike [95], our design is also technology independent by not employing any vendor-specific IPs. The detailed list of differences is provided in Section 5.2.7. The second pure hardware implementation of CRYSTALS-Kyber is reported in [49]. This implementation supports only encapsulation and decapsulation and is about an order of magnitude less efficient than the one reported in [95]. Earlier implementations of Kyber were of the software/hardware type, and many of them concerned a substantially different Round 1 version of this candidate.

The most similar hardware implementations of Saber are described in [87, 98]. Both designs follow an unified architecture approach that supports selecting parameter sets at run time. [87] employed a schoolbook-based multiplier meanwhile [98] proposed a hierarchical 8-level Karatsuba multiplier. [12] later improves area consumption of the high-speed multiplier in [87] and introduces a new lightweight architecture.

Major types of polynomial multipliers used in the hardware implementations of lattice-based PQC schemes include Schoolbook, Karatsuba [56], Toom-Cook [89, 16], and NTT-based. NTT-based multipliers have been particularly popular in the last decade, and their various architectures and optimized implementations were reported in various publications, such as [77, 84, 24, 80, 75, 61, 96, 31, 97, 93]. Karatsuba and Toom-Cook multipliers have been used for the implementation of Saber. In particular, [98] used Karatsuba, and [68] explored the use of Toom-Cook. Schoolbook multipliers have been used for years, in particular in [77, 27, 64, 87], and many others.

3 Background

3.1 Basic Features of Compared Algorithms

Selected features of all implemented KEMs are summarized in Table 2. All three KEMs are based on the underlying IND-CPA public-key encryption (PKE) schemes. In CRYSTALS-Kyber and Saber, the conversions to the corresponding IND-CCA KEMs are performed using very similar variants of the Fujisaki–Okamoto (FO) transform [34], [41]. NTRU uses a generic transformation from a deterministic public-key encryption scheme to construct a KEM. The NTRU KEM transformation provides IND-CCA2 security with a tight reduction to the well-studied OW-CPA (one-way CPA) security of the NTRU PKE [85]. The only KEMs with no Decryption Failure in the underlying PKE are NTRU-based KEMs, NTRU-HPS, and NTRU-HRSS. Consequently, these schemes require no re-encryption during decapsulation.

In all of these KEMs, the elementary operation is multiplication mod q. In Saber,

Table 2: Features of lattice-based NIST Round 3 finalists in the category of KEMs

Feature	**CRYSTALS-Kyber**	**Saber**	**NTRU-HPS**	**NTRU-HRSS**
Underlying problem	Mod-LWE: Module Learning with Errors	Mod-LWR: Module Learning with Rounding	SVP: Shortest Vector Problem	SVP: Shortest Vector Problem
Sampling	Integers are sampled from a centered binomial distribution (CBD)	Integers are sampled from a centered binomial distribution (CBD)	Fixed-weight and variable -weight polynomials are sampled from a uniform distribution	Variable-weight polynomials are sampled from a uniform distribution
Degree n	Power of 2	Power of 2	Prime	Prime
Modulus q	Prime	Power of 2	Power of 2 with $q/8-2 \leq 2n/3$	Power of 2 with $q > 8\sqrt{2}(n+1)$
Other major parameters	k: number of polynomials per vector, η: parameter of CBD	p, T: other moduli, l: number of polynomials per vector, μ: parameter of CBD	d: Fixed weight for g and m $\mathsf{Lift}(m)$: Identity $m \mapsto m$	f, g: Non-negative correlation $\mathsf{Lift}(m)$: $m \mapsto \Phi_1 \cdot S_3(m/\Phi_1)$
Hash-based functions	SHA3-256, SHA3-512, SHAKE128, SHAKE256	SHA3-256, SHA3-512, SHAKE128	SHA3-256	SHA3-256
Decryption failures	Yes	Yes	No	No
Polynomial Rings	$\mathbb{Z}_q[x]/(x^n+1)$	$\mathbb{Z}_q[x]/(x^n+1)$	R/q: $Z_q[x]/(x^n-1)$ S/q: $Z_q[x]/(\Phi_n)^*$ S/3: $Z_3[x]/(\Phi_n)^*$	R/q: $Z_q[x]/(x^n-1)$ S/3: $Z_3[x]/(\Phi_n)^*$
#Polynomial Multiplications in Encapsulation	k^2+k	l^2+l	1 in R/q	1 in R/q
#Polynomial Multiplications in Decapsulation	k^2+2k	l^2+2l	1 in R/q 1 in S/q 1 in S/3	1 in R/q 1 in S/q 1 in S/3

* $\Phi_n = (x^n-1)/(x-1)$ irreducible in $Z_q[x]$

NTRU-HPS, and NTRU-HRSS, q is a power of two, significantly simplifying the reduction mod q. In Kyber, q is a special prime, selected in such a way to support speeding up polynomial multiplication in $\mathbb{Z}_q[x]/(x^n+1)$ using the Number Theoretic Transform (NTT).

All four algorithms use SHA3-256. Saber additionally employs SHA3-512 and SHAKE128. Kyber requires the same set of hash-based algorithms as Saber, extended with SHAKE256. NTRU-based KEMs use sampling from the uniform distribution. In Kyber and Saber, a Centered Binomial Distribution (CBD) is employed.

There are two variants of NTRU described in the specification, the NTRU-HPS based on Hoffstein, Pipher, and Silverman's NTRU encryption scheme [40] and NTRU-HRSS introduced by Hülsing, Rijnveld, Schanck, and Schwabe in [50]. The NTRU-HPS parameter sets follow the approach to use fixed-weight sample spaces and allow several choices of modulus q for each degree n. Meanwhile, the NTRU-HRSS allows arbitrary-weight sample spaces but restricts q as a function of n.

In Kyber and Saber, the most time-consuming operations are matrix-by-vector and vector-by-vector multiplications, where each element of a matrix or a vector is a polynomial with n coefficients in Z_q, and the multiplication of such polynomials is performed modulo the reduction polynomial x^n+1. In the NTRU-based KEMs, the most time-consuming operation is polynomial multiplication in the rings specified in Table 2.

Parameter sets of three investigated candidates are summarized in Table 3. The specification of NTRU associates two different security categories with each parameter set of NTRU-HPS and NTRU-HRSS. In this paper, we conservatively assumed the lower security category based on the so-called non-local computational models (see [**21**], Section 5.3 Security Categories). The same computation model is implicitly assumed by the submitters of the other investigated algorithms. We implemented three parameter sets of NTRU-HPS and NTRU-HRSS, which are ntruhrss701, ntruhps2048677, and ntruhps4096821, corresponding to security levels 1, 1, and 3, respectively in non-local models of computation.

Table 3: Parameter sets of investigated algorithms. Notation: Sk - Secret Key, Pk - Public key, Ct - Ciphertext.

Algorithm	Parameter Set	Security Level	Degree n	Modulus q	Sk Size [bytes]	Pk Size [bytes]	Ct Size [bytes]
Kyber	Kyber512	1	256	3329	1,632	800	768
NTRU-HPS	ntruhps2048677	1*	677	2^{11}	1,235	931	931
NTRU-HRSS	ntruhrss701	1*	701	2^{13}	1,452	1,138	1,138
Saber	LightSaber-KEM	1	256	2^{13}	1,568	672	736
Kyber	Kyber768	3	256	3329	2,400	1,184	1,088
NTRU-HPS	ntruhps4096821	3*	821	2^{12}	1,592	1,230	1,230
Saber	Saber-KEM	3	256	2^{13}	2,304	992	1,088
Kyber	Kyber1024	5	256	3329	3,168	1,568	1,568
Saber	FireSaber-KEM	5	256	2^{13}	3,040	1,312	1,472

* assuming non-local computational models

3.2 Short Introductions to Compared Algorithms

3.2.1 NTRU

Definitions and Parameters. Φ_1 is $(x-1)$. Φ_n is $(x^n-1)/\Phi_1 = x^{n-1}+x^{n-2}+\ldots+x+1$. From the implementation point of view, all operations in NTRU are polynomial operations over the quotient rings R_q, S_q and S_p where $R_q : \mathbb{Z}_q[x]/\Phi_1\Phi_n$, $S_q : \mathbb{Z}_q[x]/\Phi_n$, and $S_p : \mathbb{Z}_p[x]/\Phi_n$. Parameter p is fixed to 3 in all parameter sets of NTRU. Thus, polynomials in S_p are in ternary form, i.e., have their coefficients in $\{-1, 0, 1\}$. In this paper, for NTRU, we use the notation S_p and S_3 interchangeably. Coefficients of polynomials in R_q and S_q have bit-widths of $\epsilon_q = log_2 q$ and those of polynomials in S_p have bit-widths of $\epsilon_p = \lceil log_2 p \rceil$.

In NTRU-HRSS, polynomial f, which is a part of the secret key, is required to have non-negative correlation property, $\sum_i f_i f_{i+1} \geq 0$. In NTRU-HPS, polynomial m in S_p has the fixed-weight property, consisting of $d/2$ coefficients equal to 1 and $d/2$ coefficients equal to -1, with $d = q/8 - 2$. Having the fixed-weight property of m ensures that the ciphertext $c \equiv 0 \pmod{(q, \Phi_1)}$ in NTRU-HPS. In NTRU-HRSS, in order to achieve the same property of c, m is lifted from S_3 to R_q by the map $m \mapsto \Phi_1 \cdot S_3(m/\Phi_1)$.

Pseudocode. The key generation, encryption and decryption of the PKE scheme of NTRU are shown in Algorithms 4, 6 and 7 in Appendix A, respectively [18]. The IND-CCA2 NTRU KEM in Algorithms 5, 8 and 9 in Appendix A, is based on the Saito-Xagawa-Yamakawa variant of the NTRU-HRSS KEM, with improvements that eliminate re-encryption during decapsulation. In the reference implementation of NTRU, the Sample function performs ternary sampling on random input, which requires kilobytes of random data per each operation of key generation or encapsulation. We chose to deviate from the reference implementation by using only 32-byte random input data and expanding it using SHAKE128. Sample generates polynomials in ternary form, which may have either an arbitrary or a fixed weight and/or non-negative correlation property.

During key generation, two polynomial inversions are performed in $S_3 \pmod{(3, \Phi_n)}$ and $S_q \pmod{(q, \Phi_n)}$. All coefficients of polynomials modulo q or p are packed together by unpack_ϵ_q and unpack_ϵ_p. Thus, they must be unpacked before being used in any operation. The Lift function lifts polynomial in S_3 to R_q. The most critical operation is polynomial multiplication in $R_q \pmod{(q, (x^n - 1))}$. Other multiplication operations in S_3 or S_q can be performed by doing multiplication in R_q, followed by modulo $(3, \Phi_n)$ or (q, Φ_n), respectively. During decryption, the ciphertext c is checked to determine if $c \equiv 0 \pmod{(q, \Phi_1)}$. As described in the specification [18], if c is unpacked by unpack_ϵ_q, we only need to check whether the unused bits of the final byte of c are all zeros. r and m are also needed to be checked if they are in the plaintext space, which means their coefficients

are in the ternary form, and for NTRU-HPS, m must have the correct fixed weight.

3.2.2 CRYSTALS-Kyber

Polynomial Multiplication. A basic operation of CRYSTALS-Kyber is the multiplication of two polynomials. In Kyber the polynomials are elements of $R_q = \mathbb{Z}_q[X]/\langle X^n + 1\rangle$.

Thus, for all security levels, polynomials are of the same degree $n = 256$, and their coefficients are members of the base prime field $\mathbb{Z}_q$, where $q = 3329$. However, a different number of polynomials is required for each security level. These polynomials are treated as a vector. The size of this vector is specified using the parameter k. k is 2, 3, and 4 for security levels 1, 3, and 5, respectively. Secret noise polynomials are sampled from a Centered Binomial Distribution (CBD), where η is either 2 or 3.

An efficient method for polynomial multiplication in R_q is through the use of the Number-Theoretic Transform (NTT) [25] which is a generalization of the Discrete Fourier Transform (DFT) to the finite ring $\mathbb{Z}_q$. In Rounds 2 and 3 of the NIST PQC standardization process, Kyber uses $n = 256$ and $q = 3329 = 13 \cdot 2^8 + 1$ where $2n \nmid q - 1 = 13 \cdot 2^8$. To make efficient NTT multiplication possible, a new definition of NTT was provided, which transforms a polynomial of degree 256 to a polynomial of degree 128 made up of degree one polynomials as its coefficients.

$$\hat{f}_k = f \bmod (X^2 - \zeta^{(2k+1)}) \quad (1)$$

where ζ=17 is the first primitive 256-th root of unity modulo q.

In other words $\hat{f}$ consists of 128 polynomials of degree one:

$$\hat{f}_k = f \bmod (X^2 - \zeta^{(2k+1)}) = \hat{\hat{f}}_{2k} + \hat{\hat{f}}_{2k+1} X \quad (2)$$

The sequence of 128 coefficient pairs of degree 1 polynomials can be viewed as a polynomial of degree 256, and then the NTT transform can be expressed separately for the odd and even coefficients:

Point-wise multiplication consists of 128 basic products $\hat{f} \cdot \hat{g} \bmod X^2 - \zeta^{(2i+1)}$:

$$\begin{aligned}\hat{\hat{h}}_{2i} + \hat{\hat{h}}_{2i+1} X &= (\hat{\hat{f}}_{2i} + \hat{\hat{f}}_{2i+1} X)(\hat{\hat{g}}_{2i} + \hat{\hat{g}}_{2i+1} X) \\ &= \left(\hat{\hat{f}}_{2i}\hat{\hat{g}}_{2i} + \zeta^{(2i+1)}\hat{\hat{f}}_{2i+1}\hat{\hat{g}}_{2i+1}\right) + \left(\hat{\hat{f}}_{2i}\hat{\hat{g}}_{2i+1} + \hat{\hat{f}}_{2i+1}\hat{\hat{g}}_{2i}\right) X \end{aligned} \quad (3)$$

Pseudocode. Pseudocode of the Kyber CPAPKE Key Generation, Encryption, and Decryption are given in algorithms 10, 11, and 12 in Appendix B, respectively. Kyber CCA KEM schemes are built upon the CPAPKE operations, multiple hashing operations, and the FO transformation to achieve the IND-CCA2 security. The detailed algorithms of the Kyber CCAKEM Key Generation, Encapsulation, and Decapsulation are shown in algorithms 13, 14 and 15 in Appendix B.

3.2.3 Saber

A distinctive feature of Saber is that rounding operations are used to avoid the noise addition step and reduce the amount of randomness required. Additionally, by using only moduli that are powers of 2, modular reduction does not require any hardware resources and rejection sampling is eliminated.

Definitions and Parameters. Saber involves operations on matrices and vectors of polynomials over the quotient rings $R_q : \mathbb{Z}_q[x]/(x^n + 1)$ with fixed $n = 256$. Polynomials in Saber are sampled from the uniform distribution or centered binomial distribution. β_μ denotes a centered binomial distribution with the parameter μ and the values of samples in the range $[-\mu/2; \mu/2]$. The module dimension l defines the size of vectors and matrices

Table 4: Features of algorithms and parameter sets affecting the choice of a multiplier type

	Small Coefficient Range	Number of coefficients	NTT-friendly ring	One operand in NTT domain	Encapsulation	Decapsulation	
					Number of "Small" × "Large" Polynomial Multiplications	Number of "Large" × "Large" Polynomial Multiplications	Number of "Small" × "Large" Polynomials Multiplications
Kyber512	[-3..3], [-2..2]	256	Y	Y	6	—	8
Kyber768	[-2..2]				12	—	15
Kyber1024	[-2..2]				20	—	24
ntruhrss701	[-1..1]	701	N	N	1	1	3
ntruhps2048677	[-1..1]	677			1	1	3
ntruhps4096821	[-1..1]	821			1	1	3
LightSaber-KEM	[-5..5]	256	N	N	6	—	8
Saber-KEM	[-4..4]				12	—	15
FireSaber-KEM	[-3..3]				20	—	24

of polynomials as $l \times 1$ and $l \times l$, respectively. We denote $R_q^{l \times l}$ and $R_q^{l \times 1}$ as a matrix and vector of polynomials in R_q. The rounding operation includes coefficient-wise addition with a constant factor and is followed by bit shifting.

Pseudocode. The pseudocode of Saber is shown as Algorithms 16, 17, 18, 20, 19, and 21 in Appendix C . The KEM key generation includes sampling uniformly random matrix A using SHAKE128. Secret vector s is sampled in binomial distribution from the uniformly random output from SHAKE128. The vector product of $A^T \cdot s$ is rounded and serves as a public vector b in the public key. The secret key includes the public key, hash of the public key, secret vector s, and a pseudo-random byte string z, which is used for implicit rejection in FO transform.

Encapsulation includes encryption with additional hashing. A "small" vector s is generated using sampling from the centered binomial distribution. The ciphertext has two parts. The first part has the rounded product of $A \cdot s'$. The second one includes the sum of the inner product of b, s', and the encoded message m. We adopt the optimization in [98] to compute $b^T \cdot s'$ before $A \cdot s'$. Since the generation of s' and A requires the same SHAKE128 function, we would need to finish generating s' before performing $A \cdot s'$ with the on-the-fly generation of A. The multiplication of b^T and s' can be performed in parallel with the sampling of s'. The shared secret is derived from the hashes of the public key, message, and ciphertext. Decapsulation involves decryption and re-encryption. During decryption, the secret key is used to compute v, which is used to extract the message. The obtained message is then re-encrypted to check whether the re-encrypted ciphertext is the same as the received one. To save bandwidth, all coefficients of polynomials modulo q or rounded to p or T are packed together by $\mathsf{pack_\epsilon_q}$, $\mathsf{pack_\epsilon_p}$ or $\mathsf{pack_\epsilon_T}$. Thus, they must be unpacked before being used in any operation.

3.3 Choice of a Multiplier Type

Major features of investigated algorithms and their parameter sets affecting the choice of a multiplier type are summarized in Table 4.

All four candidates involve multiplication of a polynomial with so-called "small" coefficients, belonging to the range listed in the second column of Table 4, by a polynomial with "large" coefficients, in the range [0..q-1], where q is given in Table 3. Out of four major multiplier types introduced at the end of Section 2, only the Schoolbook multiplier can take full advantage of the feature that "small" coefficients have significantly fewer bits than "large" coefficients. This multiplier has a very regular structure and, in each clock cycle, allows the multiplication of u coefficients of one operand by all coefficients of the second operand. The parameter u is an unrolling factor, typically set to 1, 2, 4, etc. Consequently, the execution time of this multiplier is approximately equal to n/u clock cycles, and its area in LUTs is proportional to n. Since "small" coefficients have from 2 to 4 bits, multiplication by them can be accomplished using ANDs, shifts, and additions. These operations can be efficiently implemented using LUTs and a special fast carry logic of Xilinx FPGAs,

without the need for DSP units. Consequently, all polynomial multiplications in Saber and all-but-one multiplications in NTRU can be efficiently implemented using the Schoolbook multiplier. The disadvantage is a relatively large area, even for the smallest value of the unrolling factor u=1.

An NTT-based multiplier has much smaller area, independent of n, and the execution time proportional to $n \cdot lg_2(n)$. They can be sped up using a small number of DSP units. As a result, it is practical to instantiate several such multipliers within the same design without reaching the area threshold. The improvement in execution time depends on data dependencies and the relative speed of units producing inputs to the multipliers. An additional speed-up can be accomplished by defining and/or storing some inputs in the NTT domain. This way, the conversion from the regular to NTT domain may be skipped for one operand. Among the investigated algorithms, only Kyber is defined this way. NTT-based multipliers do not offer any advantage in terms of execution time for the case when one operand has small coefficients. They also impose specific requirements on the dependence between the number of coefficients in each operand, n and the modulus q. If these dependencies do not hold, NTT may still be possible, but it requires extra computations, increasing the multiplier's area and possibly complicating control. Taking all these features into account, an NTT-based multiplier is an obvious choice only for CRYSTALS-Kyber.

As shown in Table 4, NTRU-HPS and NTRU-HRSS are the only investigated candidates that require multiplying two polynomials with "large" coefficients. Values of n and q do not fulfill the requirements of NTT. None of the operands is stored in the NTT domain. As a result, the use of the Toom-Cook multiplier appears to be the best choice. These multipliers have an area smaller than the Schoolbook and larger than NTT types. They can be sped up using a relatively moderate number of DSP units. Consequently, they appear to be the natural choice for the implementation of the "large" by "large" polynomial multiplication in the decapsulation operation of NTRU-HPS and NTRU-HRSS.

4 Design and Benchmarking Methodology

As stated in the Introduction, we follow the best understood and commonly trusted RTL methodology. The designers of each implementation worked very closely with each other to ensure a consistent approach to all optimizations. Our designs started when no pure hardware implementations of CRYSTALS-Kyber, NTRU, or Saber were reported in the literature yet. All major designs decisions were made independently of those made in related concurrent projects described in [95], [94], [87], and [98]. All code was developed from scratch without using any library components or any parts of other groups' designs. Consequently, our designs are fully portable, well-documented, and easy to improve and maintain.

All modules common for multiple algorithms, such as the SHA-3/SHAKE unit, were reused. The designs for NTRU and Saber are encoded using VHDL. The design for CRYSTALS-Kyber is encoded using Chisel [6]. We believe that in the RTL methodology, the choice of a hardware description language has a negligible effect on the obtained results. Functional verification of the hardware description language (HDL) code has been performed by comparing simulation results with precomputed outputs generated by a reference software implementation.

On top of this well-known and trusted design methodology, we define a quite straightforward benchmarking methodology. The primary goal is fairness, not a novelty.

All our hardware implementations assume the use of the FIFO interface defined in [27]. This interface is similar to the interface of the AXI4-Stream Protocol [4].

In terms of functionality of designed units, several options are possible: 1) separate units for encapsulation, decapsulation, and key generation; 2) one unit supporting encapsulation, decapsulation, and key generation, with resource sharing; 3) one unit supporting

encapsulation and decapsulation and the second unit responsible for key generation; 4) one unit (on the server-side) supporting key generation and decapsulation, and the second unit (on the client-side) supporting encapsulation. None of these assumptions meet the requirements of all applications. In this paper, we assume Scenario 1). However, whenever possible, we also report results for Scenario 2).

Similarly, there are two major assumptions regarding support for multiple parameter sets: 1. choice among parameters sets at the time of synthesis; 2. choice among parameters sets at run time. The advantage of Approach 1) is the ability to determine the minimum possible resource utilization separately for each security level. Approach 2) demonstrates the flexibility of hardware implementation. However, it will likely require a larger amount of resources than the implementation supporting the highest security level. In this paper, we adopted Approach 1.

One public key and one secret key are assumed to be loaded to the hardware unit before the start of encapsulation and decapsulation, respectively. Thus, the latency of these operations does not include the time required to load the respective keys.

The primary design goal is speed. The speed is characterized using two primary metrics: a) the minimum latency in time units and b) the maximum number of operations per second. These two metrics are related. However, any particular application may have independent requirements in terms of their values. For example, real-time applications, such as secure communication between two autonomous vehicles, may have very strict requirements regarding the time required to establish secure communication and thus the total time required for encapsulation and decapsulation. At the same time, the required number of operations per second may be very small and thus not limiting. On the other hand, a high-traffic server may have to handle thousands of secret key establishments per second. Simultaneously, the time allowed for each individual transaction (and thus the latency of decapsulation) may be quite large.

Taking into account that specific thresholds depend strongly on an application and the state of technology, no specific values are assumed in this benchmarking effort. Instead, we assume that both decreasing latency and increasing the number of operations per second are worthy goals as they will broaden the range of applications that can use a new PQC standard at a given stage of technology. For simplicity, we assume, in agreement with most of the literature, that each design processes only one set of inputs (keys, ciphertexts, random bits) at a time. As a result, the number of operations per second becomes a direct inverse of latency in time units. One, however, should keep in mind an important difference between them: duplicating a design doubles the number of operations per second, but it does not change the latency.

When choosing between multiple potential solutions during the design-space exploration, we give priority to the designs that minimize latency and thus maximize the number of operations per second. However, the parallelization is pursued only until it gives a substantial gain in speed as compared to the area increase in LUTs. Considering that speed and area can be traded one for another, we perform space exploration, using the best available approximations of the execution time in clock cycles and resource utilization in LUTs, DSP units, and BRAMs. Afterward, we choose to include in the speed rankings the implementations of each candidate that are the closest to each other in terms of resource utilization.

For our target platforms, we chose representative devices of two different FPGA / FPGA SoC families: Artix-7 and Zynq UltraScale+. Specifically, we choose the largest devices of both families supported by free versions of Xilinx tools. For each device, we assume that its highest speed grade is used. These assumptions led us to choosing a) Artix-7 XC7A200T-3, with 134,600 LUTs, 365 BRAMs, and 740 DSP units, and Zynq UltraScale+ ZU7EV-3, with 230,400 LUTs, 312 BRAMs, 96 Ultra BRAMs, and 1,728 DSP units. Based on the previous work, summarized in Section 2, these devices are sufficient

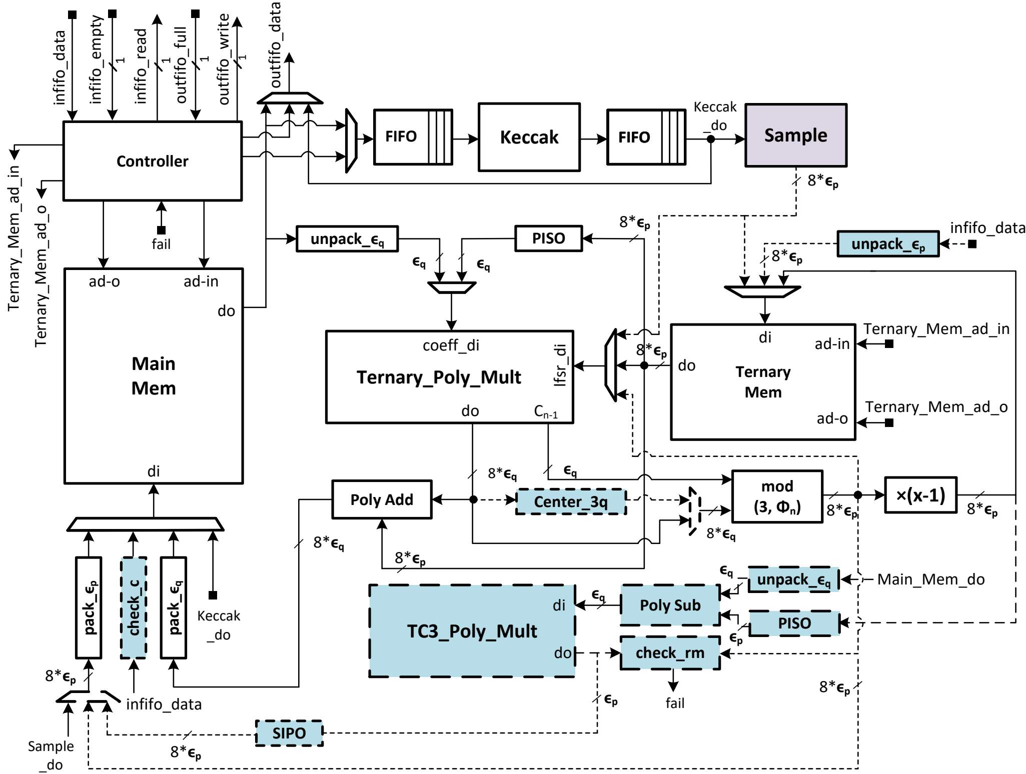

Figure 3: Top-level block diagrams of the Encapsulation and Decapsulation modules of NTRU. The purple, blue modules are used only in Encapsulation and Decapsulation, respectively.

for a vast majority of designs reported to date. Out of their resources, the number of LUTs is the most limiting. The use of BRAMs and DSP units is typically negligible. Therefore, for the purpose of design-space exploration, we use the number of LUTs as a measure of the circuit *Area*. The maximum clock frequency is determined using binary search. Only final results obtained after placing and routing are reported.

5 Hardware Designs

5.1 NTRU

The top-level diagram of NTRU is shown in Fig. 3. The scheduling of major operations during encapsulation, decapsulation, and key generation is illustrated in Figs 4, 5, and 6, respectively.

5.1.1 Ternary Sampling

For NTRU-HRSS, the generation of f and g is performed in S_3 during key generation. Random bytes coming from SHAKE128 are reduced modulo 3 to obtain the ternary coefficients stored in a first-in, first-out (FIFO) unit. The sum of products of consecutive coefficients $s = \sum_i f_i f_{i+1}$ is computed at the same time. After finishing generating all coefficients, if $s < 0$, coefficients at even indices are signed-flipped before being transferred to the next computational stage. Thus, the non-negative correlation properties of f and g are satisfied. g is later multiplied by $x - 1$, which can be carried out trivially during

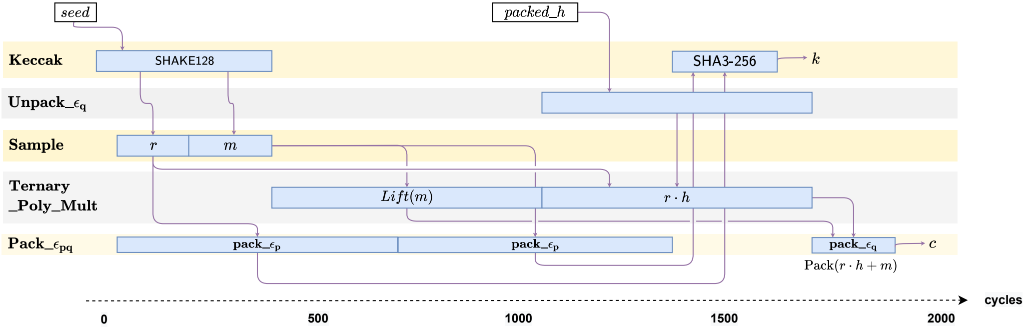

Figure 4: Operations Scheduling for Encapsulation of NTRU-HRSS.

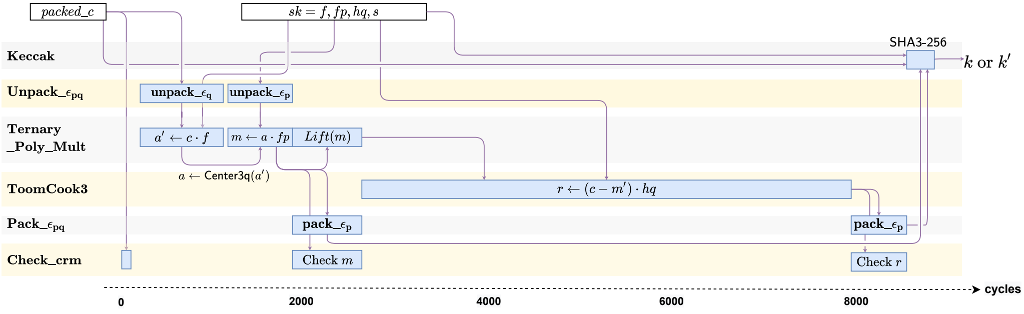

Figure 5: Operations Scheduling for Decapsulation of NTRU-HRSS.

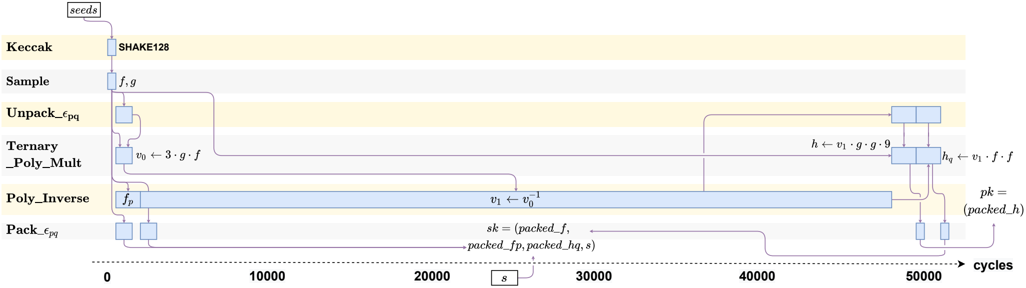

Figure 6: Operations Scheduling for Key Generation of NTRU-HRSS.

the transfer. During encryption, r and m do not have either the non-negative correlation property or fixed-weight. They can be computed by simply reducing random data modulo 3.

For NTRU-HPS, f and r have arbitrary weight and can be sampled in a straightforward manner. However, m and g have fixed weight and are sampled by creating a random permutation of a list with a fixed number of values $-1, 0$ and 1. One can simply perform Fisher-Yates shuffle to have a random non-biased permutation of such a list. However, Fisher-Yates shuffle is not constant-time and creates a risk of potential timing attacks. Given that, we adopt a constant-time merge sorting approach for the permutation. The merge-sort module requires n random elements. Each element includes 30 random bits concatenated with "01" for the first $w/2$ elements, "10" for the next $d/2$ elements, and "00" for the rest. To get a 30-bit block, a 64-bit input is passed through a PISO, to be divided into two 32-bit blocks. Each 32-bit block is then processed using a buffer register and a

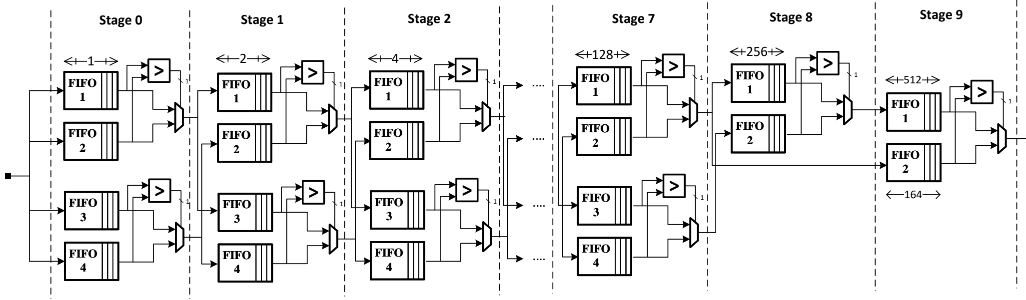

Figure 7: FIFO-based merge sort module for NTRUHPS2048677.

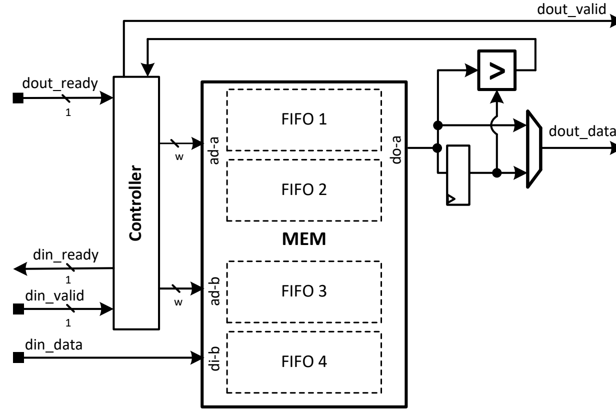

Figure 8: One stage of a FIFO-based merge sort module implemented using dual-port memory.

variable shifter to get a 30-bit block. The leftover bits are stored in the buffer register to be concatenated with the subsequent output of PISO. After sorting, the upper 30 bits are discarded, and the lower 2 bits are converted from $\{0, 1, 2\}$ to $\{0, 1, -1\}$.

Related works: Wang et al. [91] proposed a fully pipelined constant-time merge sort module to generate random permutation in the Key Generation operation of Classic McEliece. To sort a random list of n elements, the module needs $log_2(n)$ iterations, where each step requires $O(n)$ comparison operations. Therefore, the total cycle count is approximately equal to $nlog_2(n)$ cycles. Marotzke [66] implemented an iterative Batcher's merge exchange sort module for a very similar sampling function in the Streamlined NTRU Prime. Its operation also have asymptotic complexity of $O(nlog_2(n))$.

To speed up this operation, we use a merge-sort module consisting of $log_2(n)$ cascaded Sort Stages to sort the random sequences. The FIFO-based merge-sort module for NTRU-HPS677 is shown in Fig. 7. The inputs to each Sort Stage are two sorted lists, and the output is a sorted list of double input length, including all elements from the two input lists. Each input list is stored in a separate segment of memory. While the lower stages can be implemented by registers, the higher stages are implemented in dual-port memory. This approach can reduce the number of LUTs and FFs used to construct the large FIFO in higher stages at the cost of a small number of BRAMs. The internal structure of a Sort Stage is shown in Fig 8. By making use of the dual-port memory, the controller in each stage can write out the sorted list to the next stage and receive other input lists from the previous stage at the same time. By pipelining the operation of multiple Sort Stages, we can achieve a highly optimized latency for sorting. Our merge-sort module requires n clock cycles for reading n elements, roughly n cycles for sorting, and another n cycles to

Table 5: Implementation Results of the FIFO-based Merge Sort module and comparison with related works.

	Freq.	LUT	FF	BRAM	DSP	Cycles
NTRUPRime: $n = 761$, $w = 32$, Zynq Ultrascale+						
Batcher's Merge Exchange Sort [66]	279	231	87	1.0	0	49,400
FIFO-based Merge Sort	250	1,441	940	3.5	0	2,762
ClassicMcEliece: $n = 8192$, $w = 45$, Zynq Ultrascale+						
4x Pipelined Merge Sort [91]	250	583	411	20.0	0	147,505
FIFO-based Merge Sort	250	2,533	1,589	33.0	0	26,646

write out a sorted sequence. In particular, sampling m or g takes 2,678 and 3,343 cycles for NTRU-HPS677 and NTRU-HPS821, respectively.

The comparison of our FIFO-based merge sort module with previous work is shown in Table 5. We synthesize our module with the parameters used in [66] and [91]. Since the code of [91] is open-source, we can synthesize their merge-sort module targeting the same platform, Zynq Ultrascale+, and obtain results. Our FIFO-based merge sort module outperforms the previous designs by roughly an order of magnitude, excluding the time to load input and unload output. Although the increase in resource utilization is significant, it is still a quite compact design, suitable for high-speed applications that require random constant-time permutation.

5.1.2 Polynomial Multiplication

In all previous work on hardware implementations of NTRU, the polynomial multipliers always exploited the property of small ternary coefficients. The schoolbook multiplication has quadratic-complexity but enables simple, parallel, easy-to-parameterize, and very fast architecture for polynomial multiplication in NTRU. In [68], an efficient architecture based on the Toom-Cook algorithm is proposed in a Software/Hardware codesign platform. Toom-Cook 4-way was applied to divide polynomial multiplication of 256 coefficients into seven multiplications with 64 coefficients. These seven multiplications are run in parallel using seven schoolbook polynomial multipliers.

In the AVX2 implementation of the NTRU submission package [73], a multi-layer Toom-Cook and Karatsuba are used to speed up the multiplication. In the recent work [19], an NTT-based polynomial multiplication is proposed, which outperforms the Toom-Cook method. However, the NTT-based polynomial multiplication was also applied to only multiplication with ternary polynomials. Therefore, it is not applied to speed up the key generation and the final multiplication in decryption, which does not have any input polynomial in ternary form.

Toom-Cook Polynomial Multiplier. In this work, for multiplication without involving ternary polynomial, we implement a Toom-Cook 3-way polynomial multiplier, which splits an n-coefficient polynomial multiplication into five multiplications with $n/3$ coefficients. The five multiplications are performed in parallel using five Odd-Even Karatsuba multipliers. Our improvements over [68] include:

- Our implementation supports splitting input polynomials into three smaller polynomials before the Evaluation step. The Toom-Cook core in [68] relies on software to do this operation.
- Using the Odd-Even Karatsuba method significantly improves the latency of the multiplication step.
- Our core supports Recomposition, which has the output polynomial in the ring R_q.

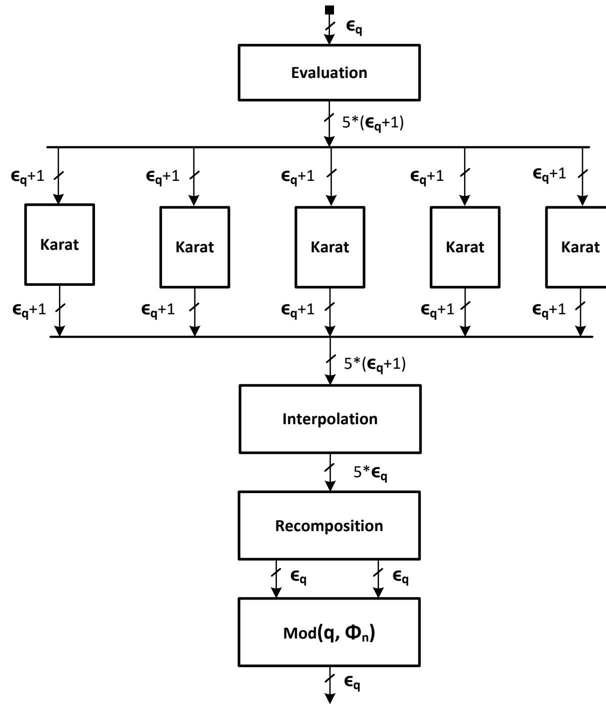

Figure 9: Toom-Cook 3 Polynomial Multiplier w/ Overlap-free Karatsuba.

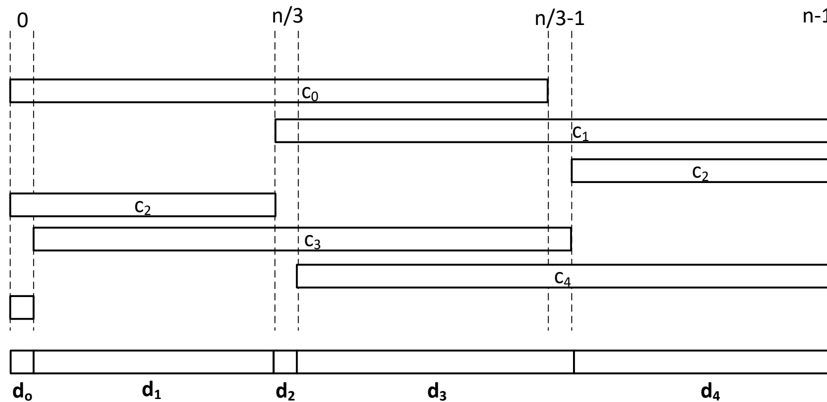

Figure 10: Recomposition in R_q.

In [68], 5 output polynomials are transferred to software and are then recomposed into a single polynomial.

Toom-Cook and Karatsuba are multiplication algorithms that have better asymptotic complexity compared to the schoolbook method. Toom-Cook k-way is a generalization of Karatsuba with $k = 2$. Both algorithms generally follow five steps: splitting, evaluation, pointwise multiplication, interpolation, and recomposition. The input polynomials are split into $2k - 1$ polynomials with n/k coefficients. These polynomials are then evaluated at $2k - 1$ points. The evaluated polynomials are multiplied in the pointwise-multiplication steps. The results are interpolated as an opposite of the evaluation step. The output polynomials of the interpolation step are finally recomposed into the final product.

The top-level diagram of the Toom-Cook 3-way module is shown in Fig. 9. Toom-

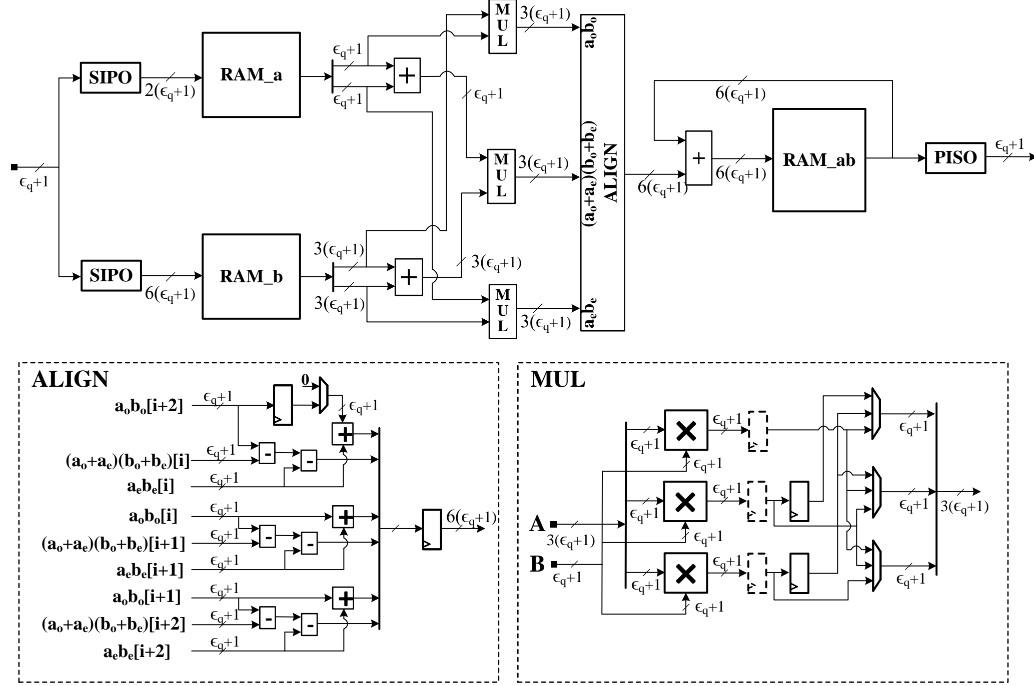

Figure 11: Overlap-free Karatsuba polynomial multiplier.

Cook 3-way splits input polynomial $A(x)$ into three polynomial a_0, a_1 and a_2 such that $A(y) = a_0 + a_1y + a_2y$, where $y = \lceil n/3 \rceil$. a_0, a_1 and a_2 are then evaluated at five points $\{0, 1, -1, 2$ and $\infty\}$. The pointwise multiplications are performed by Odd-Even Karatsuba modules. We adopt the optimal sequence for evaluation and interpolation in the Toom-Cook 3-way from Bodrato et al. [16]. We would like to highlight that during evaluation, there is a division by 2, which becomes a one-bit shift and causes a one-bit loss of precision. Therefore, the pointwise multiplication and interpolation steps require one extra bit for each coefficient.

After interpolation steps, we have 5 output polynomials $c_0, c_1, \ldots c_4$ with $2n/3$ coefficients needed to be recomposed and reduced modulo $x^n - 1$ in the ring R_q. Fig. 10 shows the positions of polynomials $c_0, c_1 \ldots c_4$ in the final product polynomial d modulo $x^n - 1$. Since the recomposition module receives five coefficients with the same index from c_0 to c_4, we need two registers d_0, d_2 and three shift registers of the size $\lfloor n/3 \rfloor - 1$. For example, d_0 will be initialized with the coefficient from c_0 at the cycle 0, then it is added to a coefficient from c_2 in the cycle $\lfloor n/3 \rfloor - 1$ and lastly added with the last coefficient from c_4 in the cycle $\lfloor 2n/3 \rfloor - 1$.

The overlap-free Karatsuba splits input polynomial $A(x)$ into two polynomials a_0 and a_1 such that $A(y) = a_0 + a_1y$ where $y = x$. It means that a_0 consists of all even coefficients of A(x); meanwhile, a_1 consists of all odd coefficients of A(x). The overlap-free Karatsuba scheme enables a more efficient alignment of product coefficients compared to the classic Karatsuba scheme. The diagram of our overlap-free Karatsuba module is shown in Fig. 11. Two polynomials are stored in `RAM_a` and `RAM_b`. The multiplication between two coefficients from `RAM_a` and `RAM_b` would normally cost 12 integer multipliers. However, this number is reduced to 9 multipliers thanks to the Karatsuba algorithm. The latency of this module can be calculated as follows:

$$\text{Pointwise-Multiplication Latency} = \left(\frac{n}{6\times3} + 1\right) \times \left(\frac{n}{6} + 1\right)$$

We note that the splitting and recomposition steps are merged into evaluation and

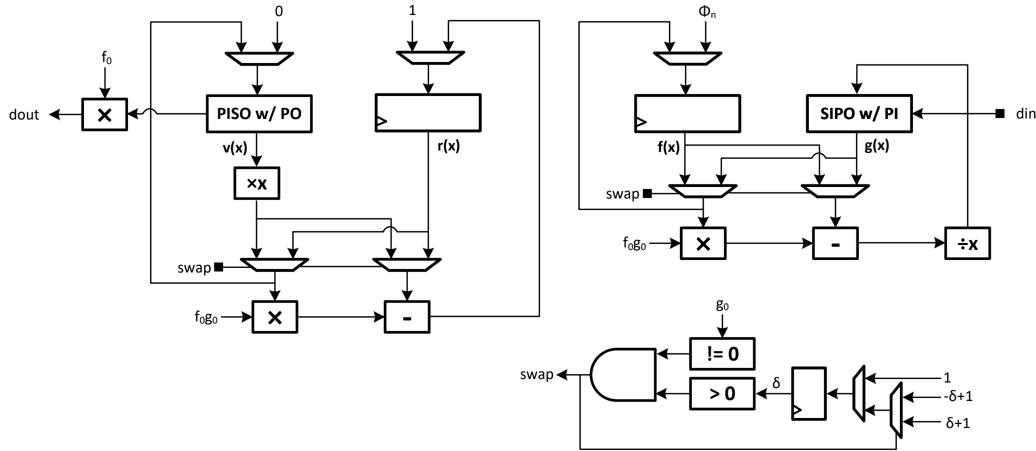

Figure 12: S_2/S_3 Inversion Module

interpolation, respectively. Each splitting/evaluation or recomposition/interpolation takes $\lceil 2n/3 \rceil$ cycles. Our Toom-Cook multiplier finishes one polynomial multiplication in R_q or S_q in 5507, 5098 and 7274 cycles for $n = 701, 677$ and 821, respectively.

Ternary Polynomial Multiplier. For multiplications involving polynomial in the ternary form $\{-1, 0, 1\}$, we use the constant-time LFSR-based polynomial multiplier, proposed in [27], which has the latency of n clock cycles. By loading the ternary polynomial with coefficients in $\{-1, 0, 1\}$ to the LFSR, instead of a polynomial with "big" coefficients, we reduce the number of flip-flops required to realize this LFSR by a factor of four. We also shorten the time required to load a polynomial into the LFSR, since eight 2-bit coefficients can be loaded in a single clock cycle. All integer multiplication-and-accumulation operations between coefficients of two operands and one product polynomials are reduced to addition, pass-through, or subtraction. The LFSR is initialized to a polynomial with ternary coefficients. Let us denote the initial state of this LFSR as $a(x)$. In each subsequent iteration, the output from LFSR contains the value $a(x) \cdot x^i \bmod x^n - 1$. In a single clock cycle, a simple multiplication by x, namely $a(x) \cdot x^{i+1} \bmod x^n - 1 = a(x) \cdot x^i \cdot x \bmod x^n - 1$, is performed.

5.1.3 Inversion in S_3 and R_q

The inverse of polynomials in R_q and S_3 plays an important role in key generation. We need to compute f_p, which is an inverse of f in S_3 for the secret key. Computation of v_1, which is an inverse of v_0 in S_q, must be completed before any later operations could proceed.

Inversion in S_3: Inversion in S_3 is done using the constant-time extended Greatest Common Divisor (GCD) unit proposed in [15]. The top-level diagram of our `S3_inverse` module is shown in Fig. 12. At first, $g(x)$ is initialized with an input polynomial in reverse order. $f(x)$, $r(x)$ and v(x) are initialized with Φ_n, 1 and 0 respectively. The module runs in exactly $2(n-1)$ cycles. All coefficients of four polynomials are updated simultaneously during each iteration according to the value of δ and g_0. All operations, including addition, subtraction, and multiplication, are reduced modulo 3. Multiply and divide by x are performed by simple bit shifting. Lastly, the inverse of input polynomial is $f_0 \times v(x)$. We note that the inverse polynomials are also stored in the reverse order. Our module also supports inversion in S_2, which is used in inversion in S_q. We compare our results for NTRU-HPS821 with $n = 821$ with the Reciprocal in $R/3$ module in the implementation of Streamlined NTRU Prime in [66]. We have shown that the extended GCD can be

Table 6: Implementation results of the Extended GCD module and comparison with related work for Streamlined NTRU Prime in Zynq Ultrascale+ platform.

	Freq.	LUT	FF	BRAM	DSP	Cycles
Extended GCD w/ $n = 761$ [66]	271	518	216	0	0	1,168,899
Extended GCD w/ $n = 821$	250	8,534	5,479	0	0	1,846

Algorithm 1 Polynomial Inversion in S_q [50]

Input: Polynomial a in S_q
Output: Polynomial b in S_q such that $a \cdot b = 1 \bmod (q, \Phi_n)$

1: $v_0 \leftarrow a^{-1} \bmod (2, \Phi_n)$
2: $i \leftarrow 1$
3: **while** $i < logq$ **do**
4: $\quad v_0 \leftarrow v_0 \cdot (2 - a \cdot v_0)$
5: $\quad i \leftarrow 2i$
6: **end while**
7: $b \leftarrow v_0$

Algorithm 2 Lift in NTRU-HRSS [50]

Input: Polynomial v in S_3
Output: Polynomial $b = \Phi_1((v/\Phi_1) \bmod (3, \Phi_n)) \bmod (q, \Phi_1\Phi_n)$

1: $z = [1/\Phi_1] \bmod (3, \Phi_n) = \sum_{i=0}^{n-2} (1 - i) \cdot x^i \pmod 3$
2: $a = vz \bmod (q, \Phi_1\Phi_n)$
3: **for** $i = 0$ to $n - 1$ **do**
4: $\quad a_i = a_i - a_{n-1} \pmod 3$ $\quad \triangleright\ a = v/\Phi_1 \bmod (3, \Phi_n)$
5: **end for**
6: $b_0 = a_{n-1} - a_0 \pmod q$
7: **for** $i = 1$ to $n - 1$ **do**
8: $\quad b_i = a_{i-1} - a_i \pmod q$ $\quad \triangleright\ b = \Phi_1((v/\Phi_1) \bmod (3, \Phi_n)) \bmod (q, \Phi_1\Phi_n)$
9: **end for**

implemented in an unrolled fashion, achieving highly optimized latency.

Inversion in R_q: To compute the inverse of h in S_q, we perform $h^{-1} \bmod (2, \Phi_n)$ and then apply a variant of the Newton iteration in R_q to obtain $h_q \equiv h^{-1} \bmod (q, \Phi_n)$. The pseudocode of inversion in R_q is given in Algorithm 1. A similar approach is presented in [50], which finds an inverse mod $(2, \Phi_n)$ using $h^{-1} \equiv h^{2^{n-1}-2} \bmod (2, \Phi_n)$. Given that squaring operation in $\mathbb{Z}_2[x]$ is particularly very efficient in software, this approach is suitable for software implementation. In our case, we can re-use our `S3_inverse` module to compute inversion in S_2. All arithmetic operations are now reduced modulo 2 instead of 3 as in inversion in S_3. Operations from lines 3 to 6 in Algorithm 1 are equivalent to 8 polynomial multiplications, which are performed by the Toom-Cook multiplier. Due to the long latency of the polynomial multiplication, inversion in R_q is the most time-consuming operation in Key Generation of NTRU.

5.1.4 `Lift` function

`Lift` function in NTRU-HPS applies a simple map to ternary coefficients of m, converting $\{0, 1, -1\}$ to $\{0, 1, q - 1\}$. This can done on-the-fly by sign extending all the coefficients from $\epsilon_p = 3$ bits to ϵ_q bits.

In NTRU-HRSS, the `Lift` function maps m from S_3 to R_q by doing $m \mapsto \Phi_1 \cdot S_3(m/\Phi_1)$. An efficient implementation of `Lift` is shown in Algorithm 2. As shown in the pseudocode,

`Lift` function can be performed by one multiplication with $z = 1/\Phi_n$ then followed by reduction modulo $(3, \Phi_n)$ and lastly multiplied by Φ_1. Since z is a constant ternary polynomial, it is stored in the memory and the multiplication can be performed by the `Ternary_Poly_Mult` in n cycles. Reduction modulo $(3, \Phi_n)$ and multiplication by $\Phi_1 = x - 1$ can be performed on-the-fly while transferring result back to the memory.

5.1.5 Operations scheduling

In Figs 4, 5, and 6, we show the scheduling of major operations for NTRU-HRSS. Similar schedulings are also applied to NTRU-HPS with two significant differences as follows:

- `Lift`(m) in NTRU-HPS is simpler compared to NTRU-HRSS. The function is executed on-the-fly while storing m to the memory without occupying the Ternary Polynomial Multiplier.
- Sampling m in Encapsulation and g in Key Generation take much longer time since they require constant-time sort-based sampling.

Almost all `pack` and `unpack` operations are hidden by overlapping them with Polynomial Multiplications and/or Sampling. In Decapsulation, the valid checks of r and m are executed in parallel with packing them. The majority of the execution time of Decapsulation is taken by the "large" $\times$ "large" polynomial multiplication. Meanwhile in Key Generation, the polynomial inversion in R_q takes up to 90% of total latency.

5.2 CRYSTALS-Kyber

The proposed hardware architecture for Round 3 Kyber supports the following variants and operations: a) CPA-PKE: Key Generation, Encryption, and Decryption, and b) CCA-KEM: Key Generation, Encapsulation, and Decapsulation. The top-level unit is shown in Fig. 13. The hardware is implemented in Chisel hardware design language [6][52] and incorporates state-of-the-art techniques for optimizing speed and minimizing the resource overhead. The scheduling of operations and units for security level 1 in Decapsulation is shown in Figs 15 and 16.

5.2.1 Polynomial NTT and Multiplication Unit

The Polynomial-Vector Multiplication Unit (PVMU) can perform forward and inverse NTT operations concurrently on up to k polynomials, where k is the security level parameter. This unit also performs polynomial point-wise multiplication (PWM) and accumulation to compute vector-vector and matrix-vector multiplications. The top-level block diagram of PVMU is shown in Fig. 14. At the security level k, the PVMU module consists of k DoubleButterfly pipelines and k memory banks (NTT RAM), each with a single read port and a single write port and datawidth of 4×12 bits (4 coefficients). On the input path, k FIFOs exist, which allow receiving up to k polynomials while a previous operation underway and the main memory bank port are busy.

A DoubleButterfly pipeline consists of two merged parallel configurable radix-2 butterflies, which can operate in three modes of operation: DIT (Decimation in Time) NTT, DIF (Decimation in Frequency) iNTT (inverse NTT), and point-wise multiplication (PWM). During the NTT/iNTT operations, each DoubleButterfly pipeline carries out two radix-2 butterfly operations in parallel for odd and even coefficients. The structure of a DoubleButterfly pipeline is shown in Fig. 17. The butterfly datapath is deeply pipelined (up to 12 stages) to achieve good operating frequency.

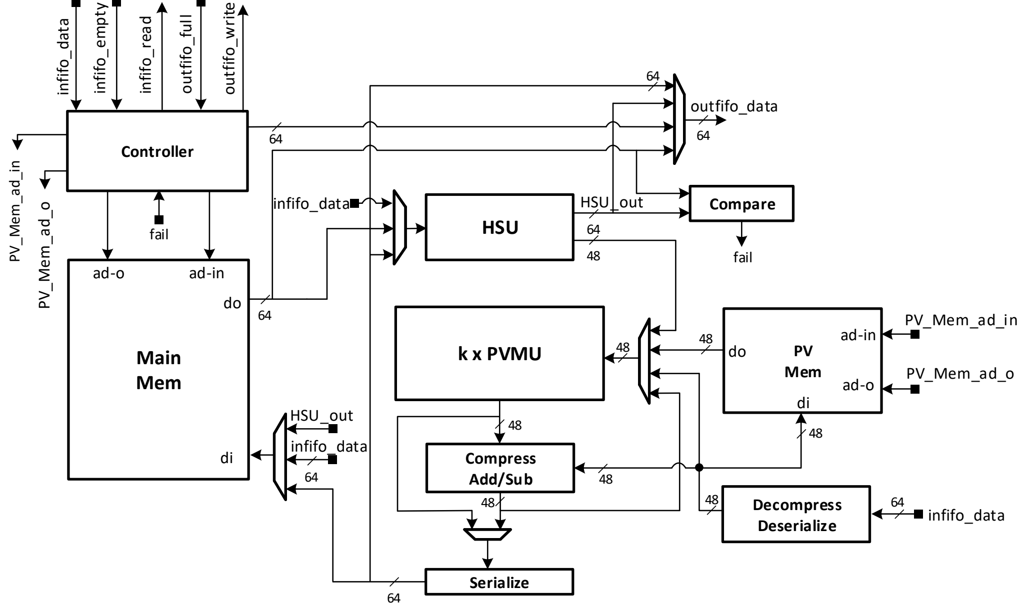

Figure 13: Block diagram of the Kyber top-level datapath

In each butterfly pipeline, a reordering of the input coefficients may be required, depending on the butterfly operation (DIF/DIT/PWM). This is performed by the Head Reorder unit at the input of the butterfly pipeline. A corresponding reordering at the end of the butterfly pipeline is performed by the Tail Reorder unit. During DIF/DIT operations, these reorder units operate as multi-path delay commutators (MDC), [96], not only enabling an efficient memory access scheme similar to [84], but also ensuring correct ordering of the stored coefficients and avoiding the need for any subsequent reordering steps.

The inverse NTT operation involves scaling all coefficients by 256^{-1}. In many software and hardware implementations, the scaling step is performed in a separate step requiring 256 additional field multiplications for each polynomial. By performing a division by $2 \pmod{q}$ at each layer of inverse NTT, the scaling step can be entirely avoided. This observation was also used by Zhang et al. [97]. In that implementation, two divide-by-2 hardware units are utilized to scale both outputs of the radix-2 iNTT butterfly. In our implementation, we use a single divide-by-2 unit for each butterfly, and the other output of each butterfly is scaled by using a scaled copy of the twiddle factors during the inverse transform.

The twiddle factors are stored in a single ROM shared by all butterfly pipelines and are mapped to BRAM-based memory during the FPGA synthesis.

Kyber's point-wise multiplication of polynomials a and b (both in NTT domain) is performed on base degree 1 polynomials in the form of $a_{2i} + a_{2i+1}X$ and $b_{2i} + b_{2i+1}X$. The resulting polynomial $c = a \cdot b$ is calculated using the following formula:

$$c_{2i} + c_{2i+1}X = (a_{2i} + a_{2i+1}X)(b_{2i} + b_{2i+1}X) \bmod X^2 - \zeta_i$$

which results to:

$$\begin{cases} c_{2i} & = a_{2i}b_{2i} + a_{2i+1}b_{2i+1}\zeta_i \\ c_{2i+1} & = a_{2i}b_{2i+1} + a_{2i+1}b_{2i} \end{cases}$$

The straightforward formulation requires 5 modular multiplications for producing a pair of coefficients. As demonstrated by Xing et al. in [95], by using the Karatsuba method, only 4 modular multiplications is required:

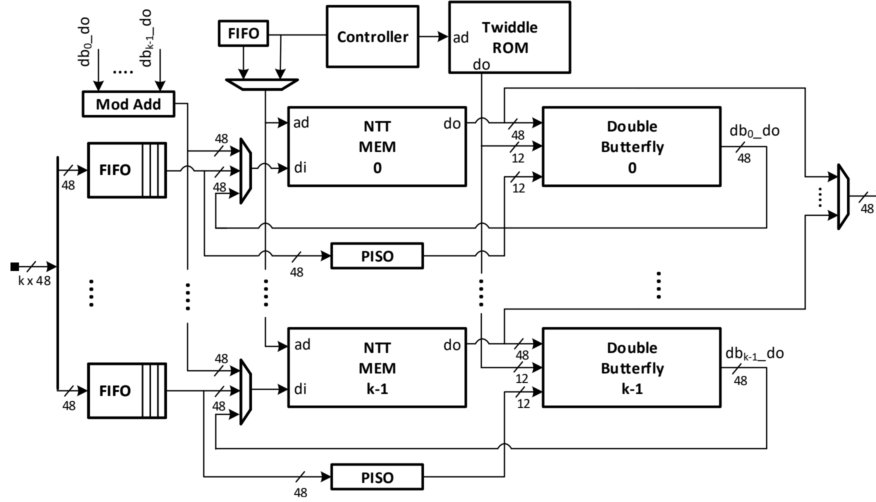

Figure 14: Block diagram of the Kyber Polynomial-Vector Multiplication Unit (PVMU)

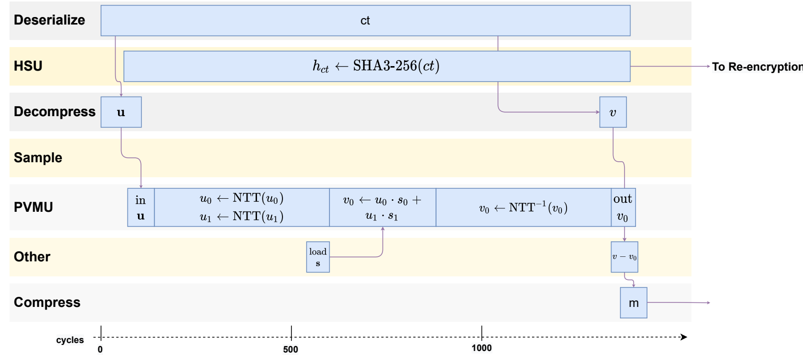

Figure 15: Kyber scheduling of operations for the Decryption stage of Decapsulation at the security level 1

$$\begin{cases} c_{2i} &= a_{2i}b_{2i} + a_{2i+1}b_{2i+1}\zeta_i \\ c_{2i+1} &= (a_{2i} + a_{2i+1})(b_{2i} + b_{2i+1}) - a_{2i}b_{2i} - a_{2i+1}b_{2i+1} \end{cases}$$

With slight adjustments to the double-butterfly structure and careful scheduling of the pipelines, the same resources are used to perform the point-wise multiplication. The scheduling of the DoubleButterfly pipeline for point-wise multiplication is shown in Figure 18. A feedback loop walks coefficients twice through the pipeline to perform 4 modular multiplications and the required addition and subtractions. The two passes through the pipeline are interleaved in such a way to allow full utilization of the multiplier and reduction units of both butterflies in each cycle and requiring only 128 cycles to perform a full polynomial point-wise multiplication (depending on the subsequent operation, a maximum of 21 additional cycles may be required to flush the pipeline).

5.2.2 Barrett reduction with support for division

Coefficients of polynomials are elements of a finite field (or ring) $\mathbf{Z}_q$, where q is a small constant modulus (less than 20 bits). In Kyber q is a prime. This choice requires a modular reduction step after most arithmetic operations to keep the bit width of the

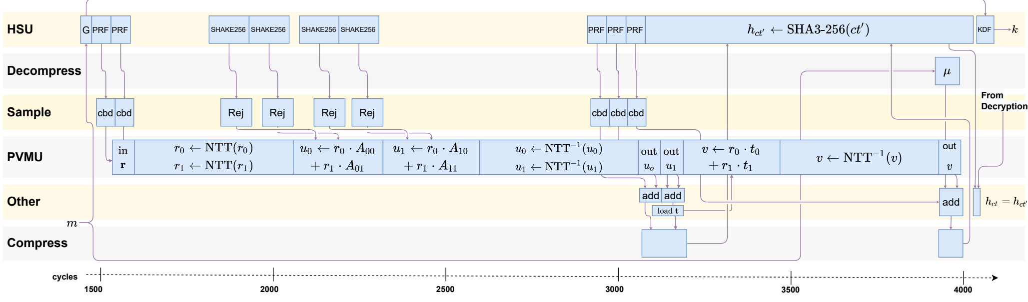

Figure 16: Kyber scheduling of operations for the Re-encryption stage of Decapsulation at the security level 1

Algorithm 3 Optimized Barrett Modular Reduction and Division

Require: $0 \le u \le (q-1)^2$
Ensure: $r = u \bmod q,\ r \in [0, q)$ ▷ remainder
Ensure: $u = d.q + r \bmod q,\ d \in [0, q)$ ▷ quotient

Generation Time: Find optimal values for α and β such that:
1. Only a single conditional subtraction is required
2. Multiplication with the constant μ has minimal hardware complexity.

For Kyber Round 3:
$q := 3329,\ n := \lceil \log_2(q) \rceil = 12,\ \alpha := 12,\ \beta := -2,\ \mu := \lfloor \frac{2^{n+\alpha}}{q} \rfloor = 5039$

function BarrettReduce(u)
  $u_h \leftarrow u \gg (n + \beta)$ ▷ discard $n + \beta$ least-significant bits
  $d \leftarrow (\mu \cdot u_h) \gg (\alpha - \beta)$ ▷ discard $\alpha - \beta$ least-significant bits
  $r \leftarrow u - d \cdot q$
  if $r \ge q$ **then** ▷ conditional subtraction
    $r \leftarrow r - q$
    $d \leftarrow d + 1$
  end if
end function

---

data bounded. Variants of Barrett [11], Montgomery [69], K-RED [65], and SAMS2 [62] reduction algorithms have been used in software and hardware implementations of R-LWE schemes.

We use an optimized variant of the Barrett reduction algorithm shown in Algorithm 3. As shown by Knezevic et al. [58], by careful selection of parameters α and β, only one level of conditional subtraction will be required. The hardware generator code creates optimized single constant multipliers (SCM) based on shift-adder trees and ternary adders based on [60].

5.2.3 Hash and Sampling Unit

Kyber uses the SHA3-256 and SHA3-512 hash functions as well as SHAKE128 and SHAKE256 extendable-output functions. All of them are based on the Keccak permutation. Hash and Sampling Unit (HSU) integrates Keccak core with centered-binomial (CBD) and uniform rejection-based samplers, performing hashing operations in the FO transform as well as generation of noise polynomials and expansion of the public matrix **A**. HSU's block diagram is depicted in Fig. 19. Our Keccak implementation takes advantage of the full-width, basic iterative architecture, which performs 24 rounds in 24 clock cycles. The

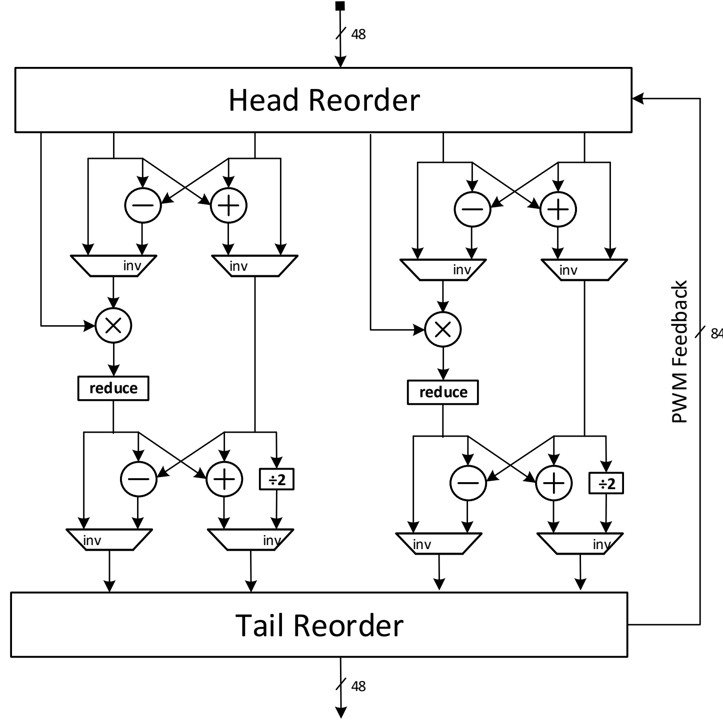

Figure 17: Kyber DoubleButterfly: 2x flexible DIF/DIT butterfly datapath

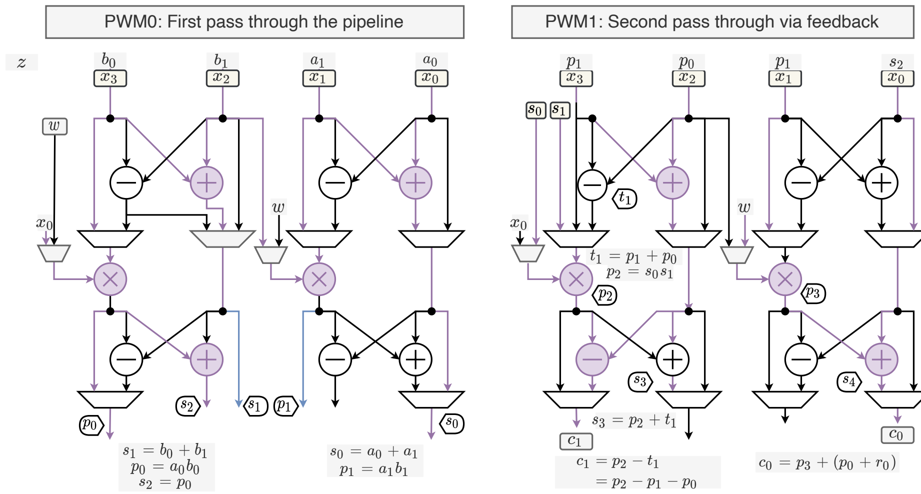

Figure 18: Point-wise multiplication (PWM) in the double-butterfly pipeline

data input and output are 64 bits wide with the valid-ready (decoupled) interface. In Kyber, all hashed data and base seed values (without the "nonce" bytes) are of lengths that are multiples of 64 bits. Based on this observation, we efficiently generate a padding word and append it to the input in a single cycle. The padding word includes specific SHA3/SHAKE padding bytes as well 1 nonce byte when generating noise polynomials (CBD sampling) or 2 nonce bytes during the expansion of matrix **A**. Keccak output is transferred from the state registers to a PISO to allow the next permutations to be

performed while the output is consumed.

the MultiwidthCoverter module converts 64 bits data from the output PISO of the Keccak code to the number of bits required for the selected sampling operation. This module is an improvement to the "Bus Width Converter" design introduced by Farahmand et al. in [30] with support for multiple output widths.

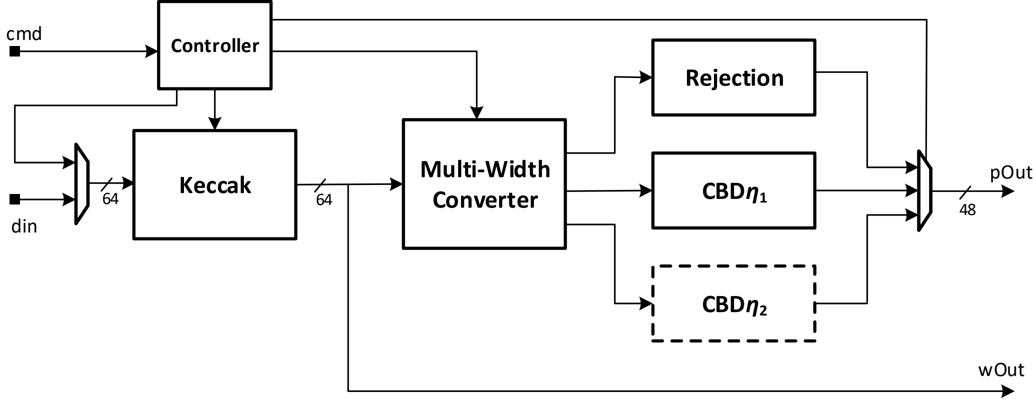

Figure 19: Block diagram of Kyber Hash and Sampling Unit (HSU)

5.2.4 Centered Binomial Sampler

The CBD module in Kyber is responsible for performing the binomial sampling. Kyber requires 12 bits of random data generated by the SHAKE module to generate four coefficients per clock cycle. Two CBD parameters η_1 and η_2 are used. $\eta_2 = 2$ for all security levels, $\eta_1 = 3$ for security Level 1 and $\eta_1 = 2$ for the other security levels. The samples are calculated from formula 4.

$$B_\eta = \sum_{i=1}^{\eta}(a_i - b_i) \tag{4}$$

Hamming weights of the input chunks of the size η are calculated, and negative results are mapped to equivalent mod^+q positive values.

5.2.5 Rejection-based Sampler

In order to minimize the size of the public key, the public matrix A (or its transpose A^T) is generated through the rejection-based sampling of a deterministic random source. The uniform random is generated using SHAKE128 from the public key seed. The output from SHAKE is partitioned into groups of 12 bits, and the resulting unsigned value is only accepted as a valid coefficient if it is less than $q = 3329$. This gives a probability of 81.27% for a sample to be valid. As k^2 sampled polynomials need to be generated through multiple invocations of the Keccak permutation and filtering of coefficients, this step is one of the bottlenecks in Kyber hardware scheduling. The rejection-based sampling of A is inherently not constant time, but any timing variation entirely depends on the public key seed and therefore would not expose any secrets. The Rejection Sampling submodule of the HSU is able to construct each constituent polynomial of 256 coefficients in an average time of 82 cycles. The generation of SHAKE256 output is stopped only when enough valid coefficients are sampled for each polynomial.

5.2.6 Comparison of re-encrypted Ciphertext

During decapsulation, instead of comparing the re-encrypted ciphertext with the received ciphertext, we first generate $H(ct)$ of the original ciphertext. After re-encryption, the hash of the re-encrypted ciphertext ($H(ct')$) is computed, and then only the hashes are compared. This eliminates the need for keeping the original ciphertext. This design decision has negligible cycle overhead but allows a simpler control circuit and also provides a path towards protection against ciphertext malleability side-channel attacks. Additional protection against power and electromagnetic side-channel attacks for this design is under development and will be presented in our future work.

5.2.7 Improvements over Previous Work

A state-of-the-art hardware implementation of Kyber is reported in [95]. Our design has been conducted independently. Both designs employ all relevant optimization techniques reported before, including:

- Flexible DIF/DIT butterflies for performing forward/inverse NTT transforms with efficient resource sharing and avoiding any extra shuffling (bit-reverse ordering) steps.
- Efficient division by two at each step of inverse NTT, eliminating the need for the scaling step.
- Parallel processing of even and odd coefficients using a double-butterfly structure.
- Reuse of DIF/DIT butterflies for performing Kyber's point-wise multiplication.
- Use of Karatsuba algorithm to reduce the number of field multiplications for point-wise multiplications from 5 to 4.

Our improvements over previous work are as follows:

Our high-level architecture and scheduling are based on the use of k DoubleButterfly units with low area utilization (and a single Hash/Sampling unit (HSU). In [95], only one pair of butterflies are used. Our DoubleButterfly datapath is developed to have a low area (around 726 LUTs), which allows efficient exploitation of Kyber's algorithm-level parallelism by deploying k instances of DoubleButterfly, with k set to 2, 3, and 4 for the security levels 1, 3, and 5, respectively.

We utilize an efficient memory access scheme, reducing the memory requirement of each DoubleButterfly unit to a 1-read 1-write (1R1W) 64x48-bit RAM. In Xilinx FPGAs, this memory is mapped to a single BRAM tile (36 Kb block RAM) in the simple dual-port (SDP) mode of operation. Efficient "Head/Tail Re-order" units of the double-butterfly structure perform online re-ordering of coefficients entering/exiting the butterfly pipeline in NTT/invNTT (as a Multi-path Delay Commutator) as well as the re-ordeing required for PWM/MAC. The double-butterfly structure computes the point-wise multiplication through the interleaved reiteration of the pipeline as depicted in Figure 18.

Our deeply pipelined butterfly implementation, including 12 stages, results in a higher maximum clock frequency. The optimized control circuit can skip pipeline flushing stalls whenever possible.

We have developed an optimized reduction unit based on a tweaked version of Barrett's algorithm. This unit has been shown to be faster and more efficient than the other implementations of modular reduction suggested in the literature. It also computes the division by q, required for a fast and efficient implementation of the compression step. As a bonus, our hardware generation code works perfectly for any value of q, including the value used in CRYSTALS-DILITHIUM.

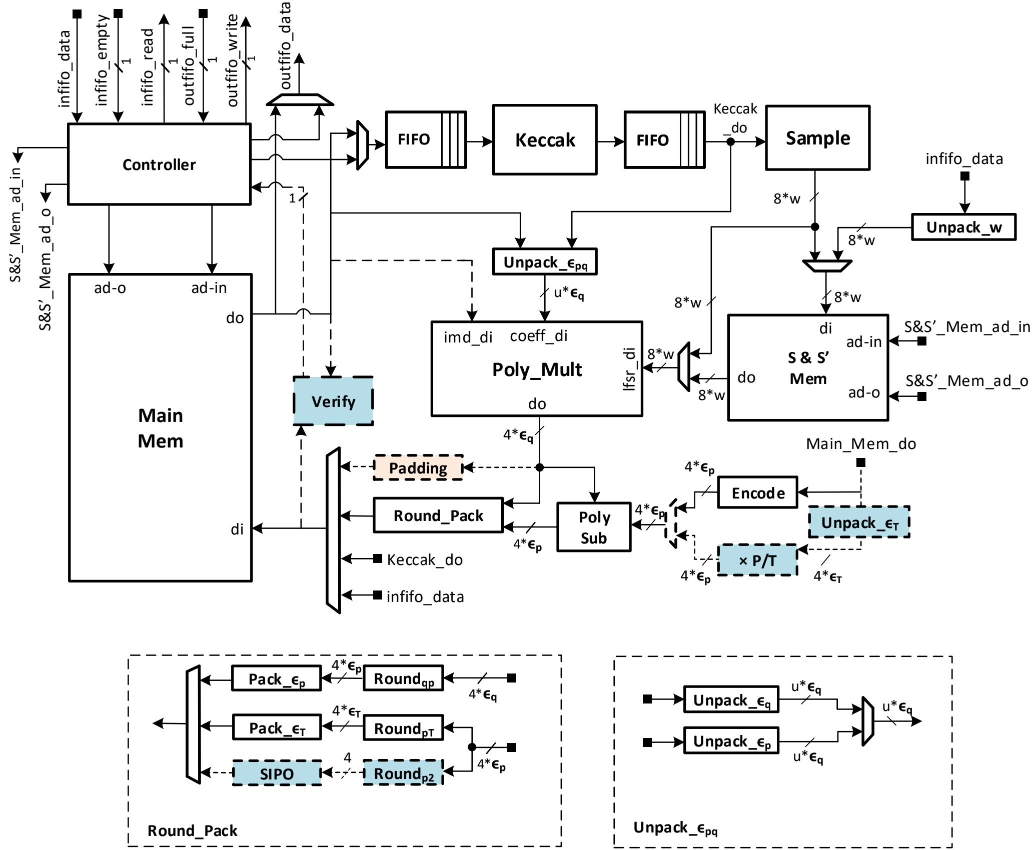

Figure 20: Top-level block diagrams of Saber. The orange, blue modules are used only in key generation and decapsulation, respectively.

Our fast and efficient implementation of the Rejection sampler processes four coefficients at a time, reducing the expansion of public matrix to 116 cycles per polynomial ($116k^2$ cycles total).

Our fast and efficient Keecak implementation has input and output widths of 64 bits, with decoupled output and efficient SHA3/SHAKE padding of the input words.

Efficient implementation of the CBD sampler which can simultaneously supports η values 2 and 3 (for security level 1) and provide output in the standard range.

Finally, unlike [95], our design is technology-independent and does not employ any vendor-specific IPs. These features allow for easy deployment on FPGA platforms other than Xilinx, use of synthesis flows other than Vivado (including open-source FPGA flows), as well as porting to ASICs.

5.3 Saber

The top-level block diagram is shown in Fig. 20.

5.3.1 Sampling

The diagram of our CBD sampling modules for three parameter sets of Saber is shown in Fig. 21. The values of coefficients sampled from CBD are in the range [-5; 5], [-4; 4], and [-3;3], corresponding to the bit-width $w = 4, 4, 3$. The 64-bit inputs are buffered in the dual-step shift register. After the shift register is full, chunks of data are read out and

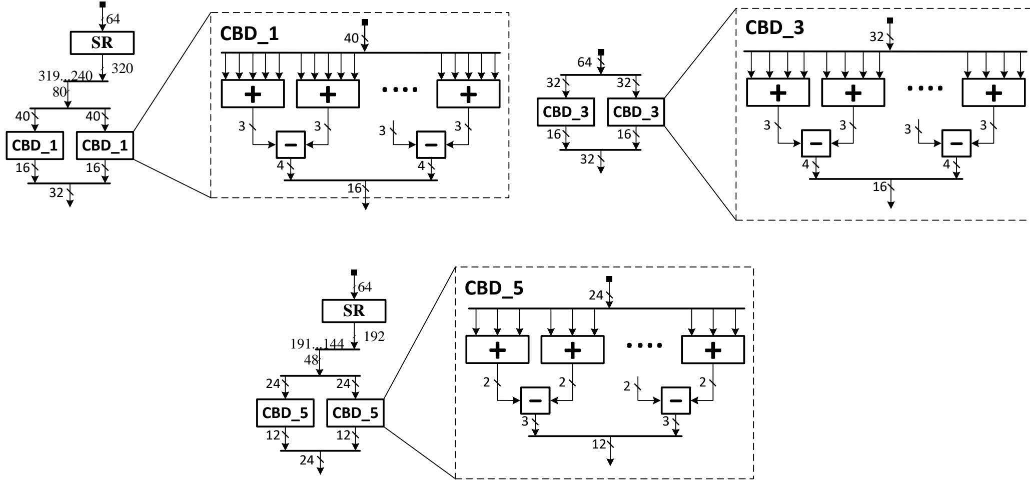

Figure 21: Sampling modules for three parameter sets of Saber.

Table 7: Polynomial Multipliers for Saber Level 3.

	Cycles	**Freq [MHz]**	**μs**	**LUT**	**FF**	**DSP**	**BR AM**	**Device**
Schoolbook-TW	256	370	0.7	11,426	8,727	0	0.0	Zynq U.+
NTT-TW	^{e}908	400	2.3	4,831	2,260	8	3.0	Zynq U.+
Karatsuba [98]	81	160	0.5	13,735	4,486	85	0.0	Zynq U.+
Toom-Cook 4 [68]	1,168	125	9.3	2,927	1,279	28	2.0	Zynq
NTT [32]*	^{e}11,008	153	71.9	2,454	1,917	7	4.5	Artix-7

* A first-order masking design that supports multiple lattice-based schemes.
^{e} Total latency for multiplication is estimated by: 2× NTT + INTT + Pointwise Multiplication.

Table 8: Implementation results of the Optimized Polynomial Multiplier using optimized integer multipliers vs. the centralized multiplier architecture in [12]

	Optimized Multiplier	Centralized Multiplier
LightSaber	**12,492 LUTs, 8,727 FFs**	13,658 LUTs, 8,727 FFs
Saber	12,492 LUTs, 8,727 FFs	**11,426 LUTs, 8,727 FFs**
FireSaber	**8,726 LUTs, 8,215 FFs**	8,734 LUTs, 8,215 FFs

fed through a pure combinational logic to generate the coefficients. The output width of sampling modules is equal to $8 * w$. Therefore, we will have 8 samples generated per clock cycle.

5.3.2 Polynomial Multiplication

The preliminary choice of a multiplier type for Saber is discussed in Section 3.3. However, considering that for Saber, other groups have attempted different multiplier types, a more detailed analysis is possible.

The high-speed SW/HW codesign of Saber in [22] uses a schoolbook-based multiplier, which requires 256 DSPs with 13-bit inputs. A Toom-Cook based multiplier for Saber is proposed in [68], also in the SW/HW co-design context. The Saber crypto-processor implementation in [88] uses a schoolbook-based multiplier, which exploits the small sizes of input coefficients. It can provide very good performance with moderate resource consumption. [12] improves the multiplier used in [88] by centralizing coefficient-wise multiplication and replacing integer multipliers with simple multiplexers. [33] proposed an approach to use an NTT module to speed up polynomial multiplication in Saber based on

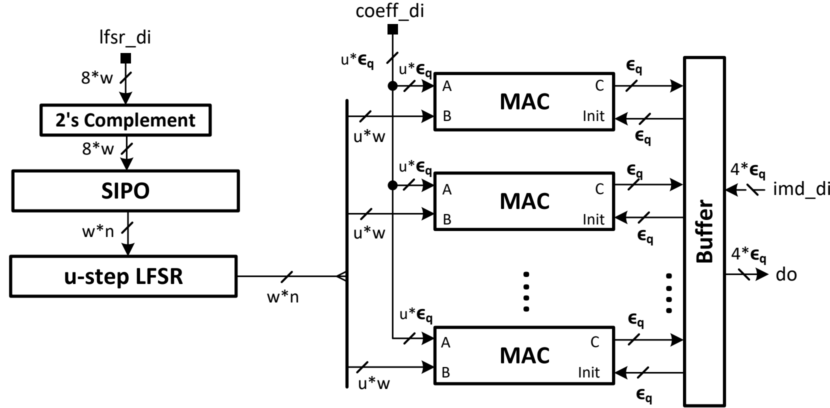

Figure 22: Schoolbook-based polynomial multiplier with unroll factor $u = 1, 2, 4$.

the Chinese Remainder Theorem. Recently, [98] introduced an 8-level Karatsuba multiplier for Saber with efficient scheduling of operations, which achieves very small latency in terms of clock cycles. However, it requires a large area and a long critical path, which leads to low clock frequency. A masked accelerator for RISC-V platform is presented in [32]. It includes an NTT-based multiplier which supports Saber and multiple other lattice-based schemes.

In Table 7, we summarize attempts at implementing Saber's polynomial multiplication using four different multiplier types. The Karatsuba and Schoolbook multipliers are clearly the fastest but also the largest (in terms of the number of LUTs and FFs) among the four attempted designs. The main advantage of the Schoolbook multiplier is no use of DSP units. However, if this resource is not critical and the Latency[μ s] is more important than the number of LUTs, then Karatsuba is a slightly better choice. The Toom-Cook 4, implemented in [68], is significantly slower than Karatsuba and Schoolbook, even considering the effect of different FPGA devices. This multiplier also uses several times fewer LUTs. The NTT multiplier reported in [32] is slightly smaller than Toom-Cook 4, but it requires about 7.7x more time. This result may be, however, affected by the first-order masking design of the multiplier in [32], so more investigation may be warranted in the future. Overall, for the purpose of this project, we decided to select the Schoolbook and NTT-based multipliers.

Schoolbook-based Polynomial Multiplier. The block diagram of the schoolbook-based polynomial multiplier for Saber is shown in Fig. 22. Since there are multiple multiplications involved in vector-vector or matrix-vector multiplication, we improve the latency of multiplication by adding input and output buffers. The buffers are capable of pre-loading the next input polynomial as well as unloading the previous product polynomial at the same time as the current multiplication is performed. The `S&S'MEM` stores all small coefficients of secret polynomials in their unpacked form. Thus, it can provide one polynomial in 32 cycles. The latencies of loading and unloading polynomials are hidden in the multiplication latency. The multiplier can also be unrolled by a factor $u = 1, 2$, or 4, which can finish one polynomial multiplication in 256, 128 or 64 cycles, respectively. Instead, having simple integer coefficient-wise multipliers, which are based on shift-add operations, as in [88], we generated optimized integer multipliers using an open-source tool FloPoCo [23]. We also tried the centralized coefficient-wise multiplier approach proposed in [12]. We report the results of the two approaches in Table 8. The centralized multipliers approach has better area consumption in the case of Saber, so we use this approach for the specific parameter set. For LightSaber and FireSaber, the optimized integer multipliers are used.

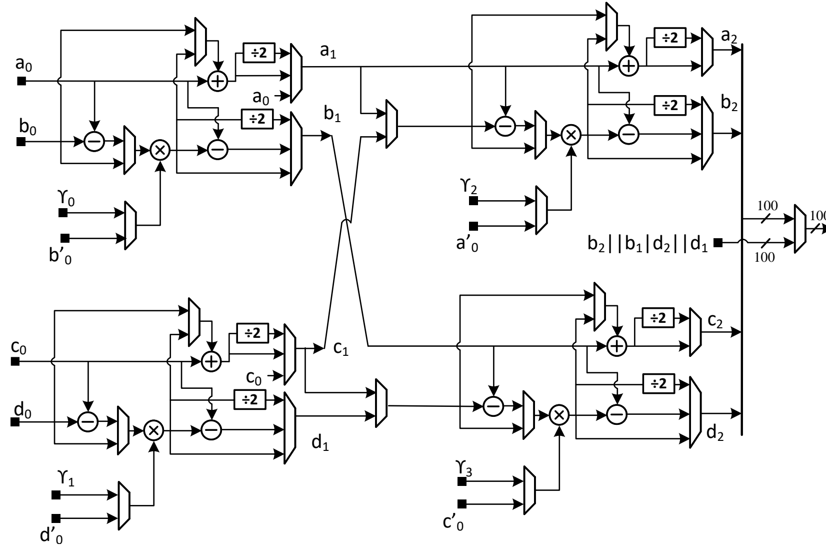

Figure 23: Block diagram of the 2x2Butterfly unit. All adders, subtractors and multipliers are followed by a signed reduction modulo p. All bus widths are 25 bits unless specified.

NTT-based Polynomial Multiplier. Given that polynomial multiplication in Saber can be treated as negative-wrapped convolution with the reduction polynomial $x^n + 1$, $n = 2^k$, NTT can be used for Saber by choosing an appropriate lifted modulus p. Recent work by Chung et al. [19] shows a trick allowing to apply NTT to Saber in software implementations targeting Cortex-M4 and AVX2. Fritzmann et al. applied the same idea in a masked accelerator for the RISC-V platform targeting Saber and other lattice-based schemes [32].

In Kyber, incomplete NTT is used by default because the modulus q does not support full NTT. This approach requires the point-wise multiplication of degree-1 polynomials using the Karatsuba method. For Saber, we chose to do complete NTT to reduce the complexity of the NTT module. To avoid padding polynomials with degree n to $2n$ in negative wrapped convolution, one can multiply input polynomials with powers of the $2n$th root of unity, γ. To recover the coefficients of the product polynomial during Inverse NTT, we multiply the product polynomials with powers of γ^{-1} and the scaling factor n^{-1}. In order to make the $2n$th roots of unity γ exist, the prime modulus p must satisfy the equation: $p = 1 \bmod 2n$. When multiplying a polynomial with coefficients in $[0, q-1]$ by a polynomial with small coefficients in $[-\frac{\mu}{2}; \frac{\mu}{2}]$, the coefficients of the product will be in the range $(-\frac{nq\mu}{2}; \frac{nq\mu}{2})$. Thus, we can choose an NTT-friendly prime $p > qn\mu$, and then use NTT to do polynomial multiplication treating coefficients as integers, and then reduce coefficients modulo the actual q. Based on this condition and the condition for the $2n$th roots of unity, we chose the value of $p = 33,550,337 = 2^{25} - 2^{12} + 1$ (25 bits) which is applicable to all Saber parameter sets. The special structure of p enables a very efficient modulo p reduction. All coefficients during the NTT-related operations are reduced to stay in the range $[\frac{-p}{2}, \frac{p}{2})$. By using NTT, a multiplication in $Z_q/(x^n + 1)$ can be computed as follows:

$$C = \mathbf{INTT}(\mathbf{NTT}(A) * \mathbf{NTT}(B))$$

The matrix-vector multiplication in Saber involves computing a product of $l \times l$ matrix of polynomials and a $l \times 1$ vector. We compute negacyclic NTT for polynomials in the matrix and vector, do pointwise multiplications, and accumulate the results to a vector in the NTT domain. Then we only need to do l inverse NTT operations to obtain the

results. Similarly, the inner product of two $l \times 1$ vectors can be computed using $2l$ NTT and 1 inverse NTT operation. Similar to our Kyber implementation, we opt to use a Polynomial-Vector Multiplication Unit (PVMU) which can process up to l polynomials at the same time.

Kyber is specifically designed with NTT as a method to do polynomial multiplications. Its matrices and vectors are sampled and stored as parts of keys or ciphertext in the NTT domain. In Saber, all components are generated and transferred into the normal domain. Therefore Saber requires substantially more NTT and inverse NTT operations. To compensate for this disadvantage, we further extend our DoubleButtefly unit to a 2x2Butterfly unit. This unit builds upon the NTT design by Nguyen et al. [70, 71]. It consists of 4 configurable radix-2 butterflies and processes 4 coefficients and 2 subsequent NTT layers in every clock cycle. The ideal latency for NTT operations using this module is $\frac{n}{4}\frac{logn}{2}$ cycles. There are 3 modes of operation: NTT (Decimation in Time), iNTT (Decimation in Frequency), and point-wise multiplication. The structure of the 2x2Butterfly unit is shown in Fig. 23. The unit has 14 stages of pipelines to achieve optimal maximum frequency. Each NTT and each iNTT operation takes 276 cycles; a point-wise multiplication takes 80 cycles. One NTT module for Saber requires 4831 LUTs, 2260 FFs, 8 DSP units, and 3 BRAMs when synthesized at 400 MHz in our Zynq-Ultrascale+ device.

5.3.3 Operations Scheduling

The timing diagrams of Key Generation, Encapsulation, and Decapsulation of LightSaber implemented using the schoolbook-based multiplier are shown in Figs. 26, 24 and 25, respectively. Similar scheduling is used when the NTT-based multiplier is employed. During encapsulation and decapsulation, the generation of a vector of polynomials with "small" coefficients in CBD takes uniform inputs from Keccak module. Whenever a polynomial is generated, it is then loaded into the polynomial multiplier and at the same time, stored in `S&S'MEM` for later use. Multiplication can start as soon as a polynomial with "small" coefficients is fully loaded. For each cycle during multiplication, 1, 2, and 4 coefficients of a polynomial in A are fetched to the multiplier with unroll factor u = 1, 2, and 4, respectively. The `unpack_`ϵ_{pq} modules serialize 64-bit data block into 13-bit coefficients. Since the multiplier consumes data at a slower rate than the `Keccak` module, `Keccak` module works intermittently when generating A. It stops its operation when there is still data left to be read by the multiplier. The next small noise polynomial is loaded in parallel with the multiplication of the current polynomials. Thus the multiplier is always busy during vector-vector and matrix-vector multiplication. The result polynomials are rounded and packed and stored into the `Main_Mem`. During key generation, matrix A is generated in column-major order. The intermediate results of polynomials in vector b have to be stored in `Main_Mem`.

5.3.4 Improvements over Previous Work

The Karatsuba-based multiplier in [98] can execute polynomial multiplications in a very low number of clock cycles. By employing pipelined 8-level Karatsuba unit with efficient scheduling, one polynomial multiplication can be completed in 81 cycles. However, its area consumption is much higher than that of our multiplier in terms of DSP units (85 vs. 0) and BRAMs (6.0 vs. 1.5). Due to the recursive nature of the algorithm, the critical path through the multiplier is longer. Consequently, the circuit can only operate at a low frequency, 100 MHz and 160 MHz, for the unified and Saber Level 3 architecture, respectively. By unrolling our schoolbook-based multiplier, with the unrolling factors equal to 2 and 4, respectively, we obtain the designs Saber x2 and Saber x4, listed in Table 13. These designs can achieve comparable cycle count while having a much higher maximum

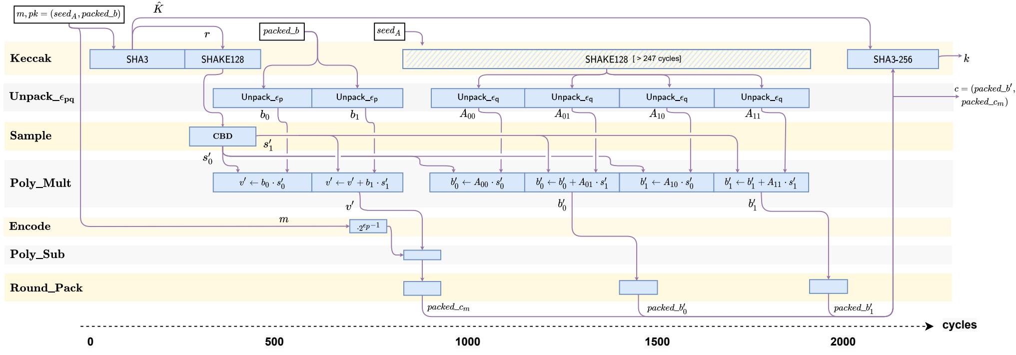

Figure 24: Operations Scheduling for Encapsulation of LightSaber.

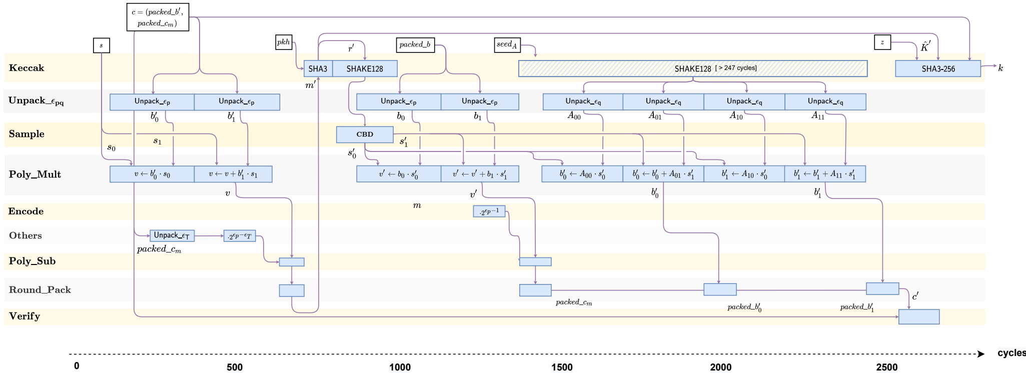

Figure 25: Operations Scheduling for Decapsulation of LightSaber.

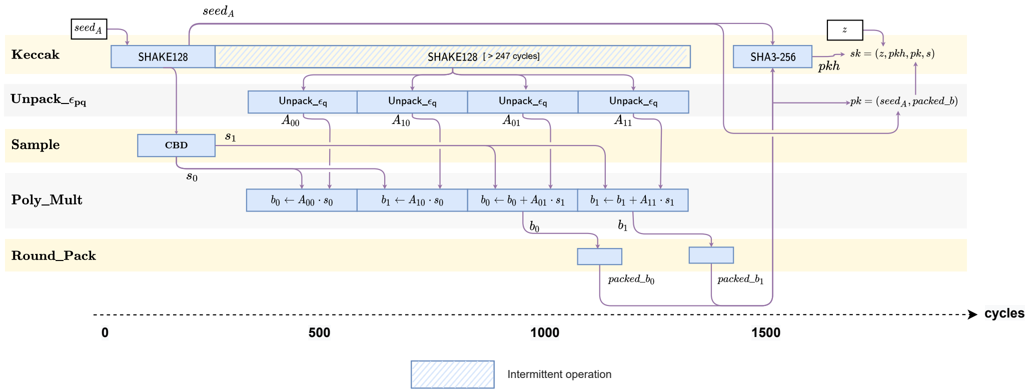

Figure 26: Operations Scheduling for Key Generation of LightSaber.

clock frequency and hence also better latency in μs. We note that the design in [98] targets an ASIC platform. Thus, both the area consumption and maximum frequency on FPGA might still be improved.

The high-speed instruction-set coprocessor in [87] offers flexibility in supporting multiple parameter sets in a unified architecture. The polynomial multiplier is then improved in [12]. Area consumption is significantly reduced while keeping the same latency in clock cycles.

The coprocessor design, however, limits exploiting parallelism between non-data-dependent operations. As we have demonstrated in our timing diagrams, many operations can be executed in parallel with polynomial multiplications. In Table 9, we show in detail how our efficient scheduling of operations for Saber Level 3 can improve the overall latency of Decapsulation. The latency of polynomial multiplications is improved by 14% using buffers, allowing loading input and unloading output in parallel with multiplications. SHAKE128 and CBD sampling are almost fully overlapped. The remaining operations only include loading random inputs, ciphertext, rounding, and packing final polynomial in the result vectors. Consequently, the total cycle count for Decapsulation at Level 3 is shortened by more than 40%.

Compared to the previous work on the implementations of Saber, reported in [87] and [98], we further optimize the schoolbook multiplier. Additionally, we optimize the scheduling of all operations in hardware by fully exploiting the potential for parallel processing of operations without data dependencies. Our implementation achieves the best latency and the usage of LUTs.

Table 9: Comparison to Saber implementation in [87] for Level 3 Decapsulation based on [[87], Table 1]. Operations with * have majority of their latency overlapped with Polynomials Multiplication in our design. Hence, only their non-overlapped cycles contributing to the total cycle count are reported.

Operations	Decapsulation	
	[87]	**TW**
SHA3-256	303	294
SHA3-512	62	47
SHAKE-128*	1,403	51
CBD Sampling*	176	89
Polynomial multiplications	4,484	3,873
Remaining operations*	1,606	328
Total Cycles	8,034	4,682

6 Results

6.1 NTRU

The results of our implementations of two variants of NTRU, NTRU-HRSS (at the security level 1) and NTRU-HPS (at the security levels 1 and 3), are summarized in Table 10.

Table 10: Implementation results of NTRU on Zynq UltraScale+

Design	Module	Freq	LUT	FF	Slice	DSP	BRAM	Latency	
								Cycles	**μs**
Security Level 1									
NTRU-HRSS701	Key Gen.	300	49,001	39,957	9,357	45	2.5	51,812	172.7
	Encaps.	300	31,494	25,120	6,652	0	2.5	2,219	7.4
	Decaps.	300	37,702	34,441	8,032	45	2.5	8,826	29.4
NTRU-HPS677	Key Gen.	250	41,047	39,037	7,968	45	6	48,179	192.7
	Encaps.	250	26,325	17,568	4,638	0	5	3,687	14.7
	Decaps.	300	29,935	19,511	5,217	45	2.5	7,522	25.1
Security Level 3									
NTRU-HPS821	Key Gen.	250	50,347	44,281	10,127	45	6.5	67,157	268.6
	Encaps.	250	33,698	30,551	7,370	0	5.5	4,576	18.3
	Decaps.	300	38,642	33,003	7,785	45	2.5	10,211	34.0

At the security level 1, NTRU-HRSS outperforms NTRU-HPS for key generation and encapsulation. However, it slightly lags behind for decapsulation. NTRU-HRSS operates at a higher clock frequency (except for decapsulation) but requires consistently more resources than NTRU-HPS. With the increase in the security level, NTRU-HPS requires more FPGA resources, with the exception of DSP units, the number of which remains the same.

In NTRU-HPS, the maximum clock frequency for the key generation and encapsulation is limited by the sort-based sampling unit. This unit is not a part of the decapsulation core. Consequently, decapsulation can be performed at a 50 MHz higher clock frequency.

6.2 CRYSTALS-Kyber

In Table 11, we report our results for CRYSTALS-Kyber and compare them with previous hardware-only implementations. We omit software/hardware implementations reported in [9], [10], [33], [3], and [94]. These implementations are clearly inferior in terms of both the latency and the product of the latency and the number of LUTs.

The implementation of Kyber presented in this work outperforms the best previous implementation, reported in [95], by approximately a factor of two in terms of the execution time in microseconds for all major operations (key generation, encapsulation, and decapsulation). The comparison in terms of resource utilization is less obvious, considering that all operations are allowed to share the same resources in this work. In [95], the resource utilization for the server side (executing key generation and decapsulation) and the client side (executing encapsulation) are reported separately. However, based on our design, extending the coverage of operations from the server side to include encapsulation has negligible influence on the circuit area. Thus, it seems fair to compare our resource utilization numbers with the corresponding numbers for the server unit in [95].

The implementation reported in [49] is significantly less efficient. It also does not support key generation. The execution time for encapsulation is 21.5x and 29.4x longer in [49] compared to this paper, at the security levels 1 and 3, respectively. For decapsulation, the corresponding ratios of the execution times are 21.7x and 29.7x. At level 5, only Virtex-7 results are reported in [49].

6.3 Saber

The results of our implementations of Saber using the schoolbook multiplier at the security levels 1, 3, and 5, targeting Zynq UltraScale+, are summarized in Table 12. This table demonstrates three clear advantages of Saber: 1) the resource utilization stays almost the same, independently of the security level, 2) the maximum clock frequency is independent of the security level, 3) implementations use no DSP units and a very small number of BRAMs. Only latency is affected considerably by using higher security levels.

The comparison with our implementation using NTT-based multiplier and the best implementations of Saber reported in the literature to date is shown in Table 13 and Figs 34, 36, and 38. In Table 13, our implementations are marked in bold.

To achieve a fair comparison with other types of multipliers, all inputs and outputs of our NTT-based Saber implementation are in normal form and the latency of computing the NTT/iNTT of the inputs and outputs are all taken into account. The designs with the terms x2 and x4 in the name are obtained by unrolling the schoolbook polynomial multiplication unit by 2 and 4 times, respectively. These designs offer significant improvements in latency at the cost of a substantial increase in the number of LUTs, flip-flops, and slices. In Figs 34, 36, and 38, the implementation described in [98] is denoted as Saber-Tsinghua, the implementation from [87] as Saber-U.Birmingham, and our designs as Saber-TW. Based on these figures and Table 13, Saber x4 and Saber x2 are the fastest, Saber x1 and Saber-NTT are the smallest among the compared high-speed implementations.

Table 11: Implementation results for different Kyber instances on various FPGAs. Notation: S/C - Server/Client, E/D - Encapsulation/Decapsulation.

Scheme	Key/Encaps/Decaps [K Cycles]	Freq. [MHz]	Key/Encaps/Decaps [us]	LUT	FF	DSP	BRAM	Device
Kyber-CCAKEM L1								
Kyber R3 [this work]	2.2/3.2/4.5	220	10.0/14.7/20.5	9,457	8,543	4	4.5	Artix-7 XC7A200
Kyber R3 [95]	3.8/5.1/6.7	S/C 161/167	23.4/30.5/41.3	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2/2	Artix-7 XA7A12
Kyber R2 [49]	–/49.0/68.8	155	–/316/444	E/D 80,322/88,901	N/A	E/D 54/354	E/D 200.5/202	Artix-7 XA7A12
Kyber R3 [this work]	2.2/3.2/4.5	450	4.9/7.2/10.0	9,504	8,957	4	4.5	Zynq-UltraScale+ XCZU7EV
Kyber-CCAKEM L3								
Kyber R3 [this work]	2.6/3.7/4.9	220	12.0/17.0/22.2	10,530	9,837	6	6.5	Artix-7 XC7A200
Kyber R3 [95]	6.3/7.9/10.0	S/C 161/167	39.2/47.6/62.3	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2/2	Artix-7 XA7A12
Kyber R2 [49]	–/77.5/102.1	155	–/500/659	E/D 97,085/110,260	N/A	E/D 36/292	E/D 200.5/202	Artix-7 XA7A12
Kyber R3 [this work]	2.6/3.7/4.9	450	5.9/8.3/10.9	10,590	10,458	6	6.5	Zynq-UltraScale+ XCZU7EV
Kyber-CCAKEM L5								
Kyber R3 [this work]	3.6/4.8/5.8	220	16.2/21.7/26.4	11,623	11,131	8	8.5	Artix-7 XC7A200
Kyber R3 [95]	9.4/11.3/13.9	S/C 161/167	58.2/67.9/86.2	S/C 7412/6785	S/C 4644/3981	S/C 3/3	S/C 2/2	Artix-7 XA7A12
Kyber R2 [49]	–/107.1/135.6	192	–/558/706	E/D 119,189/132,918	N/A	E/D 36/548	E/D 200.5/202	Virtex-7 VC707
Kyber R3 [this work]	3.6/4.8/5.8	450	7.9/10.6/12.9	11,676	11,959	8	8.5	Zynq-UltraScale+ XCZU7EV

Table 12: Implementation results of Saber on Zynq UltraScale+

Design	Module	Freq	LUT	FF	Slice	DSP	BRAM	Latency	
								Cycles	μs
Security Level 1									
LightSaber	Key Gen.	370	23,557	14,190	3,844	0	1.5	1,607	4.3
	Encaps.	370	24,199	14,457	3,984	0	1.5	2,153	5.8
	Decaps.	370	24,655	14,879	4,364	0	1.5	2,794	7.6
Security Level 3									
Saber	Key Gen.	370	20,496	13,939	3,634	0	1.5	2,709	7.3
	Encaps.	370	21,069	14,074	3,503	0	1.5	3,735	10.1
	Decaps.	370	21,342	14,233	3,816	0	1.5	4,682	12.7
Security Level 5									
FireSaber	Key Gen.	370	19,752	14,358	3,321	0	1.5	4,895	13.2
	Encaps.	370	20,696	13,949	3,455	0	1.5	5,867	15.9
	Decaps.	370	20,868	14,237	3,460	0	1.5	7,128	19.3

6.4 Comparison of Round 3 candidates

In Figs. 27–38, we illustrate the dependence between the speed of the Round 3 candidates (in the number operations per second, which, for all considered designs, is equivalent to the inverse of latency in time units) and their resource utilization in LUTs. All other components of resource utilization, such as the number of BRAMs or DSP units, are omitted for simplicity. In terms of the percentage of the total amount of FPGA resources, the utilization of LUTs is typically the highest. However, some exceptions to this typical scenario may occasionally occur. The exact correspondence between the names of designs given in these figures' legends and the related publications is shown below: <candidate_name>-TW – this work, Classic McEliece-Yale U. – [92], [91], FrodoKEM-PQShield/Bristol – [47], BIKE-R-U Bochum, Intel – [82, 81], CRYSTALS-Kyber-Tsinghua

Table 13: Implementation results of Saber and comparison with related work on ZynqUltrascale+ platform.

Design	Key/Encaps/Decaps [K Cycles]	Freq [MHz]	Key/Encaps/Decap [us]	LUT	FF	Slices	DSP	BRAM
Security Level 1								
LightSaber x4	0.9/1/1.3	310	2.9/3.3/4.2	65,890	28,230	10,404	0	1.5
LightSaber x2	1.1/1.4/1.8	345	3.2/4.1/5.2	39,423	21,467	6,610	0	1.5
LightSaber x1	1.6/2.2/2.8	370	4.3/5.8/7.6	24,688	14,785	4,309	0	1.5
LightSaber-NTT	2.0/3/3.9	400	5.0/7.5/9.8	16,617	10,575	3,201	16	7.5
Unified Saber [98]	0.6/0.9/1.1	100	6/8.6/10.8	34,886	9,858	—	85	6.0
Unified Saber [87]	2.8/4/5	150	18.4/26.9/33.6	24,979	10,732	—	0	2.0
Security Level 3								
Saber x4	1.3/1.5/1.9	310	4.3/4.8/6	48,895	27,715	7,726	0	1.5
Saber x2	1.8/2.2/2.8	345	5.2/6.5/8.1	32,099	21,037	5,294	0	1.5
Saber [98]	1.1/1.5/1.7	160	6.7/9.1/10.6	28,169	9,504	—	85	6.0
Saber x1	2.7/3.7/4.7	370	7.3/10.1/12.7	21,352	14,232	3,763	0	1.5
Saber-NTT	3.0/4.2/5.7	400	7.6/10.5/14.2	21,555	12,254	3,891	24	10.5
Unified Saber [98]	1.1/1.5/1.7	100	10.7/14.6/17	34,886	9,858	—	85	6.0
Saber [87]	5.5/6.6/8	250	21.8/26.5/32.1	25,079	10,750	—	0	2.0
Unified Saber [87]	5.5/6.6/8	150	36.4/44.1/53.6	24,979	10,732	—	0	2.0
Security Level 5								
FireSaber x4	2/2.1/2.6	310	6.5/6.9/8.5	38,268	27,677	6,348	0	1.5
FireSaber x2	2.9/3.4/4.1	345	8.4/9.8/11.9	25,760	21,035	4,239	0	1.5
FireSaber-NTT	4.5/5.7/6.2	400	11.1/14.2/15.4	25,794	15,040	4,147	32	13.5
FireSaber x1	4.9/5.9/7.1	370	13.2/15.9/19.3	20,383	14,239	3,408	0	1.5
Unified Saber [98]	1.7/2.2/2.5	100	17.2/21.9/24.8	34,886	9,858	—	85	6.0
Unified Saber [87]	9/10.3/12.3	150	60.2/68.4/82	24,979	10,732	—	0	2.0

– [95], CRYSTALS-Kyber-Nanjing U. – [49], HQC-HQC Team – [83], SIKE-FAU – [26] , Saber-Tsinghua– [98], Saber-U. Birmingham– [87], StrNTRUPrime-TUHH,NXP,NTU,IIS – [66, 76].

For security level 1, the number of implementations on Artix-7 FPGAs, illustrated in Figs. 27, 29, and 31, is 10 for key generation and 11 for encapsulation and decapsulation. These implementations represent eight Round 3 KEMs (all except NTRU Prime). Saber is the fastest for all three major operations. From left to right, Saber-TW is represented by four diamonds corresponding to the NTT, x1, x2, and x4 architectures. CRYSTALS-Kyber is clearly the second for key generation and decapsulation. In the case of encapsulation, Saber x1 is practically tied with NTRU-HRSS in terms of latency but smaller in terms of area. Saber x2 and Saber x4 are faster but bigger in terms of the LUT count. Compared to Saber x1, NTRU-HRSS and NTRU-HPS are about 3x slower for decapsulation and over 30x slower for key generation. Classic McEliece is more than two orders of magnitude slower than Saber for key generation, about an order of magnitude slower for decapsulation, and only a few times slower for encapsulation. It requires a comparable number of LUTs. FrodoKEM and SIKE are at least two orders of magnitude slower than Saber for all three operations. However, they can be implemented using fewer LUTs. BIKE trails Saber by more than one order of magnitude for encapsulation, and almost two orders of magnitude for key generation and decapsulation.

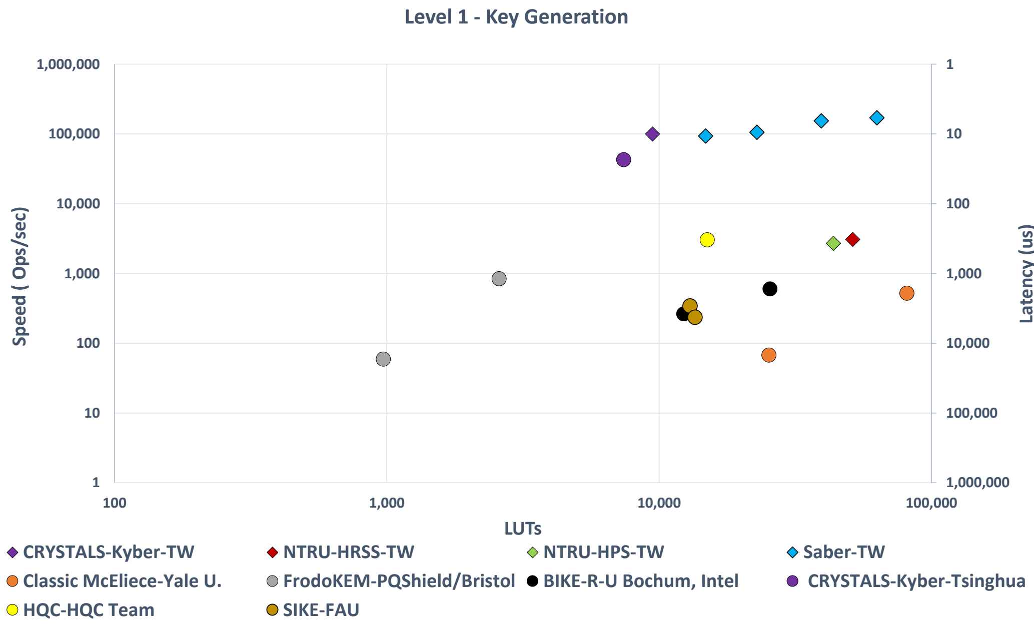

Figure 27: L1, KeyGen, Artix-7

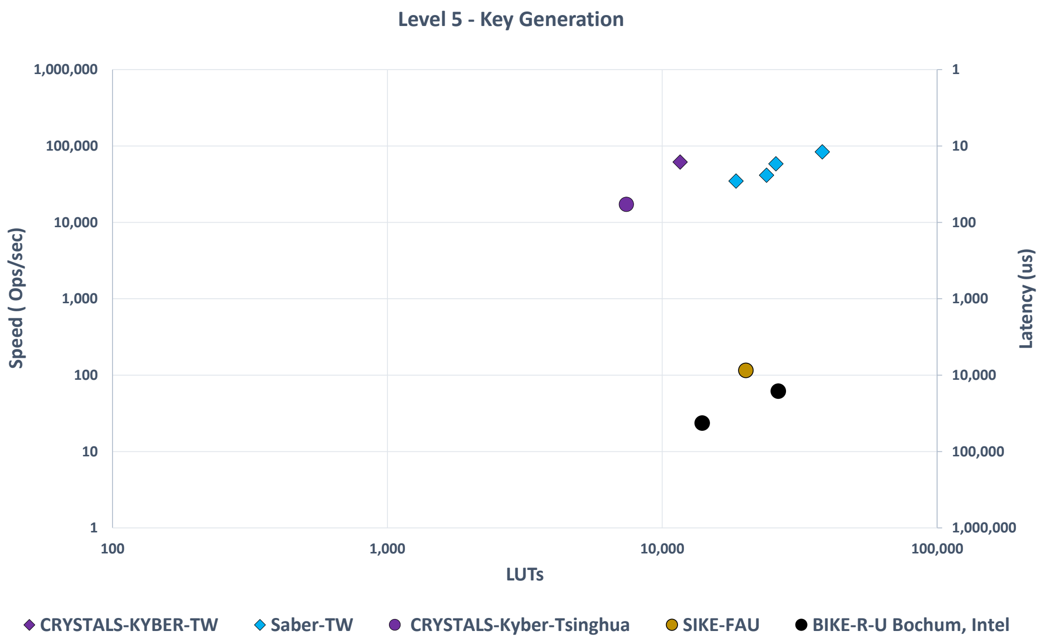

Figure 28: L5, KeyGen, Artix-7

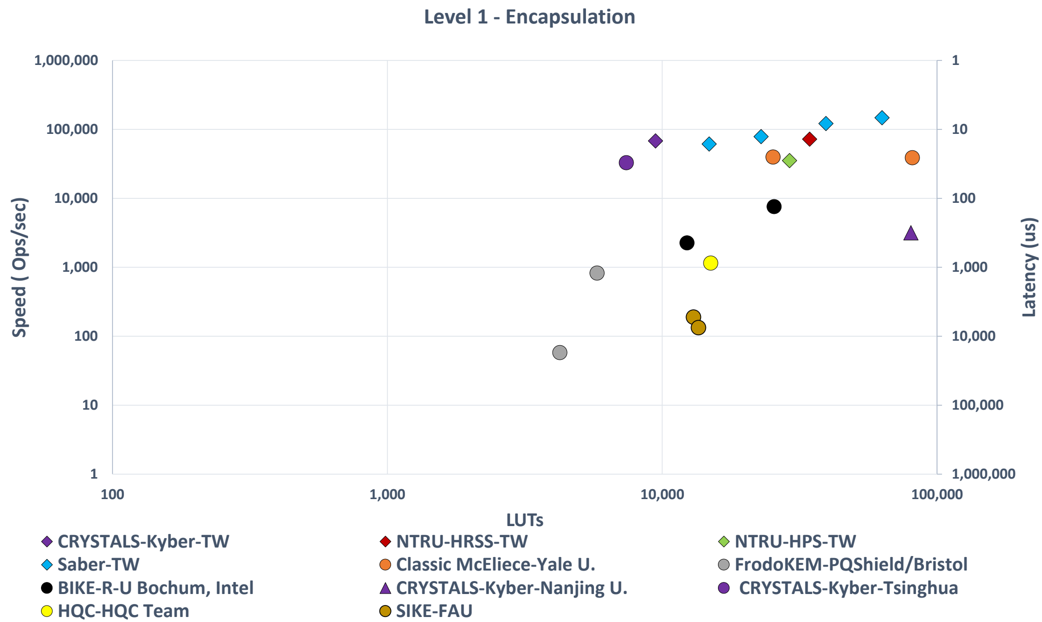

Figure 29: L1, Encaps, Artix-7

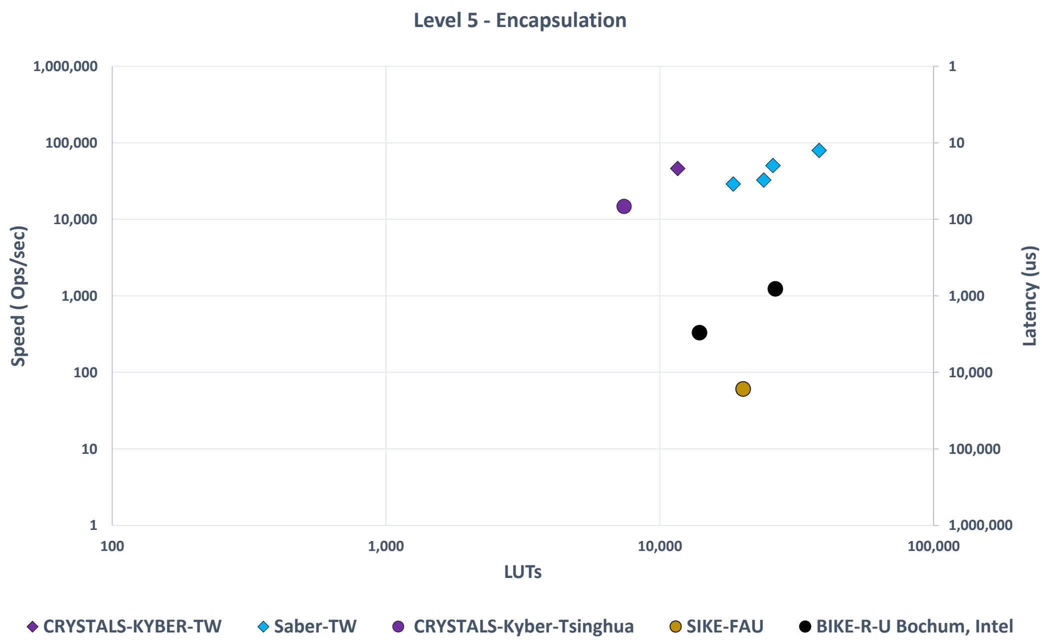

Figure 30: L5, Encaps, Artix-7

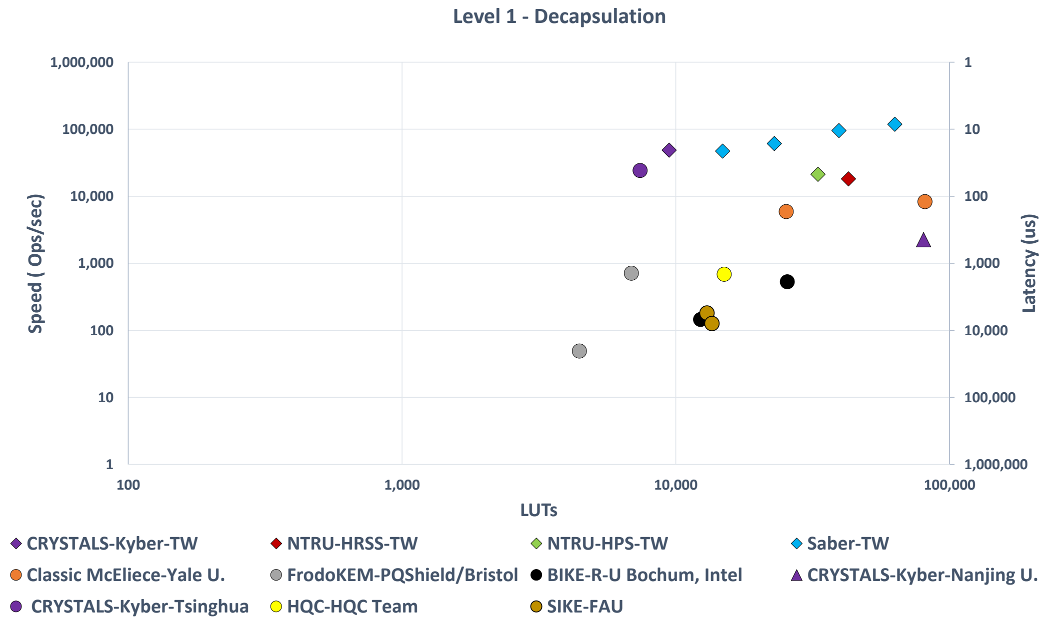

Figure 31: L1, Decaps, Artix-7

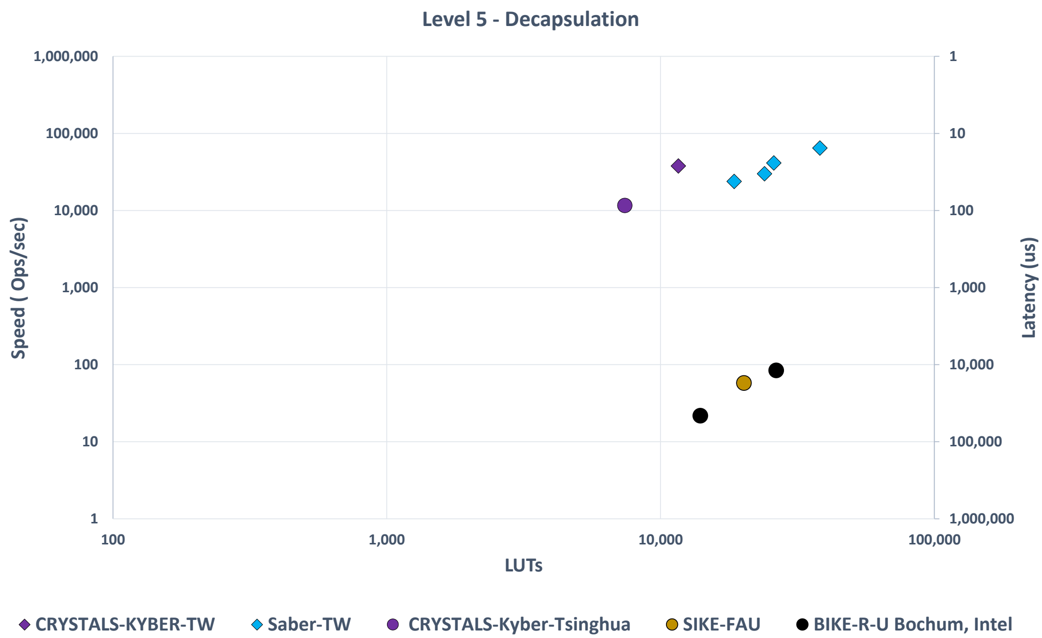

Figure 32: L5, Decaps, Artix-7

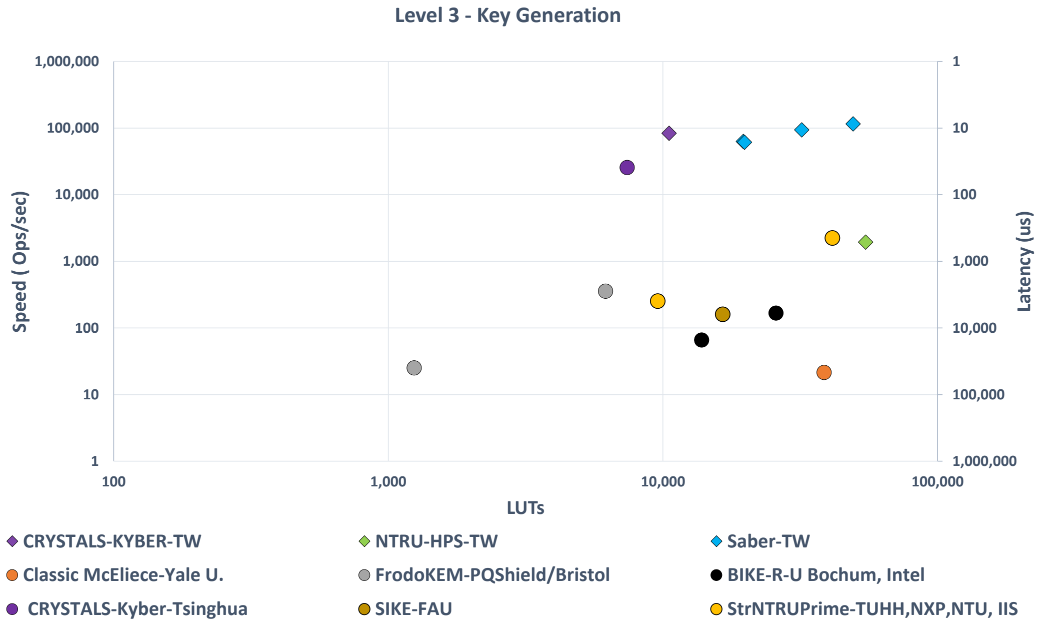

Figure 33: L3, KeyGen, Artix-7

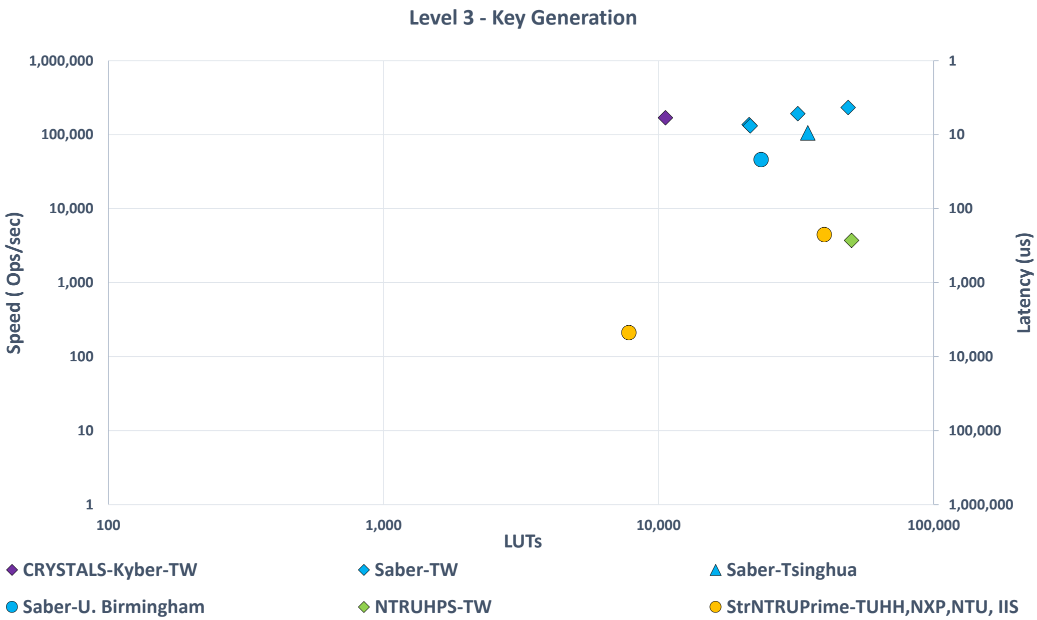

Figure 34: L3, KeyGen, Zynq UltraScale+

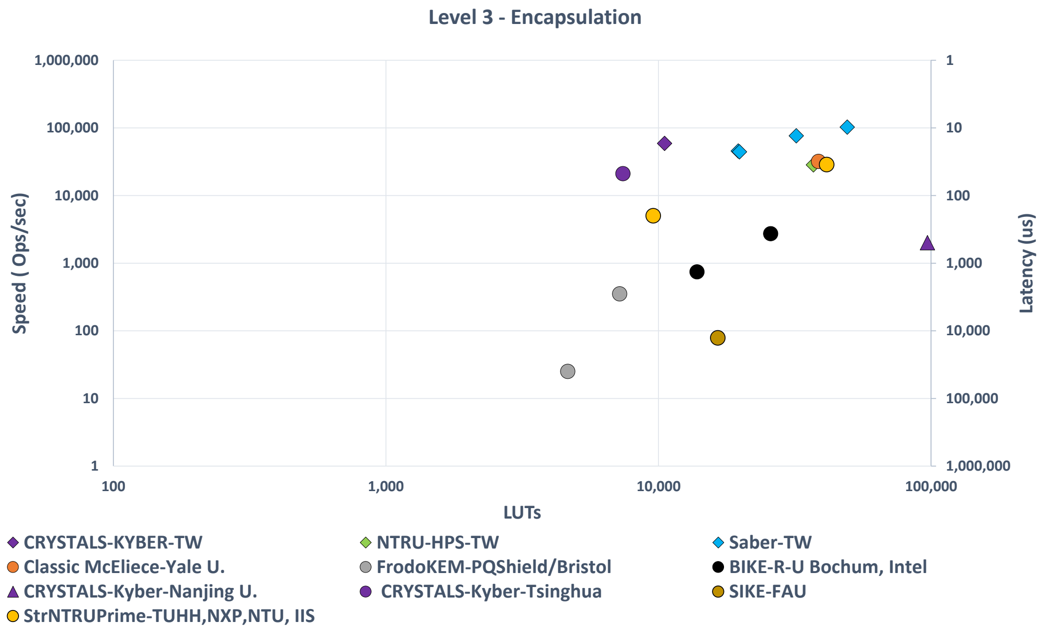

Figure 35: L3, Encaps, Artix-7

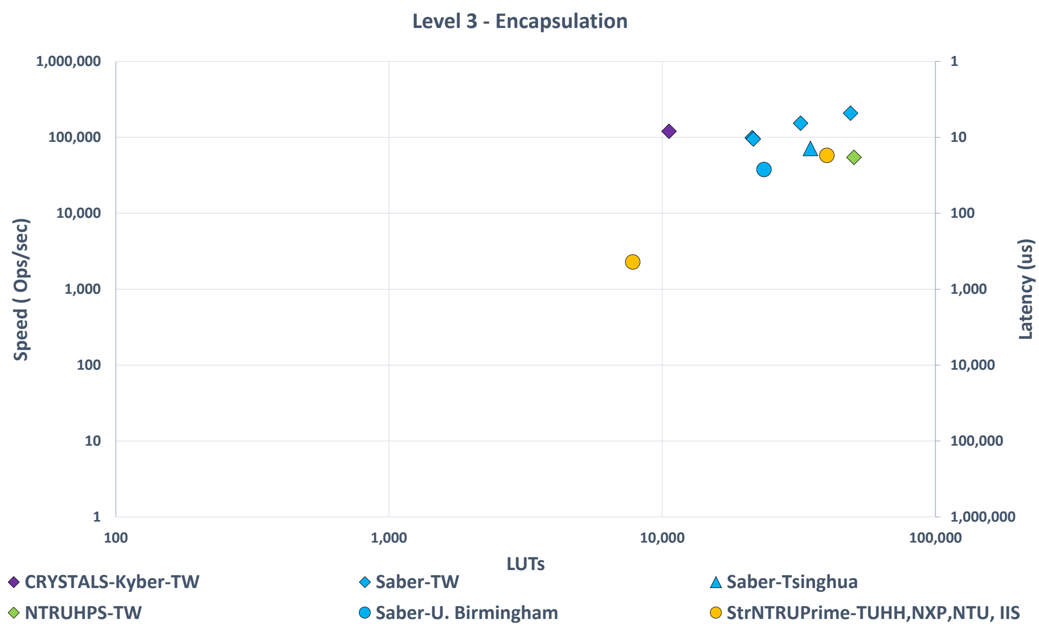

Figure 36: L3, Encaps, Zynq UltraScale+

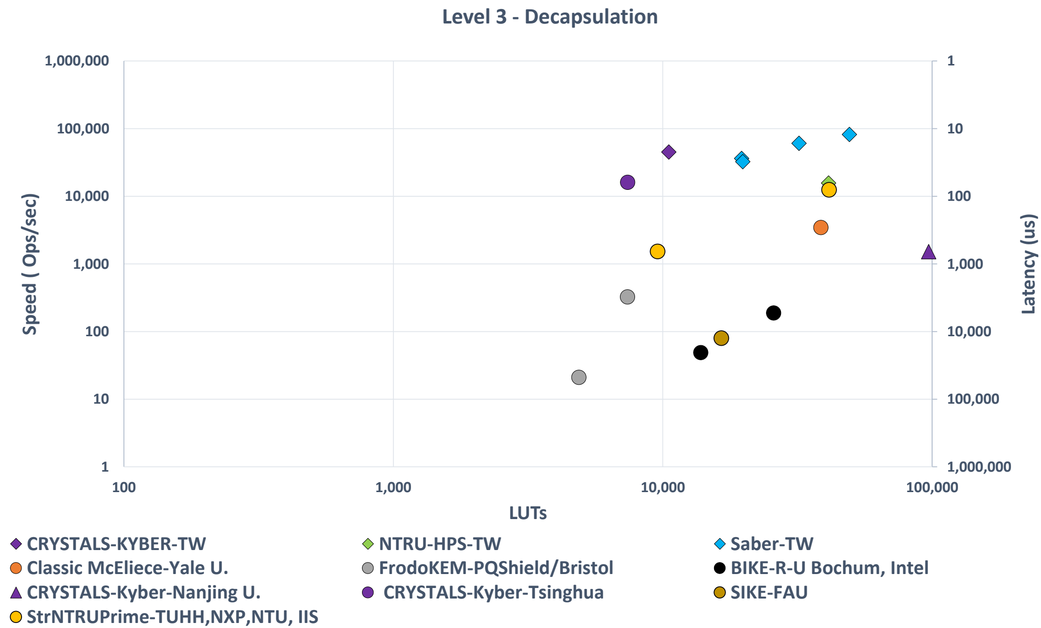

Figure 37: L3, Decaps, Artix-7

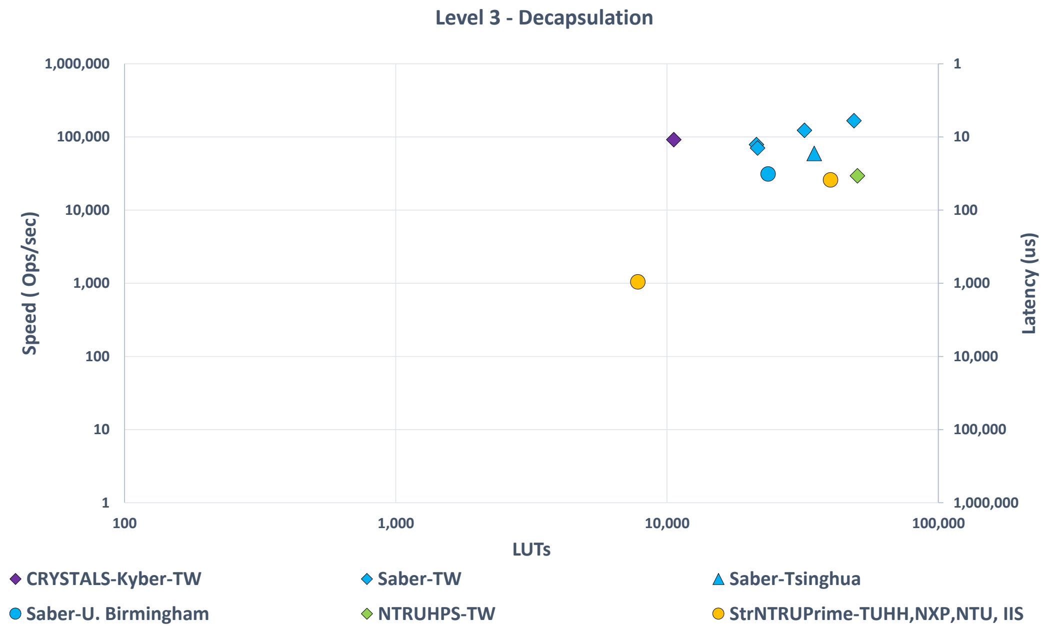

Figure 38: L3, Decaps, Zynq UltraScale+

Key generation in HQC is similar in terms of speed to NTRU and the third fastest overall. However, its encapsulation and decapsulation are about two orders of magnitude slower than for Saber and comparable in speed to the fastest reported implementation of FrodoKEM. Overall, four finalists – Saber, Kyber, NTRU, and Classic McEliece – clearly outperform four alternates – FrodoKEM, BIKE, HQC, and SIKE. Among the finalists, Saber and Kyber perform overall much better than NTRU and Classic McEliece.

The results for the security level 5 are shown in Figs. 28, 30, and 32. The majority of

Table 14: Artix-7 results for designs proposed and documented in this work

Key Generation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber x1	9.5	1.00	Kyber	12.0	1.00	Kyber	16.2	1.00
Kyber	10.0	1.05	Saber x1	15.9	1.33	Saber x1	28.8	1.78
NTRU-HRSS	323.8	34.08	NTRU-HPS	516.6	43.05			
NTRU-HPS	370.6	39.01						

Encapsulation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber x1	12.7	1.00	Kyber	17.0	1.00	Kyber	21.7	1.00
NTRU-HRSS	13.9	1.09	Saber x1	22.0	1.29	Saber x1	34.5	1.59
Kyber	14.7	1.16	NTRU-HPS	35.2	2.07			
NTRU-HPS	28.4	2.24						

Decapsulation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber x1	16.4	1.00	Kyber	22.2	1.00	Kyber	26.4	1.00
Kyber	20.5	1.25	Saber x1	27.5	1.24	Saber x1	41.9	1.59
NTRU-HPS	47.0	2.87	NTRU-HPS	63.8	2.87			
NTRU-HRSS	55.2	3.37						

Round 3 candidates either do not have implementations, or these implementations have exceeded the resources of Artix-7 FPGAs. From left to right, Saber is represented by four diamonds corresponding to the x1, NTT, x2, and x4 architecture, respectively. Kyber and Saber are in a virtual tie, with the Kyber speed matching that of Saber x2 and its area more than two times smaller.

For the security level 3, we present results for both Artix-7 and Zynq UltraScale+ in Figs 33–38. In the case of Artix-7, results are reported for all four finalists and four alternates (Streamlined NTRU Prime, FrodoKEM, BIKE, and SIKE). In the case of Zynq UltraScale+, the graphs cover three lattice-based finalists and one alternate candidate, NTRU Prime. The results for Saber-NTT are very similar to the results for Saber x1; their points in the graphs almost overlap. For all operations, at security level 3, the speed of Kyber is comparable to the speed of Saber x2, but its area in LUTs is about 3 times smaller. NTRU (represented at this level only by NTRU-HPS) is more than an order of magnitude slower than both Saber and Kyber for key generation and 2-3 times slower for encapsulation and decapsulation. Classic McEliece slightly exceeds the speed of NTRU for encapsulation but lags behind by almost an order of magnitude for decapsulation and two orders of magnitude for key generation. The high-speed implementation of Streamlined NTRU Prime reported in [76] has its speed and area in LUTs comparable to NTRU. FrodoKEM, SIKE, and BIKE are orders of magnitude slower than finalists for encapsulation and decapsulation and better only than Classic McEliece for key generation.

The results obtained using Zynq UltraScale+ (or UltraScale+) indicate that Streamlined NTRU Prime has a similar speed to NTRU and slightly better LUT usage.

In Tables 14 and 15, the exact numerical results are presented for the execution times of implementations proposed and described in this paper. In these tables, Saber is represented by Saber x1, as this design has the area closest to the area of CRYSTALS-Kyber. These results clearly indicate that NTRU is between 30 and 50 times slower than Saber for the key generation at both level 1 and level 3. NTRU is also about 2-4 times slower than Saber for decapsulation. Only for encapsulation, the performance of NTRU becomes comparable. Kyber is between 5% and 32% slower than Saber x1 at level 1. It outperforms Saber x1 (but not Saber x2 or Saber x4) in all rankings at levels 3 by a factor ranging between 17% and 33%. At level 5, the advantage of Kyber increases to the range 50%-80%. The reasons

Table 15: Zynq UltraScale+ results for designs proposed and documented in this work

Key Generation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber x1	4.3	1.00	Kyber	5.9	1.00	Kyber	7.9	1.00
Kyber	4.9	1.14	Saber x1	7.3	1.24	Saber x1	13.2	1.67
NTRU-HRSS	172.7	40.16	NTRU-HPS	268.6	44.81			
NTRU-HPS	192.7	48.18						

Encapsulation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber x1	5.8	1.00	Kyber	8.3	1.00	Kyber	10.6	1.00
Kyber	7.2	1.24	Saber x1	10.1	1.22	Saber x1	15.9	1.50
NTRU-HRSS	7.4	1.28	NTRU-HPS	18.3	1.81			
NTRU-HPS	14.7	2.53						

Decapsulation								
Level 1			*Level 3*			*Level 5*		
Algorithm	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**	**Algorithm**	**Time [us]**	**Ratio**
Saber x1	7.6	1.00	Kyber	10.9	1.00	Kyber	12.9	1.00
Kyber	10.0	1.32	Saber x1	12.7	1.17	Saber x1	19.3	1.50
NTRU-HPS	25.1	3.30	NTRU-HPS	34.0	3.12			
NTRU-HRSS	29.4	3.87						

for the change in the ranking of Kyber and Saber depending on the security level are as follows. In Kyber, the NTT-based multiplier is quite small and sequential. Therefore, it is justifiable to use 2, 3, and 4 multipliers at the security levels 1, 3, and 5, respectively (as described in Section 5.2). In Saber, the schoolbook multiplier is big and parallel. Therefore, increasing the number of multipliers is not justifiable, as a small increase in speed causes a large increase in area. On the other hand, in Kyber, the NTT multiplier is relatively small. Consequently, we scaled the number of multipliers proportionally to the parameter k of Kyber, equal to 2, 3, and 4 for security levels 1, 3, and 5, respectively. As a result, the relative performance of Kyber, as compared to Saber x1, has increased at higher security levels, while its area stayed significantly below the area for Saber.

7 Conclusions

In this paper, we have proposed, documented, and benchmarked a) the first complete hardware implementations of two variants of NTRU (NTRU-HRSS and NTRU-HPS), as defined in the submissions to Rounds 2 and 3 of the NIST PQC standardization process; b) four high-speed implementations of Saber, with two of them outperforming competing designs in terms of speed and two in terms of resource utilization, and c) the fastest implementation of CRYSTALS-Kyber. All designs are fully reproducible, and their source code will be released as open-source after the acceptance of this paper to a journal or a conference with proceedings.

We also have reviewed the related literature and collected information about hardware implementations of all Round 3 candidates in the category of Key Encapsulation Mechanisms (KEMs). Our analysis reveals that four NIST PQC finalists significantly outperform almost all alternate candidates when implemented in hardware with speed as a primary optimization target. Among the four finalists, Saber and CRYSTALS-Kyber significantly outperform NTRU and Classic McEliece for at least a subset of all operations. The differences between the two top candidates are relatively minor and may change as a result of future optimizations.

8 Future Work

Work presented in this paper could be extended with the attempts at implementing CRYSTALS-Kyber and Saber using an NTT-based multiplier with a higher radix. Additionally, using well-known techniques such as folding, resource sharing, and Domain-Oriented Masking (DOM), high-speed implementations resistant against timing attacks could be converted to lightweight implementations resistant against side-channel analysis. After the end of Round 3, during the expected Round 4, our focus will shift toward evaluating candidates qualified for that round. These candidates will likely include a significant subset of Round 3 alternates.

References

1. Alagic, G., Alperin-Sheriff, J., Apon, D., Cooper, D., Dang, Q., Liu, Y.-K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., and Smith-Tone, D.: Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process. Tech. rep. NIST IR 8240, Gaithersburg, MD: National Institute of Standards and Technology (2019)
2. Alagic, G., Apon, D.C., Cooper, D.A., Dang, Q.H., Kelsey, J.M., Liu, Y.-K., Miller, C.A., Moody, D., Peralta, R.C., Perlner, R.A., Robinson, A.Y., Smith-Tone, D.C., and Alperin-Sheriff, J.: Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. Tech. rep. NISTIR 8309, Gaithersburg, MD: National Institute of Standards and Technology (2020)
3. Alkim, E., Evkan, H., Lahr, N., Niederhagen, R., and Petri, R.: ISA Extensions for Finite Field Arithmetic - Accelerating Kyber and NewHope on RISC-V. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(3), 219–242 (2020)
4. ARM: AMBA 4 AXI4-Stream Protocol Specification. Tech. rep., (2010)
5. Atici, A.C., Batina, L., Junfeng Fan, Verbauwhede, I., and Berna Ors Yalcin, S.: Low-Cost Implementations of NTRU for Pervasive Security. In: 2008 International Conference on Application-Specific Systems, Architectures and Processors, pp. 79–84. IEEE, Leuven, Belgium (2008)
6. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K.: Chisel: Constructing Hardware in a Scala Embedded Language. In: DAC Design Automation Conference 2012, pp. 1212–1221 (2012)
7. Bailey, D.V., Coffin, D., Elbirt, A., Silverman, J.H., and Woodbury, A.D.: NTRU in Constrained Devices. In: Cryptographic Hardware and Embedded Systems — CHES 2001. Ed. by Ç.K. Koç, D. Naccache, and C. Paar, pp. 262–272. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
8. Baldwin, B., Byrne, A., Lu, L., Hamilton, M., Hanley, N., O'Neill, M., and Marnane, W.P.: FPGA Implementations of the Round Two SHA-3 Candidates. In: 2010 International Conference on Field Programmable Logic and Applications, FPL 2010, pp. 400–407, Milan, Italy (2010)
9. Banerjee, U., Ukyab, T.S., and Chandrakasan, A.P.: Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-Based Protocols. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(4) (2019)
10. Banerjee, U., Ukyab, T.S., and Chandrakasan, A.P.: Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-Based Protocols (Extended Version). Cryptology ePrint Archive 2019/1140, (2020)
11. Barrett, P.: Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO' 86. Lecture Notes in Computer Science, pp. 311–323. Springer, Berlin, Heidelberg (1987)
12. Basso, A., and Roy, S.S.: Optimized Polynomial Multiplier Architectures for Post-Quantum KEM Saber. In: 58th Design Automation Conference, DAC 2021, San Francisco (2021)
13. Basu, K., Soni, D., Nabeel, M., and Karri, R.: NIST Post-Quantum Cryptography- A Hardware Evaluation Study. Cryptology ePrint Archive 2019/047, (2019)
14. Bernstein, D.J., Heninger, N., Lou, P., and Valenta, L.: Post-Quantum RSA. In: 8th International Workshop on Post-Quantum Cryptography, PQCrypto 2017. Lecture Notes in Computer Science, pp. 312–329. Springer International Publishing, Cham (2017)
15. Bernstein, D.J., and Yang, B.-Y.: Fast Constant-Time Gcd Computation and Modular Inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(3), 340–398 (2019)
16. Bodrato, M., and Zanoni, A.: Integer and Polynomial Multiplication: Towards Optimal Toom-Cook Matrices. In: International Symposium on Symbolic and Algebraic Computation, ISSAC 2007, pp. 17–24 (2007)
17. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness - Web Page*, https://competitions.cr.yp.to/caesar.html (2019). 2019
18. Chen, C., Danba, O., Rijneveld, J., Schanck, J.M., Saito, T., Schwabe, P., Whyte, W., Xagawa, K., Yamakawa, T., and Zhang, Z.: NTRU: Algorithm Specifications And Supporting Documentation, (2020)
19. Chung, C.-M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C.-J., and Yang, B.-Y.: NTT Multiplication for NTT-Unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(2), 159–188 (2021)
20. Cryptographic Engineering Research Group (CERG) at George Mason University: *Hardware Benchmarking of CAESAR Candidates*, https://cryptography.gmu.edu/athena/index.php?id=CAESAR (2019). 2019

21. Dang, V., Farahmand, F., Andrzejczak, M., Mohajerani, K., Nguyen, D.T., and Gaj, K.: Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-Design Approaches. Cryptology ePrint Archive 2020/795, p. 86 (2020)

22. Dang, V.B., Farahmand, F., Andrzejczak, M., and Gaj, K.: Implementing and Benchmarking Three Lattice-Based Post-Quantum Cryptography Algorithms Using Software/Hardware Codesign. In: 2019 International Conference on Field Programmable Technology, FPT 2019, pp. 206–214. IEEE, Tianjin, China (Dec. 9-13, 2019)

23. de Dinechin, F.: Reflections on 10 Years of FloPoCo. In: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), pp. 187–189. IEEE, Kyoto, Japan (2019)

24. Du, C., Bai, G., and Wu, X.: High-Speed Polynomial Multiplier Architecture for Ring-LWE Based Public Key Cryptosystems. In: Proceedings of the 26th Edition on Great Lakes Symposium on VLSI - GLSVLSI '16, pp. 9–14. ACM Press, Boston, Massachusetts, USA (2016)

25. Dubois, and Venetsanopoulos: The Discrete Fourier Transform Over Finite Rings with Application to Fast Convolution. IEEE Transactions on Computers **C-27**(7), 586–593 (1978)

26. Elkhatib, R., Azarderakhsh, R., and Mozaffari-Kermani, M.: High-Performance FPGA Accelerator for SIKE. IEEE Trans. Comput. (2021)

27. Farahmand, F., Dang, V.B., Nguyen, D.T., and Gaj, K.: Evaluating the Potential for Hardware Acceleration of Four NTRU-Based Key Encapsulation Mechanisms Using Software/Hardware Codesign. In: 10th International Conference on Post-Quantum Cryptography, PQCrypto 2019. LNCS, pp. 23–43. Springer, Chongqing, China (2019)

28. Farahmand, F., Ferozpuri, A., Diehl, W., and Gaj, K.: Minerva: Automated Hardware Optimization Tool. In: 2017 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017, pp. 1–8. IEEE, Cancun (2017)

29. Farahmand, F., Sharif, M.U., Briggs, K., and Gaj, K.: A High-Speed Constant-Time Hardware Implementation of NTRUEncrypt SVES. In: 2018 International Conference on Field-Programmable Technology, FPT 2018, pp. 190–197. IEEE, Naha, Okinawa, Japan (2018)

30. Farahmand, F., Sharif, M.U., Briggs, K., and Gaj, K.: A High-Speed Constant-Time Hardware Implementation of NTRUEncrypt SVES. In: 2018 International Conference on Field-Programmable Technology, FPT 2018, pp. 190–197. IEEE, Naha, Okinawa, Japan (2018)

31. Feng, X., Li, S., and Xu, S.: RLWE-Oriented High-Speed Polynomial Multiplier Utilizing Multi-Lane Stockham NTT Algorithm. IEEE Transactions on Circuits and Systems II: Express Briefs **67**(3), 556–559 (2020)

32. Fritzmann, T., Beirendonck, M.V., Roy, D.B., Karl, P., Schamberger, T., Verbauwhede, I., and Sigl, G.: Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography. Cryptology ePrint Archive 2021/479, (2021)

33. Fritzmann, T., Sigl, G., and Sepúlveda, J.: RISQ-V: Tightly Coupled RISC-V Accelerators for Post-Quantum Cryptography. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 239–280 (2020)

34. Fujisaki, E., and Okamoto, T.: Secure Integration of Asymmetric and Symmetric Encryption Schemes. Journal of Cryptology **26**(1), 80–101 (2013)

35. Gaj, K.: Challenges and Rewards of Implementing and Benchmarking Post-Quantum Cryptography in Hardware. In: 2018 Great Lakes Symposium on VLSI, GLSVLSI 2018, pp. 359–364. ACM Press, Chicago, IL, USA (2018)

36. Gaj, K., Homsirikamol, E., and Rogawski, M.: Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs. In: Cryptographic Hardware and Embedded Systems, CHES 2010. LNCS, pp. 264–278, Santa Barbara, CA (2010)

37. Gaj, K., Homsirikamol, E., Rogawski, M., Shahid, R., and Sharif, M.U.: Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs. Cryptology ePrint Archive 2012/368, (2012)

38. Gaj, K., Kaps, J.-P., Amirineni, V., Rogawski, M., Homsirikamol, E., and Brewster, B.Y.: ATHENa - Automated Tool for Hardware EvaluatioN: Toward Fair and Comprehensive Benchmarking of Cryptographic Hardware Using FPGAs. In: 2010 International Conference on Field Programmable Logic and Applications, FPL 2010, pp. 414–421. IEEE, Milan, Italy (2010)

39. Gambetta, J.: IBM's Roadmap For Scaling Quantum Technology, IBM Research Blog. (2020). https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/ (visited on 06/01/2021)

40. Hoffstein, J., Pipher, J., and Silverman, J.H.: NTRU: A Ring-Based Public Key Cryptosystem. In: Algorithmic Number Theory. Ed. by G. Goos, J. Hartmanis, J. van Leeuwen, and J.P. Buhler, pp. 267–288. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)

41. Hofheinz, D., Hövelmanns, K., and Kiltz, E.: A Modular Analysis of the Fujisaki-Okamoto Transformation. In: Theory of Cryptography. Ed. by Y. Kalai and L. Reyzin, pp. 341–371. Springer International Publishing, Cham (2017)

42. Homsirikamol, E., Yalla, P., Farahmand, F., Diehl, W., Ferozpuri, A., Kaps, J.-P., and Gaj, K.: Implementer's Guide to Hardware Implementations Compliant with the CAESAR Hardware API. GMU Report, Fairfax, VA: GMU (2016)

43. Homsirikamol, E., Diehl, W., Ferozpuri, A., Farahmand, F., Yalla, P., Kaps, J.-P., and Gaj, K.: CAESAR Hardware API. Cryptology ePrint Archive 2016/626, (2016)

44. Homsirikamol, E., and Gaj, K.: Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study. In: Applied Reconfigurable Computing - ARC 2015. LNCS, pp. 217–228. Springer International Publishing, Cham (2015)

45. Homsirikamol, E., and Gaj, K.: Toward a New HLS-Based Methodology for FPGA Benchmarking of Candidates in Cryptographic Competitions: The CAESAR Contest Case Study. In: 2017 International Conference on Field Programmable Technology, FPT 2017, pp. 120–127. IEEE, Melbourne, Australia (2017)

46. Homsirikamol, E., Yalla, P., and Farahmand, F.: Development Package for Hardware Implementations Compliant with the CAESAR Hardware API, v2.0, (2017). https://cryptography.gmu.edu/athena/index.php?id=CAESAR

47. Howe, J.: Optimised Lattice-Based Key Encapsulation in Hardware. In: Second NIST Post-Quantum Cryptography Standardization Conference 2019, p. 13 (2019)

48. Hu, J., Wang, W., Cheung, R.C., and Wang, H.: Optimized Polynomial Multiplier Over Commutative Rings on FPGAs: A Case Study on BIKE. In: 2019 International Conference on Field-Programmable Technology (ICFPT), pp. 231–234. IEEE, Tianjin, China (2019)

49. Huang, Y., Huang, M., Lei, Z., and Wu, J.: A Pure Hardware Implementation of CRYSTALS-KYBER PQC Algorithm through Resource Reuse. IEICE Electronics Express **17** (2020)

50. Hülsing, A., Rijneveld, J., Schanck, J., and Schwabe, P.: High-Speed Key Encapsulation from NTRU. In: Cryptographic Hardware and Embedded Systems – CHES 2017. Ed. by W. Fischer and N. Homma, pp. 232–252. Springer International Publishing, Cham (2017)

51. IEEE: *IEEE Standard Specification for Public Key Cryptographic Techniques Based on Hard Problems over Lattices, P1363.1-2008*, (2009). Mar. 2009

52. Izraelevitz, A., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., Kim, D., Schmidt, C., Markley, C., Lawson, J., and Bachrach, J.: Reusability Is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. In: Proceedings of the 36th International Conference on Computer-Aided Design. ICCAD '17, pp. 209–216. IEEE Press, Piscataway, NJ, USA (2017)

53. Kamal, A.A., and Youssef, A.M.: An FPGA Implementation of the NTRUEncrypt Cryptosystem. In: 2009 International Conference on Microelectronics - ICM, pp. 209–212. IEEE, Marrakech (2009)

54. Kaps, J.-P.: Cryptography for Ultra-Low Power Devices. Worcester Polytechnic Institute (2006)

55. Kaps, J.-P., Surapathi, K.K., Habib, B., Vadlamudi, S., Gurung, S., and Pham, J.: Lightweight Implementations of SHA-3 Candidates on FPGAs. In: 12th International Conference on Cryptology in India, Indocrypt 2011. LNCS, pp. 270–289, Chennai, India (2011)

56. Karatsuba, A., and Ofman, Y.: Multiplication of Many-Digital Numbers by Automatic Computers. Doklady Akademii Nauk SSSR **145**(2), 293–294 (1962)

57. Knezevic, M., Kobayashi, K., Ikegami, J., Matsuo, S., Satoh, A., Kocabas, Ü., Fan, J., Katashita, T., Sugawara, T., Sakiyama, K., Verbauwhede, I., Ohta, K., Homma, N., and Aoki, T.: Fair and Consistent Hardware Evaluation of Fourteen Round Two SHA-3 Candidates. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **20**(5), 827–840 (2012)

58. Knezevic, M., Vercauteren, F., and Verbauwhede, I.: Faster Interleaved Modular Multiplication Based on Barrett and Montgomery Reduction Methods. IEEE Transactions on Computers **59**(12), 1715–1721 (2010)

59. Koziel, B., Ackie, A.-B., El Khatib, R., Azarderakhsh, R., and Kermani, M.M.: SIKE'd Up: Fast Hardware Architectures for Supersingular Isogeny Key Encapsulation. IEEE Transactions on Circuits and Systems I: Regular Papers (2020)

60. Kumm, M., Gustafsson, O., Garrido, M., and Zipf, P.: Optimal Single Constant Multiplication Using Ternary Adders. IEEE Transactions on Circuits and Systems II: Express Briefs **65**(7), 928–932 (2018)

61. Kuo, P.-C., Li, W.-D., Chen, Y.-W., Hsu, Y.-C., Peng, B.-Y., Cheng, C.-M., and Yang, B.-Y.: High Performance Post-Quantum Key Exchange on FPGAs. Cryptology ePrint Archive 2017/690, p. 17 (2018)

62. Liu, B., and Wu, H.: Efficient Architecture and Implementation for NTRUEncrypt System. In: 2015 IEEE 58th International Midwest Symposium on Circuits and Systems, MWSCAS 2015, Fort Collins, CO, USA (2015)

63. Liu, B., and Wu, H.: Efficient Multiplication Architecture over Truncated Polynomial Ring for NTRUEncrypt System. In: 2016 IEEE International Symposium on Circuits and Systems, ISCAS 2016, pp. 1174–1177. IEEE, Montréal, QC, Canada (2016)

64. Liu, W., Fan, S., Khalid, A., Rafferty, C., and O'Neill, M.: Optimized Schoolbook Polynomial Multiplication for Compact Lattice-Based Cryptography on FPGA. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **27**(10), 2459–2463 (2019)

65. Longa, P., Naehrig, M., Longa, P., and Naehrig, M.: Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In: Cryptology and Network Security - CANS 2016, pp. 124–139. Springer International Publishing, Cham (2016)

66. Marotzke, A.: A Constant Time Full Hardware Implementation of Streamlined NTRU Prime. In: 19th International Conference on Smart Card Research and Advanced Applications, CARDIS 2020. LNCS, pp. 3–17. Springer International Publishing, Cham (2020)

67. Massolino, P.M.C., Longa, P., Renes, J., and Batina, L.: A Compact and Scalable Hardware/Software Co-Design of SIKE. IACR Transactions on Cryptographic Hardware and Embedded Systems (2020)

68. Mera, J.M.B., Turan, F., Karmakar, A., Sinha Roy, S., and Verbauwhede, I.: Compact Domain-Specific Co-Processor for Accelerating Module Lattice-Based KEM. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE, San Francisco, CA, USA (2020)

69. Montgomery, P.L.: Modular Multiplication without Trial Division. Mathematics of computation **44**(170), 519–521 (1985)

70. Nguyen, D.T., Dang, V.B., and Gaj, K.: A High-Level Synthesis Approach to the Software/Hardware Codesign of NTT-Based Post-Quantum Cryptography Algorithms. In: 2019 International Conference on Field-Programmable Technology (ICFPT), pp. 371–374. IEEE, Tianjin, China (2019)

71. Nguyen, D.T., Dang, V.B., and Gaj, K.: High-Level Synthesis in Implementing and Benchmarking Number Theoretic Transform in Lattice-Based Post-Quantum Cryptography Using Software/Hardware Codesign. In: 16th International Symposium on Applied Reconfigurable Computing, ARC 2020 (2020)

72. NIST: *Post-Quantum Cryptography: Call for Proposals*, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals (2019). 2019

73. NTRU Submission Team: Round 2 Submissions - NTRU Candidate Submission Package, Post-Quantum Cryptography: Round 2 Submissions. (2019)

74. O'Rourke, C.M.: Efficient NTRU Implementations. Worcester Polytechnic Institute (2002)

75. Oder, T., and Guneysu, T.: Implementing the NewHope-Simple Key Exchange on Low-Cost FPGAs. In: LATINCRYPT 2017, Havana, Cuba (2017)

76. Peng, B.-Y., Marotzke, A., Tsai, M.-H., Yang, B.-Y., and Chen, H.-L.: Streamlined NTRU Prime on FPGA. Cryptology ePrint Archive 2021/1444, p. 31 (2021)

77. Pöppelmann, T., Güneysu, T., Pöppelmann, T., and Güneysu, T.: Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In: Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Hevia, A., and Neven, G. (eds.) Progress in Cryptology – LATINCRYPT 2012. LNCS, pp. 139–158. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). https://www.seceng.ruhr-uni-bochum.de/media/attachments/files/2012/07/lattice_fft.zip

78. *Post-Quantum Cryptography: Round 1 Submissions*, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions (2019). Apr. 2019

79. Reinders, A.H., Misoczki, R., Ghosh, S., and Sastry, M.R.: Efficient BIKE Hardware Design with Constant-Time Decoder. In: 2020 IEEE International Conference on Quantum Computing and Engineering (QCE), pp. 197–204. IEEE, Denver, CO, USA (2020)

80. Renteria-Mejia, C.P., and Velasco-Medina, J.: High-Throughput Ring-LWE Cryptoprocessors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(8), 2332–2345 (2017)

81. Richter-Brockmann, J., Chen, M.-S., Ghosh, S., and Güneysu, T.: Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware. Cryptology ePrint Archive 2021/1344, (2021)

82. Richter-Brockmann, J., Mono, J., and Guneysu, T.: Folding BIKE: Scalable Hardware Implementation for Recongurable Devices. IEEE Trans. Comput. (2021)

83. *Round 3 Submissions - HQC Candidate Submission Package*, https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/HQC-Round3.zip (2021). June 2021

84. Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., Verbauwhede, I., Roy, S.S., Vercauteren, F., Mentens, N., Chen, D.D., and Verbauwhede, I.: Compact Ring-LWE Cryptoprocessor. In: Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Salinesi, C., Norrie, M.C., and Pastor, Ó. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2014. LNCS, pp. 371–391. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)

85. Saito, T., Xagawa, K., and Yamakawa, T.: Tightly-Secure Key-Encapsulation Mechanism in the Quantum Random Oracle Model. In: Advances in Cryptology – EUROCRYPT 2018. Lecture Notes in Computer Science, pp. 520–551. Springer International Publishing, Cham (2018)

86. Shor, P.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: Proceedings 35th Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE Comput. Soc. Press, Santa Fe, NM, USA (1994)

87. Sinha Roy, S., and Basso, A.: High-Speed Instruction-Set Coprocessor for Lattice-Based Key Encapsulation Mechanism: Saber in Hardware. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(4), 443–466 (2020)

88. Sinha Roy, S., and Verbauwhede, I.: Lattice-Based Public-Key Cryptography in Hardware. Springer Singapore, Singapore (2020)

89. Toom, A.: The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. Soviet Mathematics Doklady **3**, 714–716 (1963)

90. Vandersypen, L.: *A "Spins-inside" Quantum Computer*, Invited Talk (2017). Utrecht, the Netherlands, June 2017

91. Wang, W., Szefer, J., and Niederhagen, R.: FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes. In: Lange, T., and Steinwandt, R. (eds.) 9th International Conference on Post-Quantum Cryptography, PQCrypto 2018. LNCS, pp. 77–98. Springer International Publishing, Fort Lauderdale, Florida (2018)

92. Wang, W., Szefer, J., Niederhagen, R., Szefer, J., and Niederhagen, R.: FPGA-Based Key Generator for the Niederreiter Cryptosystem Using Binary Goppa Codes. In: Cryptographic Hardware and Embedded Systems, CHES 2017. LNCS, pp. 253–274. Springer International Publishing, Cham (2017)

93. Wang, W., Tian, S., Jungk, B., Bindel, N., Longa, P., and Szefer, J.: Parameterized Hardware Accelerators for Lattice-Based Cryptography and Their Application to the HW/SW Co-Design of qTESLA. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(3), 269–306 (2020)

94. Xin, G., Han, J., Yin, T., Zhou, Y., Yang, J., Cheng, X., and Zeng, X.: VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture. IEEE Transactions on Circuits and Systems I: Regular Papers **67**(8), 1–13 (2020)

95. Xing, Y., and Li, S.: A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(2), 328–356 (2021)

96. Ye, J.-H., and Shieh, M.-D.: High-Performance NTT Architecture for Large Integer Multiplication. In: 2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), pp. 1–4 (2018)

97. Zhang, N., Yang, B., Chen, C., Yin, S., Wei, S., and Liu, L.: Highly Efficient Architecture of NewHope-NIST on FPGA Using Low-Complexity NTT/INTT. IACR Transactions on Cryptographic Hardware and Embedded Systems (2020)

98. Zhu, Y., Zhu, M., Yang, B., Zhu, W., Deng, C., Chen, C., Wei, S., and Liu, L.: LWRpro: An Energy-Efficient Configurable Crypto-Processor for Module-LWR. IEEE Transactions on Circuits and Systems I: Regular Papers **68**(3), 1146–1159 (2021)

A Pseudocode of NTRU

Algorithm 4 NTRU PKE Keypair

Input: fg_bits
Output: $pk = packed_h$ and $sk = (packed_f, packed_f_p, packed_h_q)$

1: $(f, g) \leftarrow \text{Sample}(fg_bits) \bmod (3, \Phi_n)$
2: $f_p \leftarrow f^{-1} \bmod (3, \Phi_n)$
3: $G \leftarrow 3 \cdot g$
4: $v_0 \leftarrow (G \cdot f) \bmod (q, \Phi_n)$
5: $v_1 \leftarrow v_0^{-1} \bmod (q, \Phi_n)$
6: $h \leftarrow (v_1 \cdot G \cdot G) \bmod (q, \Phi_1\Phi_n)$
7: $h_q \leftarrow (v_1 \cdot f \cdot f) \bmod (q, \Phi_1\Phi_n)$
8: $sk \leftarrow (\text{pack_}\epsilon_p(f), \text{pack_}\epsilon_p(f_p), \text{pack_}\epsilon_q(h_q))$
9: $pk \leftarrow \text{pack_}\epsilon_q(h)$

Algorithm 5 NTRU KEM Keypair

Input: Random seed $seeds$
Output: $pk = packed_h$ and $sk = (packed_f, packed_f_p, packed_h_q, s)$

1: $(fg_bits, prf_key) \leftarrow \text{SHAKE128}(seeds)$
2: $(packed_h, packed_f, packed_f_p,$
3: $packed_h_q) \leftarrow \text{PKE.KeyPair}(fg_bits)$
4: $sk \leftarrow (packed_f || packed_f_p || packed_h_q || bits_to_bytes(prf_key))$
5: $pk \leftarrow packed_h$

Algorithm 6 NTRU PKE Encryption

Input: $pk = packed_h$, r and m
Output: $packed_c$

1: $m' \leftarrow \text{Lift}(m)$
2: $h \leftarrow \text{unpack_}\epsilon_q(packed_h)$
3: $c \leftarrow (r \cdot h + m') \bmod (q, \Phi_1\Phi_n)$
4: $packed_c \leftarrow \text{pack_}\epsilon_q(c)$

Algorithm 8 NTRU KEM Encapsulation

Input: $pk = packed_h$ and $seed$
Output: $packed_c$ and shared key K

1: $seed_{rm} \leftarrow \text{SHAKE128}(seed)$
2: $(r, m) \leftarrow \text{Sample}(seed_{rm}) \bmod (3, \Phi_n)$
3: $packed_c \leftarrow \text{PKE.Encrypt}\ (pk, (r, m))$
4: $packed_rm \leftarrow (\text{pack_}\epsilon_p(r) || \text{pack_}\epsilon_p(m))$
5: $K \leftarrow \text{SHA3-256}(packed_rm)$

Algorithm 7 NTRU DPKE Decryption

Input: $sk = (packed_f, packed_f_p, packed_h_q)$ and $packed_c$
Output: $r, m, fail$

1: if $c \not\equiv 0 \pmod{(q, \Phi_1)}$ return $(0, 0, 1)$
2: $c \leftarrow \text{unpacked_}\epsilon_q(packed_c)$
3: $f \leftarrow \text{unpacked_}\epsilon_p(packed_f)$
4: $a' \leftarrow (c \cdot f) \bmod (q, \Phi_1\Phi_n)$
5: $a \leftarrow \mathsf{R_q_to_S_3}(a')$
6: $f_p \leftarrow \text{unpacked_}\epsilon_p(packed_f_p)$
7: $m \leftarrow (a \cdot f_p) \bmod (3, \Phi_n)$
8: $h_q \leftarrow \text{unpacked_}\epsilon_q(packed_h_q)$
9: $m' \leftarrow \text{Lift}(m)$
10: $r \leftarrow ((c - m') \cdot h_q) \bmod (q, \Phi_n)$
11: if (r, m) valid return $(r, m, 0)$ else return $(0, 0, 1)$

Algorithm 9 NTRU KEM Decapsulation

Input: $sk = (packed_f, packed_f_p, packed_h_q, s)$ and $packed_c$
Output: Shared key K

1: $(r, m, fail) \leftarrow \text{PKE.Decrypt}((packed_f, packed_f_p, packed_h_q), packed_c)$
2: $packed_rm \leftarrow (\text{pack_}\epsilon_p(r) || \text{pack_}\epsilon_p(m))$
3: $k_1 \leftarrow \text{SHA3-256}(packed_rm)$
4: $k_2 \leftarrow \text{SHA3-256}(s || packed_c)$
5: **if** $fail == 0$ **then**
6: $\quad K \leftarrow k_1$
7: **else**
8: $\quad K \leftarrow k_2$
9: **end if**

B Pseudocode of CRYSTALS-Kyber

Here is the meaning of notation used in the given below algorithms:

- CBD_η: Sample from centered binomial distribution $\eta \in \{2, 3\}$.
- Rej: Sample from uniform distribution using rejection sampling.
- Decode: Deserialize a byte array into a polynomial or a polynomial-vector.
- Encode: Serialize polynomial or polynomial vector with coefficients in $[0, q-1]$ into an array of bytes.
- Compress(x, d): Compress a polynomial or polynomial-vector by mapping coefficients $x \in \mathbb{Z}_q$ to integers in $0, \ldots, 2^d - 1$, where $d < \lceil log_2(q) \rceil$
- Decompress(x, d): Decompress a polynomial or polynomial-vector x such that coefficients of $x' = \text{Decompress}(\text{Compress}(x, d), d)$ are close to corresponding coefficients in x.

Algorithm 10 Kyber PKE.KeyGen

Input: Uniform random *seed*
Outputs: PKE keys (pk, sk)

1: $(\rho||\sigma) \leftarrow \text{SHA3-512}(seed)$
2: **for** i from 0 to $k-1$ **do**
3:     **for** j from 0 to $k-1$ **do**
4:         $\hat{\mathbf{A}}[i][j] \leftarrow \text{Rej}(\text{SHAKE128}(\rho||j||i))$
5: **for** i from 0 to $k-1$ **do**
6:     $\mathbf{s}[i] \leftarrow \text{CBD}_{\eta_1}(\text{SHAKE256}(\sigma||i))$
7: **for** i from 0 to $k-1$ **do**
8:     $\mathbf{e}[i] \leftarrow \text{CBD}_{\eta_1}(\text{SHAKE256}(\sigma||(i+k)))$
9: $\hat{\mathbf{s}} \leftarrow \text{NTT}(\mathbf{s})$
10: $\hat{\mathbf{e}} \leftarrow \text{NTT}(\mathbf{e})$
11: $\hat{\mathbf{t}} \leftarrow (\hat{\mathbf{A}} \circ \hat{\mathbf{s}} + \hat{\mathbf{e}})$
12: $pk \leftarrow \text{Encode}(\hat{\mathbf{t}}) \;||\; \rho$
13: $sk \leftarrow \text{Encode}(\hat{\mathbf{s}})$

Algorithm 11 Kyber PKE.Enc

Inputs: Public key pk, message m, random *coins*
Output: Ciphertext ct

1: $(pk'||\rho) \leftarrow pk$
2: $\hat{\mathbf{t}} \leftarrow \text{Decode}(pk')$
3: **for** i from 0 to $k-1$ **do**
4:     **for** j from 0 to $k-1$ **do**
5:         $\hat{\mathbf{A}}^T[i][j] \leftarrow \text{Rej}(\text{SHAKE128}(\rho||i||j))$
6: **for** i from 0 to $k-1$ **do**
7:     $\mathbf{r}[i] \leftarrow \text{CBD}_{\eta_1}(\text{SHAKE256}(coins||i))$
8: **for** i from 0 to $k-1$ **do**
9:     $\mathbf{e}_1[i] \leftarrow \text{CBD}_{\eta_2}(\text{SHAKE256}(coins||i+k))$
10: $e_2 \leftarrow \text{CBD}_{\eta_2}(\text{SHAKE256}(coins||2k))$
11: $\hat{\mathbf{r}} \leftarrow \text{NTT}(\mathbf{r})$
12: $\mathbf{u} \leftarrow \text{NTT}^{-1}(\hat{\mathbf{A}}^T \circ \hat{\mathbf{r}}) + \mathbf{e}_1$
13: $v \leftarrow \text{NTT}^{-1}(\hat{\mathbf{t}}^T \circ \hat{\mathbf{r}}) + e_2 + \text{Decompress}(m, 1)$
14: $c_1 \leftarrow \text{Compress}(\mathbf{u}, d_u)$
15: $c_2 \leftarrow \text{Compress}(v, d_v)$
16: $ct \leftarrow (c_1||c_2)$

Algorithm 12 Kyber PKE.Dec

Inputs: Secret key sk, Ciphertext ct
Output: Message m

1: $(c_1||c_2) \leftarrow ct$
2: $\mathbf{u} \leftarrow \text{Decompress}(c_1, d_u)$
3: $v \leftarrow \text{Decompress}(c_2, d_v)$
4: $\hat{\mathbf{s}} \leftarrow \text{Decode}(sk)$
5: $\hat{\mathbf{u}} \leftarrow \text{NTT}(\mathbf{u})$
6: $\mu \leftarrow v - \text{NTT}^{-1}(\hat{\mathbf{s}} \circ \hat{\mathbf{u}})$
7: $m \leftarrow \text{Compress}(\mu, 1)$

Algorithm 13 Kyber KEM KeyGen

Inputs: random *seed*, random z
Outputs: Public key pk, Secret key sk_{KEM}

1: $(pk, sk') \leftarrow \text{PKE.KeyGen}(seed)$
2: $h_{pk} \leftarrow \text{SHA3-256}(pk)$
3: $sk_{\text{KEM}} \leftarrow (sk'||pk||h_{pk}||z)$

Algorithm 14 KYBER KEM Encapsulation

Input: Public key pk, m'
Output: Ciphertext ct, shared secret ss

1: $m \leftarrow$ SHA3-256(m')
2: $h_{pk} \leftarrow$ SHA3-256(pk)
3: $(\bar{ss}||coins) \leftarrow$ SHA3-512$(m||h_{pk})$
4: $ct \leftarrow$ PKE.Enc$(pk, m, coins)$
5: $h_{ct} \leftarrow$ SHA3-256(ct)
6: $ss \leftarrow$ SHAKE256$(\bar{ss}||h_{ct})$

Algorithm 15 KYBER KEM Decapsulation

Inputs: Ciphertext ct, Secret key sk_{KEM}
Output: Shared secret $ss \in \{0,1\}^{256}$

1: $(sk'||pk||h_{pk}||z) \leftarrow sk_{\text{KEM}}$
2: $m' \leftarrow$ PKE.Dec(sk', ct)
3: $h_{ct} \leftarrow$ SHA3-256(ct)
4: $(\bar{ss}||coins) \leftarrow$ SHA3-512$(m'||h_{pk})$
5: $ct' \leftarrow$ PKE.Enc$(pk, m', coins)$
6: $h_{ct'} \leftarrow$ SHA3-256(ct')
7: **if** $h_{ct} = h_{ct'}$ **:**
8: $\quad s_0 \leftarrow \bar{ss}$
9: **else:**
10: $\quad s_0 \leftarrow z$
11: $ss \leftarrow$ SHAKE256$(s_0||h_{ct})$

C Pseudocode of Saber

Algorithm 16 Saber PKE Keypair

Input: $seed_A$ and $seed_s$
Output: $pk = (seed_A, packed_b), sk = (packed_s)$

1: $seed_A \leftarrow \text{SHAKE128}(seed_A)$
2: $A \leftarrow \text{Unpack_}\epsilon_q(\text{SHAKE128}(seed_A)) \in R_q^{l\times l}$
3: $s \leftarrow \text{CBD}_\mu(\text{SHAKE128}(seed_s)) \in R_q^{l\times 1}$
4: $sk \leftarrow \text{Pack_}w(s)$
5: $b \leftarrow \text{Round}_{qp}(A^T \cdot s) \in R_p^{l\times 1}$
6: $pk \leftarrow (seed_A, \text{Pack_}\epsilon_p(b))$

Algorithm 17 Saber KEM Keypair

Input: $seed_A$, $seed_s$ and z.
Output: $pk = (seed_A, packed_b), sk = (z, pkh, pk, packed_s)$

1: $(seed_A, packed_b, packed_s) \leftarrow$ Saber.PKE.Keypair($seed_A$, $seed_s$)
2: $pk \leftarrow (seed_A, packed_b)$
3: $pkh \leftarrow \text{SHA3-256}(pk)$
4: $sk \leftarrow (z, pkh, pk, packed_s)$

Algorithm 18 Saber PKE Encryption

Input: $pk = (seed_A, packed_b)$, m and $seed_{s'}$
Output: $c = (packed_c_m, packed_b'))$

1: $s' \leftarrow \text{CBD}_\mu(\text{SHAKE128}(seed_{s'})) \in R_q^{l\times 1}$
2: $A \leftarrow \text{Unpack_}\epsilon_q(\text{SHAKE-128}(seed_A)) \in R_q^{l\times l}$
3: $b' \leftarrow \text{Round}_{qp}((A \cdot s' + h) \bmod q) \in R_p^{l\times 1}$
4: $packed_b' \leftarrow \text{Pack_}\epsilon_p(b')$
5: $b \leftarrow \text{Unpack_}\epsilon_p(packed_b)$
6: $v' \leftarrow b^T \cdot (s' \bmod p) \in R_p$
7: $c_m \leftarrow \text{Round}_{pT}(v' + h_1 - 2^{\epsilon_p - 1} \cdot m \bmod p) \in R_T$
8: $packed_c_m \leftarrow \text{Pack_}\epsilon_T(c_m)$
9: $c \leftarrow (packed_c_m, packed_b')$

Algorithm 19 Saber PKE Decryption

Input: $sk = packed_s$ and $c = (packed_c_m, packed_b'))$
Output: m

1: $s \leftarrow \text{Unpack_}w(packed_s) \in R_q^{l\times 1}$
2: $b' \leftarrow \text{Unpack_}\epsilon_p(packed_b') \in R_p^{l\times 1}$
3: $v \leftarrow b'^T \cdot s \bmod p \in R_p$
4: $m' \leftarrow \text{Round}_{p2}(v + h_2 - 2^{\epsilon_p - \epsilon_T} \cdot c_m \bmod p) \in R_2$

Algorithm 20 Saber KEM Encapsulation

Input: $pk = (seed_A, BS_b), m$
Output: $c = (packed_c_m, packed_b'))$ and a shared key K

1: $(\hat{K}, r) \leftarrow \text{SHA3-512}(\text{SHA3-256}(pk)||\text{SHA3-256}(m))$
2: $c \leftarrow$ Saber.PKE.Enc(pk, m, r)
3: $h_c \leftarrow \text{SHA3-256}(c)$
4: $K \leftarrow \text{SHA3-256}(\hat{K}||h_c)$

Algorithm 21 Saber KEM Decapsulation

Input: $sk = (z, pkh, pk = (seed_A, packed_b), packed_s)$ and $c = (packed_c_m, packed_b'))$
Output: Shared key K

1: $m' \leftarrow$ Saber.PKE.Dec$(packed_s, c)$
2: $(\hat{K}', r') \leftarrow \text{SHA3-512}(pkh||m')$
3: $c' \leftarrow$ Saber.PKE.Enc(pk, m', r')
4: $h_c \leftarrow \text{SHA3-256}(c)$
5: **if** $c = c'$ **then**
6: $\quad K \leftarrow \text{SHA3-256}(\hat{K}'||h_c)$
7: **else**
8: $\quad K \leftarrow \text{SHA3-256}(z||h_c)$
9: **end if**