

Energy-Efficient NTT Sampler for Kyber Benchmarked on FPGA

Paresh Baidya^{*‡}, Rourab Paul[†], Vikas Srivastava[§], Sumit Kumar Debnath^{*}

^{*}Department of Mathematics, National Institute of Technology, Jamshedpur, India

[†]Department of Computer Science and Engineering, Shiv Nadar University, Chennai, Tamil Nadu, India

[‡]Department of Computer Science and Engineering, Siksha ‘O’ Anusandhan Deemed to be University, Bhubaneswar, India

[§]Department of Mathematics, Indian Institute of Technology Madras, Chennai, India

[‡]pareshbaidya@soa.ac.in [†]rourabpaul@snuchennai.edu.in [§]vikas.math123@gmail.com ^{*}sdebnath.math@nitjsr.ac.in

Abstract—Kyber is a lattice-based key encapsulation mechanism selected for standardization by the NIST Post-Quantum Cryptography (PQC) project. A critical component of Kyber’s key generation process is the sampling of matrix elements from a uniform distribution over the ring \mathcal{R}_q . This step is one of the most computationally intensive tasks in the scheme, significantly impacting performance in low-power embedded systems such as Internet of Things (IoT), wearable devices, wireless sensor networks (WSNs), smart cards, TPMs (Trusted Platform Modules), etc. Existing approaches to this sampling, notably conventional SampleNTT and Parse-SPDM3, rely on rejection sampling. Both algorithms require a large number of random bytes, which needs at least three SHAKE-128 squeezing steps per polynomial. As a result, it causes significant amount of latency and energy. In this work, we propose a novel and efficient sampling algorithm, namely Modified SampleNTT, which substantially reduces the average number of bits required from SHAKE-128 to generate elements in \mathcal{R}_q —achieving approximately a 33% reduction compared to conventional SampleNTT. Modified SampleNTT achieves 99.16% success in generating a complete polynomial using only two SHAKE-128 squeezes, outperforming both state-of-the-art methods, which never succeed in two squeezes of SHAKE-128. Furthermore, our algorithm maintains the same average rejection rate as existing techniques and passes all standard statistical tests for randomness quality. FPGA implementation on Artix-7 demonstrates a 33.14% reduction in energy, 33.32% lower latency, and 0.28% fewer slices compared to SampleNTT. Our results confirm that Modified SampleNTT is an efficient and practical alternative for uniform polynomial sampling in PQC schemes such as Kyber, especially for low-power security processors.

Index Terms—Kyber, SampleNTT, FPGA, Low Power, Parse, Shake-128.

I. INTRODUCTION

Public-key cryptographic systems play a fundamental role in ensuring secure communication over the internet by enabling encryption, digital signatures, and key exchange protocols. These protocols are primarily based on the hardness of mathematical problems such as integer factorization (RSA) and the discrete logarithm problem (Elliptic Curve Cryptography, ECC). However, the advent of quantum computing poses a significant threat to the security of these cryptosystems. When executed on a sufficiently powerful quantum computer, Shor’s algorithm can efficiently solve these problems in polynomial time, rendering traditional public-key cryptography insecure. Researchers have begun to study quantum-

resistant public-key cryptographic algorithms to keep information secure from the upcoming quantum computer attack. This research area is known as Post-Quantum Cryptography (PQC). To address this emerging challenge, the U.S. National Institute of Standards and Technology (NIST) initiated the Post-Quantum Cryptography (PQC) standardization process in 2016 to develop quantum-resistant cryptographic algorithms. After four rounds of evaluation in 2022, NIST has decided to standardize CRYSTALS-Kyber as a key encapsulation mechanism (KEM) algorithm and CRYSTAL-Dilithium as a signature scheme. The theoretical foundation of the various lattice-based protocols is based on the computational hardness of Learning With Error (LWE). Its variants over Ring Learning With Error (RLWE) and [1] and Module Learning With Error (MLWE) [2] have been thoroughly investigated against both the classical and quantum adversaries. A growing number of researchers are actively working on optimized implementations of PQC schemes on Central Processing Units (CPUs), Graphical Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs), aiming to achieve optimal trade-offs in terms of resource usage, latency, and energy consumption across both software and hardware platforms. In particular, processor based software implementations offer flexibility and ease of deployment, whereas Application-Specific Integrated Circuits (ASICs) and reconfigurable hardware (e.g., FPGAs) provide high-performance and low-power solutions. FPGA-based implementations are becoming more popular compared to ASICs due to their cost-effectiveness and reconfigurable nature. In contrast, a pure hardware (FPGA) implementations can achieve significant performance by applying well-known optimization techniques such as register balancing, parallel processing, and efficient resource sharing. These designs are often reconfigurable and cost-effective, making them suitable for high-performance applications.

CRYSTALS-Kyber is a lattice-based cryptosystem whose security relies on difficulty of solving the learning-with-errors (LWE) problem in module lattices. It is a quantum-resistant key encapsulation mechanism (KEM) that achieves IND-CCA2 (indistinguishability under adaptive chosen ciphertext attack) security. The fundamental algebraic operation in Kyber is of the form $As + e$, where s and e are k -dimensional

polynomial vectors, and \mathbf{A} is a $k \times k$ polynomial matrix with coefficients in the polynomial ring $R_q = \mathbb{Z}_{3329}[X]/(X^{256}+1)$. Kyber supports three security levels - Kyber512, Kyber768, and Kyber1024—corresponding to NIST security levels 1, 3, and 5, respectively, with the module dimension k set to 2, 3, and 4. To improve the practical applicability of Kyber, researchers have extensively focused on optimizing its computational efficiency and reducing latency across various algorithmic components. In the Kyber post-quantum cryptographic scheme, one of the most computationally intensive and resource-constrained operations are sampling, Number Theoretic Transform (NTT)-based polynomial multiplication, and cryptographic hashing. Wan et al. [3] implemented a high-performance CRYSTALS-Kyber using an AI accelerator on a GPU (NVIDIA GeForce RTX 3080). They employ an NTT-box to perform the NTT/INTT operations efficiently, particularly when the polynomial dimension is relatively small. The implementation achieves a speedup of $6.47\times$ compared to the state-of-the-art on the same GPU platform. In [4], authors proposed three methods to improve the NTT performance: sliced layer merging (SLM), sliced depth first search (SDFS-NTT) and entire depth-first search (EDFS NTT). They use kernel fusion-based memory optimization technique to achieve a speedup of 7.5%, 28.5%, and 41.6% over the existing implementation. Notably, EDFs-NTT is introduced for the first time on a GPU platform (NVIDIA Titan V Volta GV100 with 5120 CUDA cores). Zou et al. [5] developed a RISC-V based processor: Seesaw, specifically designed to accelerate the Kyber schemes. This paper implemented the various components of the Kyber algorithm. Also, proposed a rejection sampling architecture where as, two coefficients are generated from the three uniform random bytes. Every three 4-bit random values pad with a 4-bit zeros to form a 16-bit word and is used to extract two matrix elements. In [6], the paper proposed a dual-issue superscalar Kyber processor (Super-K) based on RISC-V instruction set architecture (ISA). It is a parallel three stage pipeline architecture which supports conflict-free hash-based sampling and polynomial arithmetic operations. The authors [6] also design a reconfigurable polynomial arithmetic unit (PAU), which employs the fast modular reduction method and optimizes the compress/decompress process. This design reduced time overhead by 25%–33% and improved the parallelism and throughput of the overall processor. The work in [7], present a resource-efficient Kyber processor on ASIC (40nm LP CMOS) platform. The design incorporates a lightweight SHA-3 engine based on a half-fold Keccak core and reconfigurable modular arithmetic units (MAU) to compute the polynomial operations. The processor achieves a minimal power consumption of $273\mu W$ and an energy efficiency of $0.72 \mu J$ per operation in Kyber. Kim et al. [8] propose a configurable architecture for the Kyber accelerator, introducing a Memory-based Number Theoretic Transform (NTT) unit. The design also uses the Dadda tree algorithm for modular reduction, which improves processing speed, reduces hardware area, and increases data throughput. The authors in [9], present a lightweight BRAM-free FPGA implementation of the NTT/INTT unit in CRYSTALS-Kyber. They propose an optimized modular multiplier based on K-

RED [10] and lookup-table techniques. The design outperforms existing works by 36–75% in hardware efficiency and achieves a $3.4\text{--}4.4\times$ improvement in point-wise multiplication performance. An instruction set coprocessor for Kyber is presented in [11] to design a high performance hardware architecture. This architecture also implements on ASIC platform which outperforms state-of-the-art implementations. Article [12] implemented a Kyber using a non-memory-based iterative NTT for polynomial multiplication, which avoids the use of Block RAM on the Artix-7 FPGA. The authors in [13] implemented a light weight crypto processor for Kyber. Among all the aforementioned Kyber implementations, only [12] and [13] report the implementation cost of conventional SampleNTT.

A key computational process in Kyber is the generation of a structured public polynomial matrix in the Number Theoretic Transform (NTT) domain. These polynomials must be uniformly distributed over the ring R_q . The standard aforementioned implementations of Kyber utilize a rejection sampling technique in their SampleNTT algorithm, where random values are iteratively extracted from an input byte stream generated from SHAKE-128, and out-of-range values are discarded until all polynomial coefficients are assigned valid values in R_q . To the best of our knowledge, all the aforementioned literature related to Kyber implementation has focused only on the NTT, polynomial multiplication, modules reduction, and the memory storage required to store polynomial coefficients and twiddle factors. The authors in [14] proposed the first alternative to the rejection sampling algorithm for *Kyber*, adopting a simple partial discard method instead of the conventional rejection method used in SampleNTT. It reduces the required byte stream compared to results in [15]. However, the article [14] did not implement their sampler on a hardware, thereby making it difficult to justify if this method will work in a practical implementation. Keep this in context, we propose a new Modified SampleNTT implemented on a hardware that not only drastically reduces energy consumption and latency but also conserves all the statistical properties of the conventional kyber SampleNTT. The major contributions of our work are discussed below.

Our Contribution

- 1) In this work, we propose a new sampling technique (namely Modified SampleNTT), that improves the efficiency of element generation in the polynomial ring \mathcal{R}_q under the Kyber lattice-based cryptographic scheme. In particular, Modified SampleNTT offers hardware efficiency improvements over the conventional SampleNTT used in Kyber. It achieves a 33.14% reduction in energy consumption, 33.32% lower latency, and a 0.28% decrease in slice utilization. All implementations were performed and verified using the Vivado 22.04 tool on the Artix-7 FPGA platform.
- 2) We demonstrate that Modified SampleNTT reduces the average number of bits required to generate an element in \mathcal{R}_q when the XOF is instantiated with SHAKE-128. Across all Kyber security levels (Kyber512, Kyber768,

and Kyber1024), our method consistently consumes only ~ 2523.8 bits, compared to ~ 3785 bits in conventional SampleNTT and ~ 3470 bits in Parse-SPDM3. The proposed algorithm successfully generates a full polynomial in \mathcal{R}_q using exactly two squeezing steps of SHAKE-128 in 99.16% of the cases. In contrast, existing approaches such as conventional SampleNTT and Parse-SPDM3 fail to achieve this, consistently requiring three or more invocations of SHAKE-128.

- 3) Furthermore, Modified SampleNTT maintains the same average rejection sampling percentage as the existing designs, SampleNTT and Parse-SPDM3, across all Kyber security levels (as shown in Table III). The rejection rate remains consistently within the narrow range of 18.84% to 18.85%.
- 4) In addition, Modified SampleNTT passes all standard statistical tests for randomness, including the Frequency, Entropy, Kolmogorov–Smirnov (KS), Wald–Wolfowitz, and Serial tests (see Table IV). These results confirm that Modified SampleNTT retains the quality of uniform sampling and exhibits randomness characteristics on par with state-of-the-art algorithms like conventional SampleNTT and Parse-SPDM3.

The organization of the article is as follows: Section II presents the preliminaries and problem statement. The proposed Modified SampleNTT and its analysis are detailed in Sections III and IV, respectively. The hardware architecture and corresponding results are discussed in Sections V and VI. Finally, the conclusions are presented in Section VII.

II. PRELIMINARIES

Algorithm 1 Kyber.CPAPKE.KeyGen(): Key Generation [15]

```

1: Output: Secret key  $sk \in \mathbb{B}^{12 \cdot k \cdot n/8}$ 
2: Output: Public key  $pk \in \mathbb{B}^{12 \cdot k \cdot n/8 + 32}$ 
3:  $d \leftarrow \mathbb{B}^{32}$ 
4:  $(\rho, \sigma) := G(d)$ 
5:  $N := 0$ 
6: for  $i$  from 0 to  $k - 1$  do
7:   for  $j$  from 0 to  $k - 1$  do
8:      $\hat{A}[i][j] := \text{Parse}(\text{XOF}(\rho, j, i))$ 
9: ...

```

A. Public Key Generation in Kyber: SampleNTT

The key generation algorithm of Kyber produces a secret key $sk \in \mathcal{B}^{12 \cdot k \cdot n/8}$ and a public key $pk \in \mathcal{B}^{12 \cdot k \cdot n/8 + 32}$. To derive a public key, the algorithm first constructs a matrix $\hat{A} \in R_q^{k \times k}$ within the NTT (Number Theoretic Transform) domain. This matrix is generated by invoking an algorithm named SampleNTT k^2 times. The SampleNTT algorithm processes a byte stream $B = b_0, b_1, b_2, \dots \in \mathcal{B}^*$ and produces the NTT representation $\hat{a} = \hat{a}_0 + \hat{a}_1 X + \dots + \hat{a}_{n-1} X^{n-1} \in R_q$ of $a \in R_q$. The input byte stream for Algorithm SampleNTT is generated using XOF. The function XOF is recommended to be instantiated with SHAKE-128.

Algorithm 2 SampleNTT: $\mathbb{B}^* \rightarrow R_q$ [15]

```

1: Input: Byte stream  $B = \beta_0, \beta_1, \beta_2, \dots \in \mathbb{B}^*$ 
2: Output: NTT-representation  $\hat{a} \in R_q$  of  $a \in R_q$ 
3:  $i := 0$ 
4:  $j := 0$ 
5: while  $j < n$  do
6:    $d_1 := \beta_i + 256 \cdot (\beta_{i+1} \bmod +16)$ 
7:    $d_2 := \lfloor \beta_{i+1}/16 \rfloor + 16 \cdot \beta_{i+2}$ 
8:   if  $d_1 < q$  then
9:      $\hat{a}_j := d_1$ 
10:     $j := j + 1$ 
11:   if  $d_2 < q$  and  $j < n$  then
12:      $\hat{a}_j := d_2$ 
13:      $j := j + 1$ 
14:    $i := i + 3$ 
15: return  $\hat{a}_0 + \hat{a}_1 X + \dots + \hat{a}_{n-1} X^{n-1}$ 

```

To sample an element uniformly from R_q , Algorithm SampleNTT extracts twelve-bit chunks sequentially from the input byte stream and assigns a coefficient to the element only if the extracted chunk falls within the required range. If the chunk is out of range, it discards the chunk and extracts another twelve-bit segment, following a simple rejection sampling method. This process repeats until all coefficients of the element in R_q are determined. Consequently, the number of bytes (or bits) required by Algorithm SampleNTT to generate $\hat{a} \in R_q$ is not fixed and depends on the input byte stream.

III. ModifiedSampleNTT

In this section, we present the design of Modified SampleNTT algorithm (refer Algorithm 4). The proposed algorithm is an optimized polynomial sampling method that is more efficient compared to the SampleNTT algorithm 2 used in Kyber. The SampleNTT algorithm employs a rejection sampling technique, takes a byte stream as input to generate the coefficients of the polynomial in R_q . In this paper, Modified SampleNTT reduces the number of required byte streams compared to SampleNTT. It ensures a consistent sample rejection rate and efficient use of randomness. Modified SampleNTT takes an input byte stream B , and converts it into a polynomial $\hat{a} \in R_q$. It extracts *twelve-bit chunks* at a time and maps them into polynomial coefficients. Two twelve-bit values, d_1 and d_2 , are extracted from two consecutive bytes in each iteration. In line [7-8], The first value, d_1 , is computed by combining the i^{th} byte and the next $(i + 1)^{\text{th}}$ byte using bitwise OR operation. Then, apply a 12-bit mask to take the relevant bits to make a number in the range $[0, 4095]$. Similarly, d_2 is derived but with the byte order reversed to ensure the randomness in the output. In line 9 and line 12, the calculated values d_1 and d_2 are compared against the modulus $q = 3329$ respectively; if the value is less than q , it is accepted as a valid coefficient; otherwise it is rejected.

IV. ANALYSIS

We first present the comparative analysis of the average number of bits required to generate an element in \mathcal{R}_q

Algorithm 3 Parse-SPDM3: $\mathbb{B}^* \rightarrow R_q$ [14]

```

1: Input: Byte stream  $B = \beta_0, \beta_1, \beta_2, \dots \in \mathbb{B}^*$ 
2: Output: NTT-representation  $\hat{a} \in R_q$  of  $a \in R_q$ 
3:  $i := 0$ 
4:  $j := 0$ 
5: while  $j < n$  do
6:    $d_1 := 16 \cdot \beta_i + \lfloor \beta_{i+1}/16 \rfloor$ 
7:   if  $d_1 < 3584$  then
8:      $i := i + 1$ 
9:    $d_2 := (256 \cdot \beta_i \bmod^+ 2^{12}) + \beta_{i+1}$ 
10:  if  $d_1 < q$  then
11:     $\hat{a}_j := d_1$ 
12:     $j := j + 1$ 
13:  if  $d_2 < q$  and  $j < n$  then
14:     $\hat{a}_j := d_2$ 
15:     $j := j + 1$ 
16:  if  $d_2 < 3584$  then
17:     $i := i + 2$ 
18:   $i := i + 1$ 
19: return  $\hat{a}_0 + \hat{a}_1X + \dots + \hat{a}_{n-1}X^{n-1}$ 

```

Algorithm 4 Modified SampleNTT: $\mathbb{B}^* \rightarrow R_q$

```

1: Input: Byte stream  $B = \beta_0, \beta_1, \beta_2, \dots \in \mathbb{B}^*$ 
2: Output: NTT-representation  $\hat{a} \in R_q$  of  $a \in R_q$ 
3:  $i := 0$ 
4:  $j := 0$ 
5:  $\text{mask} := 4095$ 
6: while  $j < n$  do
7:    $d_1 := ((\beta_i \mid (256 \cdot \beta_{i+1})) \& \text{mask})$ 
8:    $d_2 := ((\beta_{i+1} \mid 256 \cdot \beta_i) \& \text{mask})$ 
9:   if  $d_1 < q$  then
10:     $\hat{a}_j := d_1$ 
11:     $j := j + 1$ 
12:  if  $d_2 < q$  and  $j < n$  then
13:     $\hat{a}_j := d_2$ 
14:     $j := j + 1$ 
15:   $i := i + 2$ 
16: return  $\hat{a}_0 + \hat{a}_1X + \dots + \hat{a}_{n-1}X^{n-1}$ 

```

when XOF is instantiated with SHAKE-128 across three Kyber security levels: Kyber512, Kyber768, and Kyber1024 (refer Table I). This metric quantifies the amount of randomness extracted from the XOF, which in turn determines the entropy and computational effort involved in generating uniform elements in the ring. We compare the conventional SampleNTT, and Parse-SPDM3 with Modified SampleNTT. For each security level, Modified SampleNTT consistently achieves a significant reduction in the number of bits required—approximately 2523.8 bits—compared to conventional SampleNTT, which requires over 3785 bits, and Parse-SPDM3, which uses around 3470 bits. The experiments were carried out over 1 million iterations. These findings confirm that Modified SampleNTT provides a more resource-efficient approach to element generation in \mathcal{R}_q . Reducing the average number of bits needed per element in \mathcal{R}_q leads to

less expansion of XOF expansion, lower energy consumption and faster execution. Thus, Modified SampleNTT method is suitable for practical deployment in resource-constrained cryptographic environments.

Table II presents an analysis of the success percentage for generating an entire polynomial in \mathcal{R}_q using exactly two squeezing steps of the SHAKE-128. Modified SampleNTT achieves a 99.16% success rate in generating a full polynomial using only two SHAKE-128 invocations, in contrast to the 0% success rate of both conventional SampleNTT and Parse-SPDM3. In addition, both SampleNTT and Parse-SPDM3 require at least three or more invocations of SHAKE-128 to produce enough randomness for polynomial generation. Thus, they incur additional computational cost, latency, and power consumption. In contrast, Modified SampleNTT optimally utilized the XOF output, and reduce the number of hash function calls required to sample a polynomial in \mathcal{R}_q .

Table III presents the *average rejection percentage* while generating an element in \mathcal{R}_q when the XOF is instantiated with SHAKE-128. Rejection sampling plays a key role in ensuring uniformity of the generated elements. Modified SampleNTT maintains rejection percentages that are statistically equivalent to those of conventional SampleNTT and Parse-SPDM3 across all Kyber security levels. We note that our improvements in bit consumption (Table I) and reducing the number of hash calls of SHAKE-128 (Table II) do not compromise sampling correctness and efficiency.

A. Statistical Analysis

In order to evaluate the statistical quality and randomness of the sampled output sequences generated by Modified SampleNTT, a comprehensive benchmarking of well-established randomness tests has been conducted. These tests are designed to uncover different types of non-random patterns or statistical anomalies. We also conducted these randomness test on existing state-of-the-art sampling algorithm such as conventional SampleNTT and Parse-SPDM3 for the completeness. Each subsection introduces the theoretical motivation and mathematical formulation of the respective test, followed by a detailed analysis of the results observed across the three sampling algorithms.

1) *Frequency Test:* The Frequency Test is used to analyze the uniformity of a sequence of discrete random variables. The objective of this test is to determine whether all possible symbols in the output sequence of a sampling algorithm appear with approximately equal frequency, as expected in a truly uniform random process. Given a sequence $S = \{s_1, s_2, \dots, s_n\}$ of n values sampled from a discrete set $\{0, 1, \dots, k-1\}$, let O_i denote the observed frequency of symbol i and let the expected frequency under the uniform distribution be $E = n/k$ for all $i = 0, 1, \dots, k-1$. In the following, the observed frequencies are compared with the expected ones using the chi-square (χ^2) statistic, defined as

$$\chi^2 = \sum_{i=0}^{k-1} \frac{(O_i - E)^2}{E}.$$

Under the null hypothesis H_0 that the data is uniformly distributed, the χ^2 statistic follows a chi-square distribution

TABLE I: Average numbers of bits required to generate an element in \mathcal{R}_q when XOF is instantiated with SHAKE-128

Cipher	SampleNTT	Parse-SPDM3	Modified SampleNTT
Kyber512	3785.8446	3470.3500	2523.8184
Kyber768	3785.7993	3470.3177	2523.8201
Kyber1024	3785.8269	3470.3223	2523.8084

TABLE II: Success percentage that an element in \mathcal{R}_q is generated using exactly two SHAKE-128 squeezing steps

SampleNTT	Parse-SPDM3	Modified SampleNTT
0%	0%	99.16%

TABLE III: Average rejection percentage while generating an element in \mathcal{R}_q when XOF is instantiated with SHAKE-128

Cipher	SampleNTT	Parse-SPDM3	Modified SampleNTT
Kyber512	18.84%	18.85%	18.84%
Kyber768	18.85%	18.84%	18.85%
Kyber1024	18.81%	18.82%	18.85%

with $(k - 1)$ degrees of freedom. To determine whether the observed sequence deviates significantly from uniformity, we computed the p -value associated with the observed χ^2 statistic. If this p -value is smaller than a pre-determined significance level $\alpha = 0.05$, the null hypothesis is rejected, indicating that the sequence does not exhibit uniform randomness. It is also important to consider the standard deviation of the frequencies, which gives an auxiliary metric of spread around the mean frequency E . The standard deviation σ of the frequencies is defined as:

$$\sigma = \sqrt{\frac{1}{k} \sum_{i=0}^{k-1} (O_i - E)^2},$$

A low standard deviation means values occur with nearly equal frequencies, indicating uniformity, while a high value suggests potential non-randomness. All three algorithms were tested using 25,600,000 samples drawn over a value space of size $k = 3329$. For an ideal uniform distribution, the expected frequency per symbol is approximately 7689.997, with a theoretical standard deviation around 87.66. The Modified SampleNTT method yielded a mean of 7690.00 and a standard deviation of 87.66, producing a chi-square statistic of 3326.6658 with a p -value of 0.503265, thereby passing the test at the 0.05 significance level. Similarly, the standard SampleNTT method resulted in a chi-square statistic of 3342.4143 and a p -value of 0.426774, with a standard deviation of 87.87, also passing the test. The Parse-SPDM3 method exhibited slightly higher variability, with a standard deviation of 88.85, a chi-square value of 3417.3402, and a p -value of 0.137072, but still comfortably passed the threshold for statistical uniformity. These results confirm that all three algorithms exhibit near-uniform output distributions and demonstrate no statistically significant deviation from ideal randomness as assessed by the frequency test and the chi-square goodness-of-fit method.

2) *Serial Test*: The Serial Test (also known as the two-dimensional frequency test) is a statistical tool that is used for the evaluation of the uniformity and independence of adjacent elements in a sequence. Let a sequence $S = \{x_1, x_2, \dots, x_n\}$ consist of n samples where each x_i takes values from a finite set $\mathcal{A} = \{0, 1, \dots, k - 1\}$. In the following, we construct ordered pairs $(x_1, x_2), (x_2, x_3), \dots, (x_{n-1}, x_n)$ resulting in $n - 1$ adjacent pairs. Under the hypothesis that S is generated by a truly uniform and independent process, each possible pair $(a, b) \in \mathcal{A} \times \mathcal{A}$ should appear with approximately equal probability, i.e., $\frac{1}{k^2}$. Since, there are $n - 1$ total pairs, the expected count for each such pair is given by

$$E = \frac{n - 1}{k^2},$$

Let $O_{i,j}$ denote the observed frequency of the pair (i, j) in the sequence. We employ the chi-square statistic to measure the deviation of observed pair frequencies from their expected values

$$\chi^2 = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} \frac{(O_{i,j} - E)^2}{E}. \quad (1)$$

Equation 1 approximately follows a chi-square distribution with $k^2 - 1$ degrees of freedom under the null hypothesis that the sequence is uniformly distributed and consecutive elements are independent. The corresponding p -value is calculated using the CDF of the chi-square distribution. A small p -value (typically $p < 0.01$ or $p < 0.05$) indicates that the observed distribution of pairs is significantly different from the expected uniform distribution. For all three sampling algorithms Modified SampleNTT, SampleNTT, and Parse-SPDM3—the chi-square statistic was approximately 11081986, with 11082240 degrees of freedom. This yielded a consistent p -value of ≈ 0.521457 in each case. Since the

p -value exceeds the common significance level of 0.05, the results indicate that all three algorithms produce sequences that exhibit randomness in terms of serial correlation.

3) *Runs Test*: The Runs Test evaluate the randomness of a binary sequence by analyzing the occurrence and distribution of uninterrupted subsequences (called “runs”) of similar elements. In other words, the test checks whether the number and lengths of such runs are consistent with what would be expected in a truly random sequence, where each bit is independently and uniformly distributed. Let us consider a binary sequence $S = \{x_1, x_2, \dots, x_n\}$ of length n , where each $x_i \in \{0, 1\}$. A *run* is defined as a maximal contiguous subsequence of identical bits. For example, in the sequence $S = 00110011$, there are four runs: two of 0s and two of 1s, with various lengths. We assume that the bits are independent and identically distributed with equal probability of 0 or 1. Let n_0 and n_1 denote the total number of 0s and 1s, respectively, in the sequence. The total number of runs, denoted R , can be determined by iterating through the sequence and incrementing the run count each time a change in bit value is observed. Let μ_R , and σ_R^2 denotes respectively, the expected number of runs under the hypothesis of randomness, and the corresponding variance. Using these parameters, the test statistic is computed as a standard normal variable:

$$Z = \frac{R - \mu_R}{\sigma_R}.$$

Under the null hypothesis that the sequence is random, the value of Z follows a standard normal distribution. A p -value is then computed from Z , and the null hypothesis is rejected if this p -value falls below a pre-determined threshold, such as 0.05 or 0.01. By mapping integer sequences to binary (e.g., thresholding), one can apply the test in our case.

For the Modified SampleNTT implementation, the observed number of runs was 499470 against an expected value of 500000.9997, yielding a Z -score of -1.0620 and a p -value of 0.4882. The conventional SampleNTT implementation showed 499777 runs against expected 500000.8330, with a Z -score of -0.4477 and a p -value of 0.3544. Lastly, the Parse-SPDM3 variant produced 500621 runs against the expected 500000.9804, resulting in a Z -score of 1.2400 and a p -value of 0.2150. In all three cases, the p -values were well above the conventional significance threshold of 0.05, leading to the conclusion that the null hypothesis of randomness could not be rejected. Thus, the output sequences from all three protocols are consistent with what would be expected from a random source in terms of run behavior.

4) *KS Test*: The Kolmogorov–Smirnov (KS) Test is a non-parametric statistical method used to assess the goodness-of-fit between an empirical distribution function (EDF) derived from a given data sample and a reference CDF. In other words, KS test can be used to test whether a sample of data conforms to a specified probability distribution. The EDF of the observed data is compared to the CDF of the theoretical distribution. The test statistic, denoted D_n , is defined as the supremum of the absolute differences between these two functions over the entire range of the data: $D_n = \sup_x |F_n(x) - F(x)|$, where $F_n(x)$ is the EDF and

$F(x)$ is the CDF of the reference distribution. In the case of a two-sample K-S test, the test compares two empirical distributions $F_n(x)$ and $G_m(x)$ from independent samples, using the test statistic $D_{n,m} = \sup_x |F_n(x) - G_m(x)|$. To determine statistical significance, the computed test statistic is used to compute a p -value. A smaller KS statistic value suggests a closer match to the reference distribution. We took the reference distribution to be uniform. For all three sampling algorithms Modified SampleNTT, SampleNTT, and Parse-SPDM3—the KS statistics were 0.00208, 0.00082, and 0.00106 respectively, with corresponding p -values of 0.7777, 0.5073, and 0.2041. Since all p -values are greater than the common significance threshold of 0.05, the null hypothesis that the sample follows the reference distribution cannot be rejected in any case. These results confirm that the outputs of all three protocols conform closely to the uniform distribution and can be considered statistically random under the KS test.

5) *Entropy Test*: The Entropy Test quantifies the level of uncertainty or unpredictability in a sequence of discrete random variables. The motivation behind this test is based on the idea that a truly random sequence should exhibit maximum entropy. Mathematically, for a discrete random variable X with a finite set of outcomes $\{x_1, x_2, \dots, x_k\}$ and corresponding empirical probabilities $p_i = \Pr(X = x_i)$ for $i = 1, 2, \dots, k$, the Shannon entropy $H(X)$ is defined as:

$$H(X) = - \sum_{i=1}^k p_i \log_2 p_i. \quad (2)$$

This measure reaches its maximum value when all outcomes are equally likely, i.e., when $p_i = 1/k$ for all i , resulting in $H_{\max} = \log_2 k$. We consider a sequence $S = \{s_1, s_2, \dots, s_n\}$ consisting of n symbols drawn from an alphabet of size k . The test involves counting the occurrences of each symbol x_i in S to estimate their empirical probabilities $p_i = \frac{f_i}{n}$, where f_i is the frequency count of x_i . The entropy of the sequence is then computed using the formula above. This value is then compared to the theoretical maximum entropy $\log_2 k$ expected from a perfectly uniform random distribution. A small deviation from the maximum indicates good randomness, while a significant drop suggests potential bias or pattern structure in the data. We utilized the entropy test to assess the unpredictability in the sequence outputted by SampleNTT, Modified SampleNTT, and Parse-SPDM3. The entropy H of a discrete probability distribution over a set of size $k = 3329$ is maximized when all possible outcomes are equally likely, in which case the expected entropy value is $H_{\text{expected}} = \log_2(3329) \approx 11.7009$. For each of the three sampling schemes, the observed entropy was computed using Equation 2. All three methods yielded nearly identical empirical entropy values of $H = 11.7008$, which are extremely close to the theoretical maximum.

V. HARDWARE ARCHITECTURE

This section discusses the hardware architectures of conventional SampleNTT and Modified SampleNTT.

TABLE IV: Statistical Test Results for Randomness

	SampleNTT	Parse-SPDM3	Modified SampleNTT
Frequency Test	✓	✓	✓
Entropy Test	✓	✓	✓
Kolmogorov-Smirnov (KS) Test	✓	✓	✓
Wald-Wolfowitz Test	✓	✓	✓
Serial Test	✓	✓	✓

A. Conventional SampleNTT

As shown in Fig. 1, the conventional SampleNTT used in Kyber has 9 sub components.

1) *Seed Memory (SeedMem)*: Seed Memory (*SeedMem*) is a First In First Out (FIFO) memory which stores 504 bytes generated by SHAKE-128 algorithm used in Kyber variants. This block has 7 ports : *clk_rd*, *clk_wr*, *rd_en*, *wr_en*, *rst*, *din* and *dout*. The *SeedMem_ctrl* generates the control signals : *rd_en*, *wr_en*, *rst* of *SeedMem* to read the output $B = [\beta_0, \beta_1, \beta_2, \dots]$ from *SeedMem*. The output of *SeedMem* is buffered in β_i Block, β_{i+1} Block and β_{i+2} Block blocks.

2) *Seed Memory Controller SeedMem_ctrl*: This *SeedMem_ctrl* module generates control signals : *rd_en*, *wr_en*, *rst* for the *SeedMem*. Once the conventional SampleNTT is enabled, *SeedMem_ctrl* reads bytes from *SeedMem* on each rising clock edge and sends them to the β_i , β_{i+1} , and β_{i+2} blocks.

3) *Controller (CTRL)*: The Controller (*CTRL*) block generates enable (*en*) and reset (*rst*) signals for *SeedMem_ctrl*, β_i , *D1_Gen*, *D2_Gen*, *Rejecter* block, β_{i+1} block and β_{i+2} blocks.

4) β_i Block: The β_i block latches the 0_{th} , 3_{rd} , 6_{th} , ... bytes from *SeedMem* when β_{i_en} from *CTRL* is high.

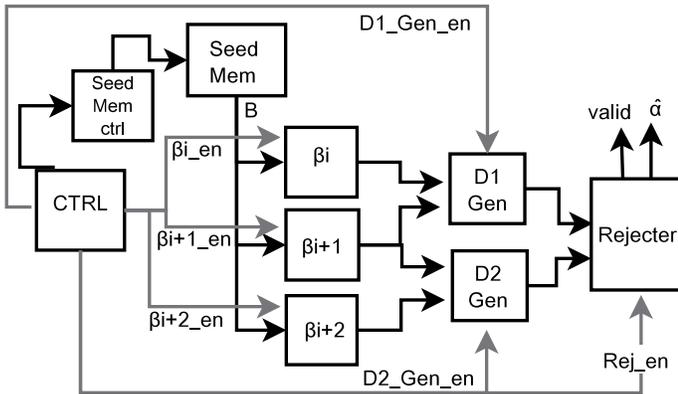


Fig. 1: Hardware Architecture of Conventional SampleNTT used in Kyber

5) β_{i+1} Block: The β_{i+1} block latches the 1_{st} , 4_{th} , 7_{th} , ... bytes from *SeedMem* when β_{i+1_en} from *CTRL* is high.

6) β_{i+2} Block: The β_{i+2} block latches the 2_{nd} , 5_{th} , 8_{th} , ... bytes from *SeedMem* when β_{i+2_en} from *CTRL* is high.

7) *D1 Generator (D1_Gen)*: When *CTRL* sets *D1_Gen_en* high, the D1 Generator (*D1_Gen*) reads

β_i from the β_i Block and β_{i+1} from the β_{i+1} Block, then executes line 6 of Algorithm 2.

8) *D2 Generator (D2_Gen)*: When *CTRL* sets *D2_Gen_en* high, the D2 Generator (*D2_Gen*) reads β_{i+1} from the β_{i+1} Block and β_{i+2} from the β_{i+2} Block, then executes line 7 of Algorithm 2.

9) *Rejecter Block (Rej_Block)*: When *CTRL* sets *Rej_en* high, the Rejecter Block (*Rej_Block*) checks whether *D1* and *D2* are less than q or not (line 8 and line 11 2). If this condition holds true, *D1* and *D2* are accepted; otherwise, they are rejected.

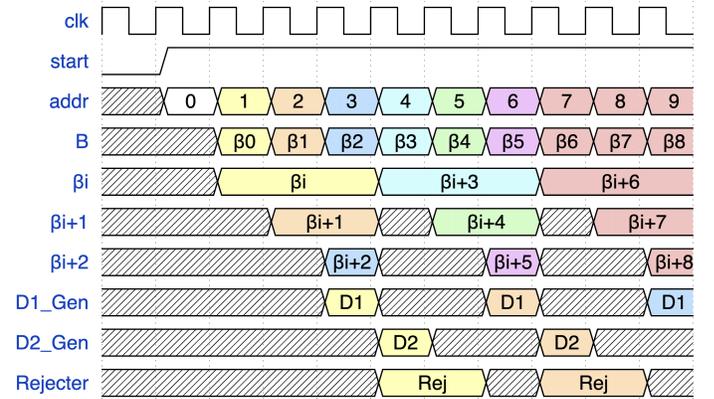


Fig. 2: Timing Diagram of Conventional SampleNTT used in Kyber

Fig. 2 shows the timing diagram of conventional SampleNTT. The $\beta_0, \beta_1, \beta_2, \dots$ are read in every clock cycle from *SeedMem*. The values $\beta_0, \beta_3, \beta_6, \dots$ are latched by the β_i Block every 3 clock cycles. Similarly, $\beta_1, \beta_4, \beta_7, \dots$ are latched every 3 clock cycles by the β_{i+1} -Block, and $\beta_2, \beta_5, \beta_8, \dots$ are latched every 3 clock cycles by the β_{i+2} -Block. The β_i, β_{i+1} and β_{i+2} are latched for 3 clock cycles, 2 clock cycles and 1 clock cycles respectively. Once β_i and β_{i+1} are latched by the β_i Block and β_{i+1} Block, respectively, the *D1_Gen* is ready to compute *D1*. Similarly, once β_{i+1} and β_{i+2} are latched by the β_{i+1} Block and β_{i+2} Block, respectively, the *D2_Gen* is ready to compute *D2*. The *Rejecter* Block starts once *D1* is computed and compares *D1* with q . After *D1* is ready, *D2* is computed in the next clock cycle. The *Rejecter* Block then continues by comparing *D2* with q . As a result, the *Rejecter* Block is activated in every 3-clock-cycle period. For the first 2 clock cycles, the *Rejecter* Block remains active, and for the remaining 1 clock cycle, it is inactive.

B. Proposed Modified SampleNTT

The proposed modified SampleNTT stated in algorithm 4 does not require β_{i+2} Block. As shown in Fig. 3, this Modified SampleNTT has three primary changes.

1) β_{i+2} Block: The modified SampleNTT does not require β_{i+2} Block. The $D1$ and $D2$ can be computed directly from β_i and β_{i+1} .

2) CTRL: As β_{i+2} Block is absent, the control signals of β_{i+2} Block from CTRL are not required. Therefore, it reduces the implementation cost of CTRL.

3) $D1$ & $D2$ Generator ($D1_Gen$ & $D2_Gen$): As shown in lines 7 and 8 of Algorithm 4, apart from the OR operation, both lines involve a multiplication by 256 and an AND operation with the $mask$ value 4095. To make the hardware more efficient, we replace these two operations with two lightweight operations.

- The multiplication by 256 is replaced by the left-shifting β_{i+1} (line 7) and β_i (line 8) by 8 bits.
- The mask operation with 4095 is replaced by truncating all bits beyond the 12th bit (since $2^{12} = 4096$).
- 4) *Seed Memory*: The modified SampleNTT requires only ~ 336 bytes to store in *SeedMem* for Kyber standard, while the conventional SampleNTT requires ~ 504 bytes for storage in *SeedMem*.

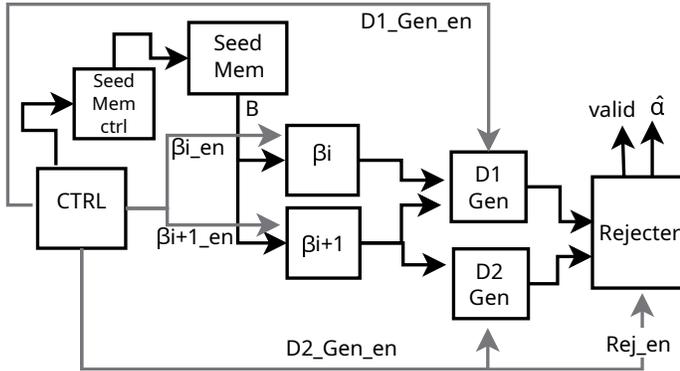


Fig. 3: Hardware Architecture of Modified Sample NTT

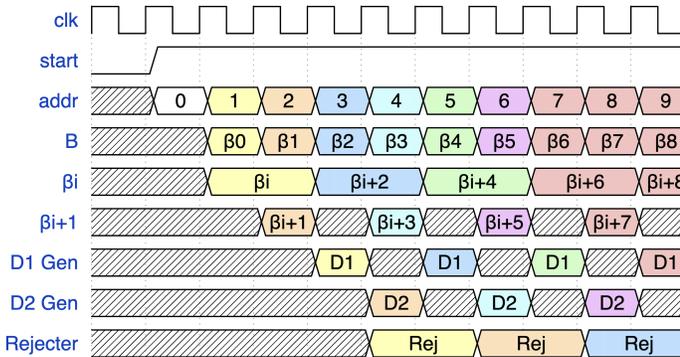


Fig. 4: Timing Diagram of Modified SampleNTTs

VI. HARDWARE RESULTS & DISCUSSIONS

The conventional SampleNTT used in Kyber, Parse-SPDM3 and proposed Modified SampleNTT are imple-

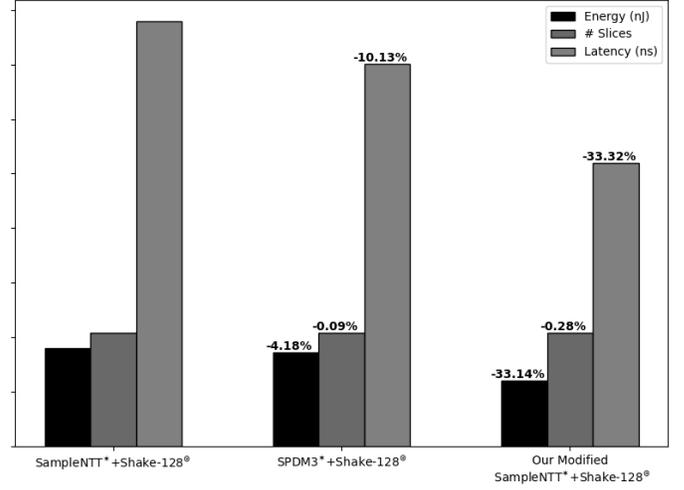


Fig. 5: Reduction of Implementation Cost of Our Modified SampleNTT* + SHAKE – 128[®] and SPDM3* + SHAKE – 128[®] as Compared to SampleNTT* + SHAKE – 128[®]

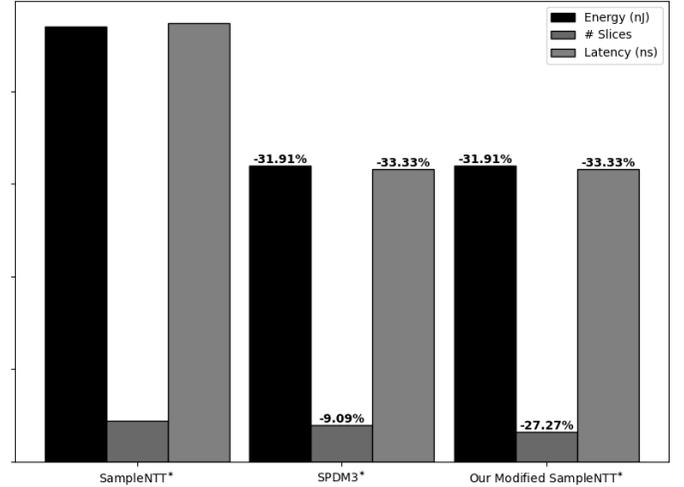


Fig. 6: Reduction of Implementation Cost of Our Modified SampleNTT* and SPDM3* as Compared to SampleNTT*

mented on *Artix-7 (xc7a100tcsq324-3)* FPGA with the *Vivado 22.02* tool and the VHDL language. This section discusses the impacts of the adopted changes on the implementation cost of our Modified SampleNTT. The detailed implementation costs of our conventional SampleNTT*, the SampleNTT from [12], the SampleNTT from [13], Parse-SPDM3* from [14], and the proposed Modified SampleNTT* are presented in Table V. It is to be noted that the * symbols indicate the hardware designs are implemented by us. To the best of our knowledge, articles [12] and [13] are the only works in the literature that reported the implementation cost of the SampleNTT.

1) *Impact on Energy and Time*: Our proposed modified SampleNTT* can generate the required number of $d1$ s and $d2$ s using 336 input bytes generated from the SHAKE – 128[®],

Design Names	# Slices	# LUTs	# FFs	# DSPs	Energy (nJ)	Clock Period (ns)	# Clock Cycles
Sample NTT [12]	62	116	141	0	NA*	NA*	NA*
Sample NTT [13]	78	246	133	0	NA*	3.7	4623
Sample NTT *	44	113	96	0	~470	10	~474
- SeedMem_ctrl	9	15	20	0			
- β_i Block	1	0	8	0			
- β_{i+1} Block	3	0	8	0			
- β_{i+2} Block	1	0	8	0			
- d1_gen	4	6	12	0			
- d2_gen	7	13	12	0			
- rejecter	9	3	15	0			
- seed_mem	22	72	10	0			
- CTRL	6	4	6	0			
SHAKE-128 [⊗]	4115	37577	47898	0	~3114	3.264	3324
SPDM3 *	40	106	79	0	~320	10	~316
- SeedMem_ctrl	6	9	19	0			
- β_i Block	2	0	8	0			
- β_{i+1} Block	3	0	8	0			
- d1_gen	9	14	12	0			
- d2_gen	4	4	12	0			
- rejecter	10	4	15	0			
- seed_mem	22	72	10	0			
- CTRL	3	3	6	0			
SHAKE-128 [⊗]	4115	37577	47898	0	~3114	3.264	3324
Our Modified Sample NTT *	32	106	79	0	~320	10	~316
- SeedMem_ctrl	5	9	20	0			
- β_i Block	2	0	8	0			
- β_{i+1} Block	3	0	8	0			
- seed_mem	23	72	10	0			
- d1_gen	7	14	15	0			
- d2_gen	4	4	15	0			
- rejecter	9	4	15	0			
- CTRL	2	3	3	0			
SHAKE-128 [⊗]	4115	37577	47898	0	~2076	3.264	2216

The proposed Modified SampleNTT* is successfully tested with Kyber-512 on the Artix-7 FPGA (xc7a100tcs324-3).

Note: NA* = Data Not Available; * = Designed by us; [⊗] = AMD-Xilinx Vitis Security Library [16].

TABLE V: Implementation Costs of Sample NTTs

whereas the conventional SampleNTT* and Parse-SPDM3* require 504 bytes from the SHAKE – 128[⊗]. These extra bytes generated in SHAKE – 128[⊗] for both the conventional SampleNTT* and the Parse-SPDM3* require extra clock cycles, which causes a significant amount of latency and energy. As a result, as shown in Fig. 5, our Modified SampleNTT* alone efficiently reduces energy consumption by 31.91% and latency by 33.33%, compared to the conventional SampleNTT* used in Kyber. Our Modified SampleNTT*+SHAKE – 128[⊗] efficiently reduces energy consumption by 33.14% and latency by 33.32%, compared to the SampleNTT*+SHAKE – 128[⊗] used in Kyber. On the other hand, Parse-SPDM3* is able to reduce energy consumption by 4.18% and latency by 10.13%, compared to the SampleNTT* used in Kyber. It is to be noted that the main SampleNTT* and SHAKE – 128[⊗] can run in parallel in all the 3 designs. However, the latencies of Parse-SPDM3*, the conventional SampleNTT* and our Modified

SampleNTT* are calculated as the sum of the latencies of SampleNTT*/Parse-SPDM3* and SHAKE – 128[⊗]. The detailed timing diagram of conventional SampleNTT used in Kyber and our Modified SampleNTTs are shown in Fig. 2 and Fig. 4 respectively. As the SampleNTT used in Kyber requires 3 bytes from *SeedMem* to generate one set of d_1, d_2 , there is an empty clock cycle after generating each set. However, our modified SampleNTT requires only 2 bytes from *SeedMem* to generate one set of d_1, d_2 , and thus does not incur any empty clock cycles. As a result, considering the rejection rate to generate 256 sets of d_1, d_2 , our SampleNTT requires ~ 316 clock cycles, whereas the conventional SampleNTT requires ~ 474 clock cycles. In our modified design, using fewer bytes from *Shake*-128 and eliminating the empty clock cycle in SampleNTT significantly reduces both energy consumption and latency.

2) *Impact on Area*: Our modified SampleNTT* eliminates the β_{i+2} blocks. Therefore, our modified SampleNTT* can

produce the required number of d_1 and d_2 using only 336 bytes generated from SHAKE – 128[®], instead of the 504 bytes required by the conventional SampleNTT*. As a result, it reduces the FIFO depth of the *SeedMem* used to store the bytes from SHAKE – 128[®]. Additionally, the *CTRL* block of our SampleNTT* becomes lightweight as it no longer needs to generate control signals for the β_{i+2} block. As a result, the slice consumption of our modified SampleNTT* is reduced by 27.27% compared to the conventional SampleNTT* used in Kyber, whereas Parse-SPDM3* achieves only a 9.09% reduction in slice overhead relative to conventional SampleNTT*.

Availability of Codes

The RTL and the statistical test code for this work are uploaded to GitHub¹.

VII. CONCLUSION

SampleNTT and *Shake* – 128 are among the most fundamental and critical components of next-generation security processors incorporating Kyber. However, SampleNTT and *Shake* – 128 cause significant energy and latency overhead, which makes them unsuitable for low-power field embedded systems. SampleNTT requires an adequate number of random bytes from *Shake* – 128 in Kyber. Generating more random bytes using *Shake* – 128 leads to increased latency and energy consumption. To make Kyber suitable for low-power, resource-constrained devices, the proposed Modified SampleNTT adopts two measures without affecting its statistical properties: (i) It reduces the required number of random bytes from *Shake* – 128 and (ii) The proposed algorithm for Modified SampleNTT avoids extra buffering of random byte and develops light weight Controller. As a result, our Modified SampleNTT+*Shake* – 128 reduces energy consumption by 33.14%, latency by 33.32% and slice utilization by 0.28%, compared to the SampleNTT+*Shake* – 128 used in Kyber. Meanwhile, our Modified SampleNTT alone reduces energy consumption by 31.91%, latency by 33.32% and slice utilization by 27.27% compared to the SampleNTT used in Kyber. As part of future work, we aim to further reduce the rejection sampling percentage while preserving uniformity and correctness. Additionally, an in-depth study of the rejection sampling behaviour in other lattice-based schemes such as Dilithium will be performed to explore the broader applicability of our optimization techniques.

REFERENCES

- [1] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, pages 1–23. Springer, 2010.
- [2] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3):565–599, 2015.

- [3] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuwu Wang, Jingqiang Lin, and Jiankuo Dong. A novel high-performance implementation of CRYSTALS-kyber with AI accelerator. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 514–534. Springer Nature Switzerland, Cham, 2022.
- [4] Xinyi Ji, Jiankuo Dong, Tonggui Deng, Pinchang Zhang, Jiafeng Hua, and Fu Xiao. Hi-kyber: A novel high-performance implementation scheme of kyber based on gpu. *IEEE Transactions on Parallel and Distributed Systems*, 2024.
- [5] Xiaofeng Zou, Yuanxi Peng, Tuo Li, Lingjun Kong, and Lu Zhang. Seesaw: A 4096-bit vector processor for accelerating kyber based on risc-v isa extensions. *Parallel Computing*, 123:103121, 2025.
- [6] Jiaming Zhang, Jiahao Lu, Aobo Li, Mingbo Wang, Xiang Li, Tianze Huang, Lei Chen, and Dongsheng Liu. Super k: A superscalar crystals kyber processor based on efficient arithmetic array. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2024.
- [7] Aobo Li, Jiahao Lu, Dongsheng Liu, Shuo Yang, Tianze Huang, Jiaming Zhang, Siqi Xiong, Chenjun Yang, and Xiang Li. A 273 μ w 0.34 mm² efficient crystals-kyber processor for pqc towards edge computing. In *2024 IEEE European Solid-State Electronics Research Conference (ESSERC)*, pages 472–475. IEEE, 2024.
- [8] Hyunseon Kim, Haesung Jung, Ardianto Satriawan, and Hanho Lee. A configurable ml-kem/kyber key-encapsulation hardware accelerator architecture. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2024.
- [9] Ziying Ni, Ayesha Khalid, Weiqiang Liu, and Maire O’Neill. Towards a lightweight crystals-kyber in fpgas: an ultra-lightweight bram-free ntt core. In *IEEE International Symposium on Circuits and Systems 2023*. IEEE, 2023.
- [10] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016.
- [11] Mojtaba Bisheh-Niasar, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Instruction-set accelerated implementation of crystals-kyber. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(11):4648–4659, 2021.
- [12] Trong-Hung Nguyen, Duc-Thuan Dam, Phuc-Phan Duong, Binh Kieu-Do-Nguyen, Cong-Kha Pham, and Trong-Thuc Hoang. Efficient hardware implementation of the lightweight crystals-kyber. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 72(2):610–622, 2025.
- [13] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. A configurable crystals-kyber hardware implementation with side-channel protection. *ACM Trans. Embed. Comput. Syst.*, 23(2), March 2024.
- [14] Dongyoung Roh and Sangim Jung. Applying the simple partial discard method to crystals-kyber. *IEEE Access*, 12:3476–3487, 2024.

¹<https://github.com/rourabpaul1986/SampleNTT>

- [15] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation. *NIST PQC Round*, 2(4):1–43, 2019.
- [16] AMD Xilinx. Vitis security library 2019.2 for sha-3 algorithms. 2019.