Efficient Accelerator for NTT-based Polynomial Multiplication

Raziyeh Salarifard and Hadi Soleimany

***Abstract*—The Number Theoretic Transform (NTT) is used to efficiently execute polynomial multiplication. It has become an important part of lattice-based post-quantum methods and the subsequent generation of standard cryptographic systems. However, implementing post-quantum schemes is challenging since they rely on intricate structures. This paper demonstrates how to develop a high-speed NTT multiplier highly optimized for FPGAs with few logical resources. We describe a novel architecture for NTT that leverages unique precomputation. Our method efficiently maps these specific pre-computed values into the built-in Block RAMs (BRAMs), which greatly reduces the area and time required for implementation when compared to previous works. We have chosen Kyber parameters to implement the proposed architectures. Compared to the most well-known approach for implementing Kyber's polynomial multiplication using NTT, the time is reduced by 31%, and AT (area $\times$ time) is improved by 25% as a result of the pre-computation we suggest in this study. It is worth mentioning that we obtained these improvements while our method does not require DSP.**

***Keywords*—** Post quantum cryptography, latticed-based cryptography, Crystals-Kyber, Number Theoretic Transform, polynomial multiplication.

I. Introduction

Various applications have utilized modern public-key cryptosystems to guarantee secrecy, data integrity, and authentication. Conventional public-key cryptography techniques derive their security from the hardness of NP-complete problems such as factorization or discrete logarithms of large integers. Still, if large-scale quantum computers are built, Shor's algorithm could be used to solve these mathematical problems. The National Institute of Standards and Technology (NIST) started a standardization process in 2016 to identify post-quantum public-key cryptography algorithms. This was done because research and development of quantum computers had made significant progress. The NIST announced the winners on July 5, 2022. Lattice-based cryptography (LBC) is a viable alternative because it provides robust security and a trade-off between efficiency and security that is suitable for a wide range of applications. The only key encapsulation mechanism (KEM) accepted for standardization by NIST is Kyber. Kyber's security is built on module learning with errors (M-LWE). Kyber is thought to be quantum-resistant while maintaining the efficiency advantages of lattice-based approaches. Kyber is a part of "Cryptographic Suite for Algebraic Lattices" (CRYSTALS), which shares a framework with the Dilithium signature scheme. NTT-based multiplication is a fast way to perform n element vector multiplication, reducing the number of required calculations from $O(n^2)$ to $O(n)$. Since NTT-based multiplication is an expensive operation, a great deal of focus has recently been placed on its implementation. While substantial research has been conducted on the software implementation of NTT for multiple architectures, the hardware implementation has gotten less attention.

R. Salarifard is with the Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran. Email: r_salarifard@sbu.ac.ir

H. Soleimany is with Cyber Research Center, Shahid Beheshti University, Tehran, Iran, e-mail: h_soleimany@sbu.ac.ir

A. Related Work

[1] proposed the first hardware implementation of the ideal lattice-based cryptographic scheme, which is a fully parallel architecture and requires a large area. [2] proposed using negative-wrapped convolution for $Z_q[x]/< x^n + 1 >$ arithmetic to reduce the number of evaluations in NTT. The authors propose a sequential polynomial multiplier, the ROM component of which contains all of the twiddle factors required for the NTT computation. However, using negative wrapped convolution incurs pre- and post-computing costs. In a subsequent paper [3], the authors included the polynomial multiplier [1] into a ring-LWE-based encryption system and proposed a number of system-level improvements.

The authors of [4] proposed a compact implementation in which the twiddle factors are computed on demand rather than being stored in ROM in order to reduce the space. In [5], the authors reduced the number of multiplications by rearranging the loops within the NTT computation and combining the pre-processing of NTT with butterfly operations. In addition, [6] introduces low-complexity NTT and inverse NTT (INTT) by incorporating the pre-processing of NTT and the post-processing of INTT into the Fast Fourier Transform (FFT) algorithm.

Longa et al. [7] developed two novel modular reduction techniques, KRED and KRED-2X, to accelerate the NTT computation at the expense of an increase in memory requirements. [8] proposed an energy-efficient modular arithmetic implementation by employing Cooley-Tukey (CT) and Gentleman-Sande (GS) butterfly configurations [7] in order to avoid bit-reverse operation by using CT for NTT and GS for INTT. The authors of [9] presented a processor that reduces memory access overhead by optimizing the NTT algorithm with GS butterfly. In [10, 11], a parametric NTT architecture that is adaptable and scalable was presented. In addition, [12] provides an efficient and flexible NTT architecture on RISC-V. [13] proposes a low-power-optimized NTT.

A compact implementation of NTT based on a single butterfly core guarantees a small footprint, but cannot achieve the high throughput demanded by particular applications. Four butterfly cores are utilized to propose a pipelined polynomial multiplication architecture in [14]. This architecture provides a high-speed implementation for Ring-LWE-based cryptosystems by increasing memory accesses. To address this

challenge, the authors of [15] merged NTT layers based on a 2×2 butterfly structure, which was subsequently used in follow-up works for the implementation of NewHope and Kyber [16, 17, 18]. These designs employ the KRED and KRED-2X reductions, which roughly double the hardware area and necessitate additional memory to store two sets of pre-computed values.

B. Our Contributions

Due to the limited resources available to the reconfigurable hardware, an efficient implementation of post-quantum schemes is crucial in an FPGA. The FPGA resources are not always utilized properly. For example, BRAMs are utilized less frequently compared to other sources, such as DSPs. We provide a novel implementation of NTT multiplication that makes effective use of FPGA resources. Our architecture possesses the following attributes:

- We provide an innovative implementation of NTT multiplication that employs particular precomputations. The overhead of our method for storing pre-computed values consists of utilizing Block RAMs (BRAMs), which are building blocks available in nearly every implementation using an FPGA.
- Rather than being used as stand-alone primitives, cryptographic schemes are frequently employed as building blocks for larger secure systems. While BRAMs are not commonly employed in the implementation of secure systems, DSPs are in great demand. Our architecture's lack of DSPs is a clear indication of its minimal resource consumption, which is suitable for FPGA. Compared to the most common method for implementing NTT multiplication, our architecture reduces area and time complexity.
- We introduce a novel, unified module that can be reconfigured to execute CT butterfly (useful for NTT), GS butterfly (helpful for INTT), and point-wise multiplication utilizing pre-computation. This unified module takes up less area and can be run in a single clock cycle.
- A pipeline architecture for the NTT multiplier with several stages is given (three pipeline stages). In addition, the pipeline architecture employs four butterfly cores in parallel, which divides the input by four. With three stages and divided inputs, the latency is decreased in comparison to earlier works.

The rest of this paper is organized as follows. Section II presents preliminaries about Kyber, NTT and reduction algorithm. The proposed NTT multiplication algorithm is presented in Section III. Additionally, an efficient and high-speed multiplier architecture is presented in Section IV. Section V presents the experimental results. Finally, we conclude the paper in Section VI.

II. Preliminaries

We have chosen Kyber's parameter to implement our proposed architecture; hence, in the following, Kyber is briefly described. Then the details of Number Theoretic Transform (NTT) and modular reduction (which is frequently called in NTT) are discussed.

A. Kyber

Kyber is an IND-CCA-secure technique for key encapsulation (KEM), which is designed to be secure in the presence of large quantum computers. Kyber is the only KEM selected for standardization at the conclusion of NIST's standardization competition. Kyber's security relies on the difficulty of solving the learning-with-errors (LWE) problem over module lattices. As a result, it possesses the typical advantages of lattice-based schemes, including a short runtime and a small ciphertext size. Kyber provides three different levels of security, each with its own set of settings. Kyber512, Kyber768, and Kyber1024 are Kyber variants that offer a level of security comparable to AES-128, AES-192, and AES-256, respectively. Kyber employs polynomials with 256 coefficients over k-dimensional vectors, where k is one of 2, 3, or 4 dimensions, based on the post-quantum security level.

The implementation of Kyber includes all Keccak variations, additions, and multiplications in Z_q and the NTT over the $Z_q[X]/(X^{256}+1)$ ring, where q is 3329. Like other key-encapsulation mechanisms, Kyber's KEM consists of three probabilistic algorithms: key generation (KeyGen), encapsulation (Encaps), and decapsulation (Decaps). KeyGen generates both public and private keys (pk, sk). Encaps takes the public key pk as an input and returns a ciphertext c and a key K. Decaps accepts as inputs the secret key sk and the ciphertext c and returns either K or the sign to indicate rejection. KeyGen samples a matrix $\mathbf{A}$ from a uniform distribution, a secret key $\mathbf{s}$, and a noise $\mathbf{e}$ from a binomial distribution. We refer interested readers to [19] for additional information.

B. The Number Theoretic Transform

NTT is a variation of the Discrete Fourier Transform which is defined over the ring Z_q. NTT can be viewed as a generalization of FTT, which is defined in a finite field. NTT transforms an $(n-1)$ degree polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$ to an $(n-1)$ degree polynomial $\hat{A}(x) = \sum_{i=0}^{n-1} \hat{a_i}$ where $\hat{a_i} = \sum_{j=0}^{n-1} a_j \omega_n^{ij} \bmod q$ and $\omega_n \in \mathbb{Z}_q$ denotes the n-th primitive root of unity in $\mathbb{Z}_q$. ω is a constant that is usually called the twiddle factor. Similarly, the inverse of NTT (INTT) employs a similar formula with $\omega^{-1} \bmod q$ and a multiplication of coefficients with $n^{-1} \bmod q$, i.e. $a = INTT(\hat{a})$ where $a_i = n^{-1} \sum_{j=0}^{n-1} \hat{a}_j \omega_n^{-ij} \bmod q$.

In a general case, the multiplication of two polynomials $A(x)$ and $B(x)$ in R_q requires the doubling of input's sizes by n zero padding and the computation of an explicit reduction modulo q: $INTT_{2n}(NTT_{2n}(a)NTT_{2n}(b)) \bmod q$. To avoid such padding, the negative wrapped convolution (NWC) is presented, which requires a pre-processing step for NTT and a post-processing step for INTT. The coefficients of input and output polynomials should be multiplied by $[\psi^0, \psi^1, \cdots, \psi^{(n-1)}]$ and $[\psi^0, \psi^{-1}, \cdots, \psi^{-(n-1)}]$, respectively, where ψ is $2n$-th root of unity ($\psi = \sqrt{\omega_n}$). By precomputing and storing these values in memory, substantial savings can be realized. Prior studies demonstrate how to mix multiplications by powers of ψ and powers of ψ^{-1} within the NTT calculation. Using the Cooley-Tukey butterfly, which

was leveraged in the early implementations of R-LWE-based schemes, [5] demonstrates how to reduce the overhead of memory access. Similarly, [20] explains how to combine the powers of ψ^{-1} and the power of ω_n in the INTT using the Gentleman-Sande (GS) butterfly. As we will explain later, our design incorporates both CT and GS techniques.

Algorithm 1 Computing NTT Based on Cooley-Tukey butterfly.

Require: A polynomial $a(x) \in Z_q[X]/(X^n+1)$
Ensure: $\hat{a}(x) = NTT(a(x))$
1: $t = n$
2: **for** $m = 1, m < n, m = 2m$ **do**
3: $t = t/2$
4: **for** $i = 0, i < m, i++$ **do**
5: $j_1 = 2 \cdot i \cdot t$
6: $j_2 = j_1 + t - 1$
7: $S = \psi^{br(m+i)}$
8: **for** $j = j_1, j \leq j_2; j++$ **do**
9: $u = a_j$
10: $v = a_{j+t}$
11: $a_j = u + S \cdot v \bmod q$
12: $a_{j+t} = u - S \cdot v$
13: **end for**
14: **end for**
15: **end for**
16: **return** $\hat{a} \leftarrow a$

An n-poit NTT operation can be carried out in $\log_2(n) - 1$ steps, with each step consisting of $\frac{n}{2}$ butterfly operations (see lines 4 through 14 of the Algorithm 1). Thus, it is possible to calculate an n-point NTT by performing $(\log_2(n) - 1) \cdot \frac{n}{2}$ butterfly operations. Using $u = a_j$, $v = a_{j+t}$, and S as inputs, a butterfly operation computes $(u + S \cdot v) \bmod q$ and $(u - S \cdot v) \bmod q$ where S is a power of ψ (lines 9-12 of Algorithm 1). A similar method, called GS butterfly, can be utilized for performing INTT. Using u, v, and S as inputs, the GS butterfly operation computes $(u+v) \bmod q$ and $((u-v)S) \bmod q$ where S is a power of ψ. We refer interested readers to [8] for more details.

C. Modular Reduction

Let us assume that $q = k \cdot 2^m + 1$ where $k \geq 3$ is an odd and small integer. These numbers are known as Proth numbers in the literature. As described before, the computation of NTT can be done based on the $2n$-th root of unity, which implies that $2n|2^m$. Consider two integers: $0 \leq a, b \leq q$. Then for the integer product of these integers, we have: $0 \leq C = a \cdot b \leq q^2 = k^2 2^{2m} + k2^{m+1} + 1$. [7] makes use of a special form of q to reduce C in the module of q. Let us consider C as $C = C_l + 2^m C_h$, where $0 \leq C_l < 2^m$. Since $k2^m = -1 \bmod q$, then $0 \leq C_h = \frac{(C-C_l)}{2^m} < k^2 2^m + 2k + \frac{1}{2^m} = kq + k + \frac{1}{2^m}$. salari: ? It is easy to see that $kC \equiv kC_l - C_h$ holds where $|kC_l - C_h| < (k + \frac{1}{2^m})q$ by considering the above bounds for C_l and C_h. [7] introduced the K-RED function, which takes $C = C_l + 2^m C_h$ as input and computes $D = kC_l - C_h$ as described in Algorithm 2. While D does not necessarily reduce C to $C \bmod q$, the K-RED function is usually called a reduction in the literature as D is quite close to the desired range. In the proposed method in this paper, we use the K-RED function.

Algorithm 2 K-RED function

Require: An integer $C = C_l + 2^m C_h$
Ensure: kC
1: $C_l = C \bmod 2^m$
2: $C_h = \frac{C}{2^m}$
3: **return** $kC_l - C_h$

III. The Proposed Low-complexity NTT Multiplication Algorithm

We describe a new computation of the butterfly operation that makes use of a particular pre-computation and can significantly improve the performance of the NTT implementation. In this regard, we provide two techniques, which are explained in Section III-A and Section III-B. Moreover, in Section III-C, a novel NTT algorithm using four butterfly cores is proposed.

A. A Cooley-Tukey Butterfly using Pre-computation and K-RED

We aim to implement CT butterfly, which takes $u, v \in Z_q$ as inputs and computes the outputs $u' = (u + S \cdot v) \bmod q$ and $v' = (u - S \cdot v) \bmod q$ where S is a power of ψ as described in Section II-B. In the first step of our technique, we divide v into two equal parts, v_h and v_l, each consisting of $\frac{|q|}{2}$ bits, where $|q|$ represents the size of q. In Kyber, v_h and v_l each contain six bits since $|q| = 12$. If the size of q is odd, a zero is added to the msb of q. Hence, v_h and v_l represent the v's $\frac{|q|}{2}$ most significant bits and $\frac{|q|}{2}$ least significant bits. To compute u' and v', it is necessary to find the value of $v \cdot S = (v_h \cdot 2^{\frac{|q|}{2}} + v_l) \cdot S$. To reduce computational complexity, we pre-compute the values of $v_h \cdot S$ and $v_l \cdot S$. Actually, we pre-compute the values of $z \cdot \psi^{br(\ell)} \cdot k^{-1} \bmod q$ for all $0 \leq z < 2^{\frac{|q|}{2}}$ and $1 \leq \ell \leq \frac{n}{2} - 1$, where $br(\ell)$ is the bit-reversed representation of ℓ ($\psi^{br(\ell)}$ instances for S) and $q = k \cdot 2^m + 1$ (k^{-1} is multiplied to allow using the K-RED function for reduction). Since $n = 256$ and $q = 3329$ in the instance of Kyber, we have $0 \leq \ell \leq 127$, and $k = 13$.

Our proposed method for performing CT butterfly makes use of Algorithm 3. The values of $z \cdot \psi^{br(\ell)} \cdot k^{-1} \bmod q$ are computed in the pre-computation step and stored in memory. The notation $M[\ell \| z]$ in Algorithm 3 instances for reading each of the $z \cdot \psi^{br(\ell)} \cdot k^{-1} \bmod q$ values from the memory. Instead of passing the value of S as input for the butterfly, only the value of ℓ is passed as input. To compute the value of C, v_h should first be multiplied by $2^{\frac{|q|}{2}}$ (i.e., shifted by $\frac{|q|}{2}$ bits) and then be added to the corresponding value of v_l. After applying K-RED reduction to C, which consists of $(|q| + \frac{|q|}{2} + 1)$ bits, C' can be calculated. Finally, $u' = u \oplus C'$ and $v' = u \ominus C'$ are computed. The operations $\oplus$ and $\ominus$ are modular addition and modular subtraction, respectively. These operations are slightly more complicated than regular addition and subtraction.

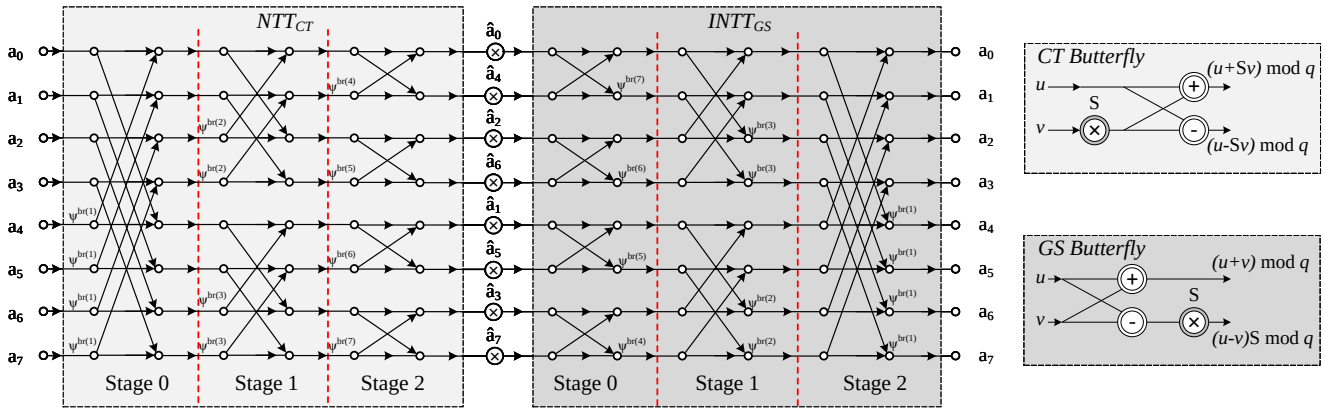

Fig. 1. NTT Multiplication of Two 8-degree polynomials using CT and GS Butterfly[21].

Algorithm 3 Proposed Cooley-Tukey Butterfly using Pre-computation and K-RED.

Require: $u = (u_{|q|-1}, u_{|q|-2}, \ldots, u_1, u_0)_2$, $v = (v_{|q|-1}, v_{|q|-2}, \ldots, v_1, v_0)_2$, ℓ, $q = k \times 2^m + 1$.
Ensure: u' and v'
1: $v_l = (v_{\frac{|q|}{2}-1}, \cdots, v_1, v_0)_2$
2: $v_h = (v_{|q|-1}, v_{|q|-2}, \cdots, v_{\frac{|q|}{2}})_2$
3: $C = M[\ell \| v_h] \times 2^{\frac{|q|}{2}} + M[\ell \| v_l]$
4: $C_l = (C_{m-1} \cdots, C_1, C_0)_2$
5: $C_h = (C_{|q|+\frac{|q|}{2}}, \cdots, C_m)_2$
6: $C' = k \cdot C_l - C_h$
7: $u' = u \oplus C'$
8: $v' = u \ominus C'$
9: **return** u', v'.

B. A Cooley-Tukey Butterfly without Reduction

In this part, we present a method for implementing CT butterfly without reduction. Similar to Algorithm 3, we make use of a pre-computation for performing CT butterfly to compute the outputs $u' = (u + S \cdot v) \bmod q$ and $v' = (u - S \cdot v) \bmod q$ for the inputs u and v. Algorithm 4 describes our implementation of CT butterfly, which requires no reduction. We pre-compute the values of $z \cdot \psi^{br(\ell)} \cdot k^{-1} \bmod q$ for all $0 \leq z < 2^{\frac{|q|}{2}}$ and $1 \leq \ell \leq \frac{n}{2} - 1$, and store them in memory. The notation $M_l[\ell \| z]$ in Algorithm 4 instances is for reading these values from the memory. In addition, we pre-compute the values of $z \cdot \psi^{br(\ell)} \cdot k^{-1} \cdot 2^{\frac{|q|}{2}} \bmod q$ for all $0 \leq z < 2^{\frac{|q|}{2}}$ and $1 \leq \ell \leq \frac{n}{2} - 1$, and store them in memory. The notation $M_h[\ell \| z]$ in Algorithm 4 instances for reading these values from the memory.

According to Algorithm 4, initially, the input v is divided into two $\frac{|q|}{2}$-bit parts, v_h and v_l. C can be obtained by computing the modular addition of $M_h[\ell \| v_h]$ and $M_l[\ell \| v_l]$. Finally, $u' = u \oplus C'$ and $v' = u \ominus C'$ are computed.

Algorithm 4 Proposed Cooley-Tukey Butterfly without Reduction.

Require: $u = (u_{|q|-1}, u_{|q|-2}, \ldots, u_1, u_0)_2$, $v = (v_{|q|-1}, v_{|q|-2}, \ldots, v_1, v_0)_2$, ℓ.
Ensure: u' (a 12-bit number), v' (a 12-bit number)
1: $v_l = (v_{\frac{|q|}{2}-1}, \cdots, v_1, v_0)_2$
2: $v_h = (v_{|q|-1}, v_{|q|-2}, \cdots, v_{\frac{|q|}{2}})_2$
3: $C = M_h[\ell \| v_h] \oplus M_l[\ell \| v_l]$
4: $u' = u \oplus C$
5: $v' = u \ominus C$
6: **return** u', v'.

C. Proposed Algorithm for NTT based on 4 Butterfly Cores

In this part, using the proposed butterflies, an algorithm for NTT is proposed. It is assumed that four CT butterfly cores are utilized (which we refer to as BFC_0, BFC_1, BFC_2, BFC_3) to execute four butterfly operations in each stage concurrently. As seen in Algorithm 1, the computation of NTT consists of multiple stages (7 stages in Kyber). All butterflies' S values in the first stage are $\psi^{br(1)}$. In the second stage, the S value for the first half of butterflies is $\psi^{br(2)}$ and for the second half, is $\psi^{br(3)}$. Similar characteristics apply to later stages. That means the S value of several consecutive butterflies in each stage is comparable. Using this observation, we propose a method for performing the butterfly operations in parallel and in a memory-efficient manner with four cores, as shown in Algorithm 5. Each stage contains $\frac{n}{2}$ butterfly operations (in the case of Kyber, 128 butterfly operations). The butterfly operation c is assigned to $BFC_{\lfloor \frac{c}{\frac{n}{8}} \rfloor}$ where $0 \leq c < \frac{n}{2}$. Thus, Algorithm 5 reduces the redundancy and size of the dedicated memory required by each butterfly core (for storing pre-computed values).

IV. Efficient and High-speed Multiplier Architecture

In this section, using Algorithm 3 and Algorithm 4 a unified CT/GS butterfly core is proposed. Then, using the

Algorithm 5 NTT based on four CT Butterfly Cores.

Require: A polynomial $a(x) \in Z_q[X]/(X^n+1)$.
Ensure: $\hat{a}(x) = NTT(a(x)) \in Z_q[X]/(X^n+1)$
1: $\ell = 1$
2: **for** $s = \frac{n}{2}, s \geq 2, s = s \gg 1$ **do**
3: $c = 0$
4: **for** $i = 0, i < n, i = j + s$ **do**
5: **for** $j = i, j < i + s, j = j + 4$ **do**
6: **if** $c < \frac{n}{8}$ **then**
7: $(a_j, a_{j+s}) \leftarrow BFC_0(a_j, a_{j+s}, \ell)$
8: $c = c + 1$
9: **else if** $c < 2 \cdot \frac{n}{8}$ **then**
10: $(a_j, a_{j+s}) \leftarrow BFC_1(a_j, a_{j+s}, \ell)$
11: $c = c + 1$
12: **else if** $c < 3 \cdot \frac{n}{8}$ **then**
13: $(a_j, a_{j+s}) \leftarrow BFC_2(a_j, a_{j+s}, \ell)$
14: $c = c + 1$
15: **else**
16: $(a_j, a_{j+s}) \leftarrow BFC_3(a_j, a_{j+s}, \ell)$
17: $c = c + 1$
18: **end if**
19: **end for**
20: $\ell = \ell + 1$
21: **end for**
22: **end for**
23: **return** $\hat{a}(x) = a(x)$.

proposed butterfly core and Algorithm 5, a high-speed and low-complexity NTT multiplier is suggested. Finally, the area and time complexity of the proposed multiplier are described.

A. Proposed Unified CT/GS Butterfly Core

We design a reconfigurable architecture capable of performing both CT and GS in a single module, as illustrated in Figure 2-a. As we will explain later in this part, this module can perform point-wise multiplication using Type-0 and Type-1 of butterfly, as illustrated in Figure 2-a. The Type-0 of the butterfly core requires five multiplexers, one modular addition, one modular subtraction, one modular multiplier, and three 12-bit registers for storing the outputs. This design is executable using either Algorithm 3 or Algorithm 4. The type of selected algorithm affects the modular multiplier design.

Figure 2-b depicts the application of Algorithm 3 in this scenario. The modular multiplier (lines 3-6 in Algorithm 3) includes the following operations: $(|q|+\frac{|q|}{2})$-bit addition (line 3), multiplication by a constant number (k) and $(|q|+|k|)$-bit subtraction (line 6). In the case of Kyber, the operations are an 18-bit adder (line 3), 10-bit, 11-bit and 13-bit adders/subtractors (line 6). The design based on Algorithm 3, has a light reduction. Besides, the multiplier is realized by a $|q|$-bit adder.

Figure 2-c depicts the application of Algorithm 4. The modular multiplier (line 3 of Algorithm 4) includes a single $|q|$-bit modular adder. This design consists of addition and subtraction operations with almost the same level of complexity as conventional addition and subtraction. However, the overall design complexity is very low due to the absence of a reduction module.

For the proposed core to allow point-wise multiplication, it must be able to multiply two arbitrary $|q|$-bit numbers as follows:

$$\begin{aligned} C = A \cdot B = (A_h \cdot 2^{\frac{|q|}{2}} + A_l) \cdot (B_h \cdot 2^{\frac{|q|}{2}} + B_l) = \\ A_h \cdot B_h \cdot 2^{|q|} + A_h \cdot B_l \cdot 2^{\frac{|q|}{2}} + A_l \cdot B_h \cdot 2^{\frac{|q|}{2}} + A_l \cdot B_l = \\ (A_h \cdot B_h \cdot 2^{\frac{|q|}{2}} + (A_h \cdot B_l + A_l \cdot B_h)) \cdot 2^{\frac{|q|}{2}} + A_l \cdot B_l. \end{aligned} \tag{1}$$

According to Equation (1), each point-wise multiplication consists of two modular multiplications, as shown in Algorithm 3 and Algorithm 4. We should precompute and store $a \times b \times k^{-1}$ where $0 < a, b < 2^{\frac{|q|}{2}}$ when the modular multiplication of Algorithm 3 is used. We should precompute and store $a \times b$ where $0 < a, b < 2^{\frac{|q|}{2}}$ when the modular multiplication of Algorithm 4 is used.

We consider a 2-bit control signal. When the control signal is "1x", $(u + \psi v) \bmod q$ and $(u - \psi v) \bmod q$ are computed using the CT butterfly. $(u + v) \bmod q$ and $(u - v)\psi \bmod q$ are computed using the GS butterfly When the control signal is "x1". When the control signal is "00", the point-wise multiplication is performed. Taking Equation (1) into consideration, this point-wise multiplication requires two proposed modular multiplier (Figure 2-b or Figure 2-c). For this purpose, we consider type-0 and type-1 butterfly architectures. Type-0 performs the inner multiplication in Equation (1) $(A_h \times B_h \times 2^{\frac{|q|}{2}} + (A_h \times B_l + A_l \times B_h))$. Type-1 performs the outer multiplication. Hence, two consecutive butterflies perform one point-wise multiplication.

B. NTT Multiplier

In this part, using four proposed reconfigurable butterfly cores and algorithm 5 a high-speed and low-complexity NTT multiplier is suggested. In the following, each part of the multiplier is described.

Pipelined architecture: Our architecture is illustrated in Figure 3. In order to perform an NTT multiplication, the architecture is capable of performing all NTT/INTT conversions, and polynomial base multiplication.

Butterfly cores: We consider an implementation of NTT proposed in Algorithm 5, which has four butterfly cores. In order to let the butterfly cores operate in parallel, eight instances of memory are necessary (to read eight inputs). Each butterfly core has its own ROM memory, which includes some precomputed values $(a \times \psi^{br(i)}\ (\times k^{-1}) \bmod q$ for NTT/INTT and $a \times b\ (\times k^{-1}) \bmod q$ for the point-wise multiplier).

ROM Memories: As described in Algorithm 5, all the butterfly operations of the first stage work with $\psi^{br(1)}$. Hence, each butterfly core for the first stage requires $(1 \times 2^{|q|} \times |q|)$-bit memory. Half of the butterfly operations of the second stage work with $\psi^{br(2)}$, the other ones work with $\psi^{br(3)}$. So, each butterfly core for the second stage requires $(1 \times 2^{\frac{|q|}{2}} \times |q|)$-bit memory. Similarly, the butterfly operations of the i-th stage work with 2^{i-1} various powers of ψ. Hence, each butterfly

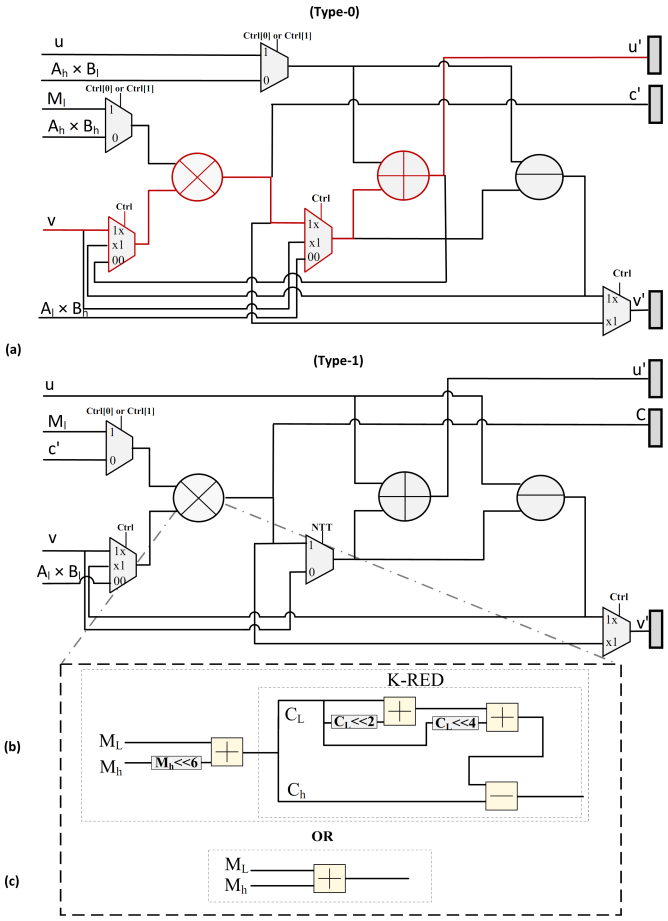

Fig. 2. The Proposed Re-configurable CT/GS Butterfly Core. a) Type-0 and Type-1 of the re-configurable butterfly. b) The proposed modular multiplier with a light reduction. c) The proposed modular multiplier without reduction.

core for the i-th stage requires $(\lceil \frac{2^{i-1}}{4} \rceil \times 2^{\frac{|q|}{2}} \times |q|)$-bit memory. So, each butterfly requires $(\sum_{1}^{\frac{|q|}{2}} \lceil \frac{2^i}{4} \rceil \times 2^{\frac{|q|}{2}} \times |q|)$-bit ROM memory in total for all the stages.

Multiplexers: After performing butterfly operations, the outputs $(v^{'}\ u^{'})$ of each butterfly core should be stored in RAM. These results will be used by a butterfly core in the next round. Hence, the RAM should be attached to the corresponding butterfly core in the next round. To this en, the butterfly cores are routed to the corresponding RAM by means of eight 8-to-1 multiplexers.

Control Unit: We designed the control unit using an FSM. All control signals, including multiplexer selection signals, and the read/write address of the precomputed values are calculated in this unit, which is not illustrated for the sake of brevity.

C. Area and Time Complexity Analysis

The area and time complexity of butterfly cores, NTT/INTT modules, and point-wise multipliers are reported in Table I. The area and time complexity of the modular multipliers, which are designed based on Figure 2-b and Figure 2-c are illustrated on rows 1 and 2 of the table, respectively. Table I

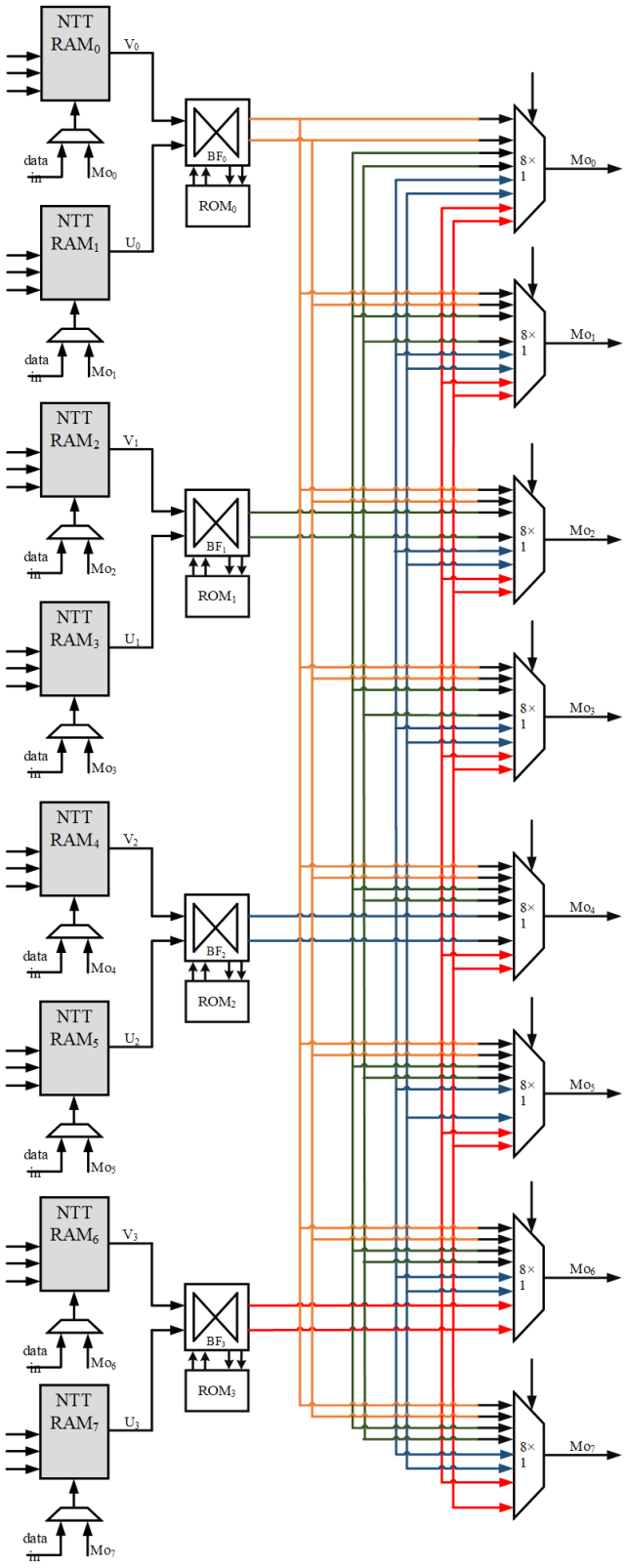

Fig. 3. The Proposed Pipelined NTT Multiplier.

TABLE I
AREA AND TIME COMPLEXITIES ESTIMATION OF NTT MULTIPLIER USED IN KYBER

	#Butterfly Core	# 12-bit Modular Multiplier	# 12-bit Modular Adder	# 12-bit Modular subtractor	# 12-bit 4-to-1 Multiplexer	# FFs	CPD	Latency
#Modular Multiplier1	-	-	3	1	0	0	3adder+1subtractor	-
#Modular Multiplier2	-	-	1	0	0	0	1adder	-
#Butterfly Core	-	1	1	1	4	36	3mux+1mult+1sub	1
NTT	4	4	4	4	$4\times4+4\times2.5$	$4\times2\times2\times12$	Butterfly Core	$(7\times32)+(6\times4)+4-1=251$
INTT	4	4	4	4	$4\times4+8\times2.5$	$4\times2\times2\times12$	Butterfly Core	$(7\times32)+(6\times4)+4-1=251$
Point-Wise Multiplication	4	4	4	4	$4\times4+8\times2.5$	$4\times2\times2\times12$	Butterfly Core	$\frac{128\times5}{2}=640$

TABLE II
COMPARISON OF NTT MULTIPLIER IMPLEMENTATION RESULT ON FPGA (USING KYBE PARAMETER; N = 265, Q = 3,329)

Reference	Platform	Butterfly	NTT/INTT cycles	Freq MHz	Time (μs)	LUTs	FFs	DSP	BRAM	Speedup (latency)	$A\times C$	$A\times T$
[22]	Zynq-7000	2	1,935/1,930	-	-	2,908	170	9	0	7.71	5.6 (94.6%)	25.3 (96.4%)
[12]	Virtex-7	1	43,756/-	-	-	417	462	0	0	174.33	18.2 (98.4%)	82.2 (98.9%)
[23]	Artix-7	1	6,868/6,367	59	116.41	-	-	-	-	27.36	-	-
[24]	Artix-7	2	1,834/-	155	11.83	-	-	-	-	7.31	-	-
[25]	Artix-7	2	512/576	161	3.18	1.737	1.167	2	3	2.04	0.9 (77.8%)	5.5 (83.6%)
[21]	Artix-7	2×2	324/324	222	1.46	801	717	4	2	1.30	0.3 (33.3%)	1.2 (25%)
proposed2	Artix-7	4	251/251	285	0.88	644	260	0	28	1.00	0.2	0.6
proposed1	Virtex-7	4	251/251	250	1.00	912	260	0	16	1.00	0.2	0.9

indicates that each butterfly core contains one modular multiplier, one modular adder, one modular subtractor, four 4-to-1 multiplexers, and three 12-bit registers. Moreover, according to the butterfly presented in Algorithm 3 and Algorithm 3, each modular multiplier can consist of only one adder (Figure 2-c) or four adders/subtractors (Figure 2-b).

The unified architecture of NTT/INTT and point-wise multiplier is depicted in Figure 3. The architecture contains four butterfly cores. Each core requires two RAMs for reading the coefficients of the polynomial and one ROM for reading the precomputed values (i.e. $a\times\psi^{br(i)}\times k^{-1}$ mod q for NTT/INTT and $a\times b\times k^{-1}$ mod q for point-wise multiplier). After butterfly computations, eight 8-to-1 multiplexers (can be implemented using almost 2.5 4-to-1 multiplexers) will direct the results (v' and u') to the appropriate RAMs. Namely, the RAMs from which the butterfly core will read in the subsequent rounds.

Our NTT multiplier architecture is a pipeline architecture with four stages: reading from BRAMs, butterfly operation, direct the results to the appropriate RAMs (using multiplexers), and writing to BRAMs. Each NTT includes seven levels, each of which performs 128 butterflies. Given that our architecture has four cores, this multiplier must be called $\frac{128\times7}{4}=7\times32$ times to perform one NTT. Considering the four stages of the proposed pipeline architecture and delaying four clock cycles between each NTT level to unload the pipeline (6×4), the latency of this pipeline architecture is $(7\times32)+(6\times4)+4-1=251$.

The critical path delay (CPD) of the NTT design and the unified butterfly core are identical. Figure 2-a depicts the crucial path in the butterfly core in red: a multiplexer, a multiplier, another multiplexer, and finally an addition.

V. RESULTS

We chose FPGA to implement our proposed architecture because FPGA is reconfigurable hardware and offers an acceptable trade-off between flexibility and performance. It has also been the focus of the majority of research. FPGA contains a variety of resources, including LUTs, Flip-Flops, DSPs, distributed RAMs, Block RAMs (BRAMs), and others. Using BRAMs is restricted by read-write bandwidth limitations, read-write times, and other factors. As a result, BRAMs are utilized less frequently compared to the other sources. The overhead of our method for storing pre-computed values consists of utilizing Block RAMs (BRAMs) which are utilized less frequently compared to other sources.

The proposed methods in the literature cannot be directly compared due to the fact that they employ different FPGA target devices and have different optimization aims. As a basis for comparison, we have chosen earlier works that implemented NTT on the FPGA platform. The results of resource consumption and performance are given in Table II. Table II displays $A\times T$ and $A\times C$ for each implementation, where A, T, and C stand for the utilized LUT, time in μs, and latency, respectively. Our suggested architecture is synthesized with Xilinx Vivado 2019.2 and implemented on a NIST-recommended Xilinx Artix-7 FPGA chip.

The proposed design in [12] does not require DSP, albeit at the expense of a large increase in latency. Both of our proposed designs in this study have a latency of 251 cycles, which is significantly smaller than 43,756 cycles (the latency of [12]). Compared to other DSP-required architectures, ours offers reduced latency. Furthermore, our design's frequency is superior to previous works. The best-known architecture is that proposed in [21]. Relative to [21], our architecture with fewer BRAMs has reduced latency, improved time by 31%, and increased frequency by 13%. Our architecture has a smaller area than [21] since it does not require DSP, and the number of registers is reduced while the number of LUTs is somewhat increased. Our design employs additional BRAMs, which are not essential for the majority of applications.

VI. CONCLUSION

The Number Theoretic Transform (NTT) is able to effectively execute polynomial multiplication. Hence, NTT has become an essential feature in the implementation of lattice-based post-quantum algorithms, which are the next generation of standard cryptography systems. In this research, we showed how to make a fast NTT multiplier that is highly optimized for FPGAs with limited logical resources. Our method for storing intermediate results makes use of Block RAMs (BRAMs),

which are commonplace in the majority of the FPGAs. As a result of our novel architecture, the time required to perform polynomial multiplication using NTT is decreased by 28%, and the $A \times T$ (area $\times$ time) is improved by 25%, compared to the best-known technique. It is also worth noting that we were able to achieve these enhancements without requiring DSP.

REFERENCES

[1] N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. A. Huss, "On the design of hardware building blocks for modern lattice-based encryption schemes," in *CHES*, ser. Lecture Notes in Computer Science, vol. 7428. Springer, 2012, pp. 512–529.

[2] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *LATINCRYPT*, ser. Lecture Notes in Computer Science, vol. 7533. Springer, 2012, pp. 139–158.

[3] ——, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, vol. 8282. Springer, 2013, pp. 68–85.

[4] A. Aysu, C. Patterson, and P. Schaumont, "Low-cost and area-efficient FPGA implementations of lattice-based cryptography," in *HOST*. IEEE Computer Society, 2013, pp. 81–86.

[5] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," in *CHES*, ser. Lecture Notes in Computer Science, vol. 8731. Springer, 2014, pp. 371–391.

[6] N. Zhang, B. Yang, C. Chen, S. Yin, S. Wei, and L. Liu, "Highly efficient architecture of newhope-nist on FPGA using low-complexity NTT/INTT," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 2, pp. 49–72, 2020.

[7] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *CANS*, ser. Lecture Notes in Computer Science, vol. 10052, 2016, pp. 124–139.

[8] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 4, pp. 17–61, 2019.

[9] Z. Chen, Y. Ma, T. Chen, J. Lin, and J. Jing, "Towards efficient kyber on fpgas: A processor for vector of polynomials," in *ASP-DAC*. IEEE, 2020, pp. 247–252.

[10] A. C. Mert, E. Karabulut, E. Öztürk, E. Savas, M. Becchi, and A. Aysu, "A flexible and scalable NTT hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *DATE*. IEEE, 2020, pp. 346–351.

[11] A. C. Mert, E. Karabulut, E. Öztürk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Trans. Computers*, vol. 71, no. 11, pp. 2829–2843, 2022.

[12] E. Karabulut and A. Aysu, "RANTT: A RISC-V architecture extension for the number theoretic transform," in *FPL*. IEEE, 2020, pp. 26–32.

[13] T. Fritzmann and J. Sepúlveda, "Efficient and flexible low-power NTT for lattice-based cryptography," in *HOST*. IEEE, 2019, pp. 141–150.

[14] Y. Xing and S. Li, "An efficient implementation of the newhope key exchange on fpgas," *IEEE Trans. Circuits Syst. I Regul. Pap.*, vol. 67-I, no. 3, pp. 866–878, 2020.

[15] C. Du, G. Bai, and X. Wu, "High-speed polynomial multiplier architecture for ring-lwe based public key cryptosystems," in *ACM Great Lakes Symposium on VLSI*. ACM, 2016, pp. 9–14.

[16] P. Kuo, Y. Chen, Y. Hsu, C. Cheng, W. Li, and B. Yang, "High performance post-quantum key exchange on fpgas," *J. Inf. Sci. Eng.*, vol. 38, no. 4, pp. 1211–1229, 2022.

[17] D. T. Nguyen, V. B. Dang, and K. Gaj, "A high-level synthesis approach to the software/hardware codesign of ntt-based post-quantum cryptography algorithms," in *FPT*. IEEE, 2019, pp. 371–374.

[18] ——, "High-level synthesis in implementing and benchmarking number theoretic transform in lattice-based post-quantum cryptography using software/hardware codesign," in *ARC*, ser. Lecture Notes in Computer Science, vol. 12083. Springer, 2020, pp. 247–257.

[19] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "Crystals-kyber algorithm specifications and supporting documentation," *NIST PQC Round*, vol. 2, no. 4, pp. 1–43, 2017.

[20] T. Pöppelmann, T. Oder, and T. Güneysu, "High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers," in *LATINCRYPT*, ser. Lecture Notes in Computer Science, vol. 9230. Springer, 2015, pp. 346–365.

[21] M. Bisheh-Niasar, R. Azarderakhsh, and M. M. Kermani, "Area-time efficient hardware architecture for signature based on ed448," *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 68, no. 8, pp. 2942–2946, 2021.

[22] T. Fritzmann, G. Sigl, and J. Sepúlveda, "RISQ-V: tightly coupled RISC-V accelerators for post-quantum cryptography," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 4, pp. 239–280, 2020.

[23] E. Alkim, H. Evkan, N. Lahr, R. Niederhagen, and R. Petri, "ISA extensions for finite field arithmetic accelerating kyber and newhope on RISC-V," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 3, pp. 219–242, 2020.

[24] Y. Huang, M. Huang, Z. Lei, and J. Wu, "A pure hardware implementation of CRYSTALS-KYBER PQC algorithm through resource reuse," *IEICE Electron. Express*, vol. 17, no. 17, p. 20200234, 2020.

[25] Y. Xing and S. Li, "A compact hardware implementation of cca-secure key exchange mechanism CRYSTALS-KYBER on FPGA," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 2, pp. 328–356, 2021.

Raziyeh Salarifard received the B.Sc. degree from the Sharif University of Technology ,Tehran, Iran, in 2012, the M.Sc. and PhD degrees from the same university, in 2014 and 2019 respectively, all in computer engineering (hardware). She is now working as an Assistant Professor at the Faculty of Computer Science and Engineering, Shahid Beheshti University. Her research interests include hardware security, cryptographic engineering, and secure, efficient computing and architectures.

Hadi Soleimany has been working as an Assistant Professor at Cyberspace Research Institute at Shahid Beheshti University, Iran since 2015. He received his PhD in Theoretical Computer Science from Aalto University, Finland in 2015. He was also a postdoctoral researcher at Technical University of Denmark (DTU), Denmark in Summer 2016 and 2017. His main research interests are practical aspects of cryptography.