Tensor Based Multivariate Polynomial Modulo Multiplier for Cryptographic Applications

Bikram Paul, Angana Nath, Srinivasan Krishnaswamy, Jan Pidanic, Zdenek Nemec and Gaurav Trivedi

Abstract—

Modulo polynomial multiplication is an essential mathematical operation in the area of finite field arithmetic. Polynomial functions can be represented as tensors, which can be utilized as basic building blocks for various lattice-based post-quantum cryptography schemes. This paper presents a tensor-based novel modulo multiplication method for multivariate polynomials over $GF(2^m)$ and is realized on the hardware platform (FPGA). The proposed method consumes $6.5\times$ less power and achieves more than $6\times$ speedup compared to other contemporary single variable polynomial multiplication implementations. Our method is embarrassingly parallel and easily scalable for multivariate polynomials. Polynomial functions of nine variables, where each variable is of degree 128, are tested with the proposed multiplier, and its corresponding area, power, and power-delay-area product (PDAP) are presented. The computational complexity of single variable and multivariate polynomial multiplications are $O(n)$ and $O(np)$, respectively, where n is the maximum degree of a polynomial having p variables. Due to its high speed, low latency, and scalability, the proposed modulo multiplier can be used in a wide range of applications.

Index Terms—Multivariate polynomial, Cryptography, Tensor, Homomorphic encryption, Modulo multiplication, Field programmable gate array (FPGA).

1 Introduction

With the rapid advancement of computational capabilities, modern cloud computations, network data transactions, etc., are facing tremendous security threats against various unwanted adversaries. Post-quantum cryptography (PQC) schemes exhibit stronger resistance to classical and quantum computing-based attacks and have become prime interest to the cryptography community. Among various classes of PQC (hash-based, lattice-based, code-based, supersingular isogeny, chaotic dynamic system based, etc.), the lattice-based encryption schemes have gotten the most attention while designing alternate cryptosystems due to their easy implementation on the hardware. For post-quantum cryptosystems, many other security-related applications, such as homomorphic encryption, key generation, encapsulation, and hash or signature generation protocols, can be derived from the polynomial lattice. In general, the lattice-based cryptosystems are implemented using polynomial rings and execute modulo multiplications and additions of two large-degree polynomials over $GF(2^m)$ (Galois Field) basis, which is a compute intense process.

Many efficient polynomial modulo multipliers are proposed in the literature, which play a critical role in lattice-based encryption schemes. Among them, Bit-parallel canonical methods [1], [2], [3], [4], systolic multipliers based on Montgomery techniques [5], [6], digit-serial and systolic Karatsuba methods [7], [8], [9], digit-serial dual basis multipliers [10], [11], number theoretic transform (NTT) based multipliers [12], [13], matrix-vector based Hankel and Toeplitz multiples [14], [15], [16], [17] are quite well-known. The salient features and limitations of the above multipliers are listed in Table 1. Most of these works primarily focus on implementing single variable polynomials and deal with trinomials and pentanomials only, whereas our proposed method is generic and optimally implemented on the FPGA. Various FPGA hardware implementations of polynomial modulo multipliers can be found in [7], [8], [9], [13], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]. Further, the efficiency of our proposed multiplier is estimated using various parameters, such as computational complexity, resource utilization, and scalability with a polynomial degree. It is found that the proposed method has less computational complexity then [4], [25] and consumes fewer hardware resources with respect to [9], [13], [16], [17], [18]. It also exhibits higher scalability with a greater polynomial degree than the methods presented in [7], [8], [15], [19].

In this work, our main contribution is to develop a generic, scalable power-efficient multivariate polynomial multiplier, which can provide less delay so that it can be used in various time-critical applications. Our proposed multiplier works efficiently with Learning-with-Errors (LWE) and Ring-LWE (RWE) based homomorphic

- *Bikram Paul is PhD research scholar with the Department of Electronics and Electrical Engineering, Indian Institute of Technology, Guwahati, Assam, 781039 India, e-mail: bikram@iitg.ac.in*
- *Angana Nath is Masters scholar with the Department of Electronics and Electrical Engineering, Indian Institute of Technology, Guwahati, Assam, 781039 India, e-mail: angananath7@gmail.com*
- *Srinivasan Krishnaswamy and Gaurav Trivedi are Professor at Indian Institute of Technology, Guwahati, e-mails: trivedi@iitg.ac.in and srinikris@iitg.ac.in*
- *Jan Pidanic and Zdenek Nemec are Professor at University of Pardubice, Pardubice, Czech Republic, e-mails: jan.pidanic@upce.cz and zdenek.nemec@upce.cz*

The work has been supported jointly by the Ministry of Education, Youth and Sports, Czech Republic, through a project entitled "Artificial Intelligence Enabled Smart Contactless Technology Development for Smart Fencing" and Department of Science and Technology, India through a project entitled "Smart Contactless Technology Development for Smart Fencing". The support provided by above mentioned funding agencies is greatly acknowledged.

TABLE 1
Comparison of Existing Polynomial Modulo Multiplication Methodologies

Existing Methodologies	Salient Features	Limitations	Reference
Bit-parallel Canonical Multiplier	Multiplexer based approach in the construction of sub-quadratic complex space multipliers	Only irreducible trinomial is employed in this scheme.	[1], [2], [3]
Systolic Montgomery Multiplier	Efficient implementation of Montgomery multiplication over GF(2^m) for all-one polynomials	Produces estimated value of quotients, which is useful for RSA and DiffieHellman key exchange, but it is not applicable for lattice based PQC algorithms	[5], [6]
Dual-basis digit serial multiplier	This method is based on irreducible trinomial and a look-ahead technique in the dual basis multiplication, processed by one cell of tree structure in the most significant digit.	Limited to irreducible trinomial only, and cannot be applied for general multivariate polynomial operations	[10], [11]
Hankel matrix-vector Multiplier	This is an iterative Fast Fourier Transform based sparse matrix multiplication method with high degree of accuracy	Slower throughput as compared to most of the methods	[14], [15], [16]
Digit-serial Karatsuba Multiplier	This is a fast multiplication algorithm for multi-precision numbers with O($m^{1.58}$) complexity	Less throughput due to longer gate delays	[7], [8]
Chiou's Multiplier	The multiplication block is decomposed in four mutually independent sub-multiplication units, which improves the performance	Less scalability and more hardware resource intense process	[19]
Toeplitz matrix-vector Multiplier	A single digit is represented by 2-bits and polynomial multiplication in every clock cycle	Higher resource utilization for hardware implementation	[17]
Systolic Karatsuba Multiplier	Efficient implementation of single variable NIST polynomial based multiplier	Less throughput, less scalable and more power consumption	[9]
Number Theoretic Transform (NTT) Multiplier	Efficient implementation of single variable Kyber scheme polynomial for FFT like architecture	No implementation exists for the multivariate polynomial multiplications	[12], [13]

encryption schemes. This method is embarrassingly parallel and substantially improves the proposed method's throughput. Along with sequential design, 4-parallel column multiplication (4-PCM) and 16-PCM implementations are presented in Table 9. The computational complexity of the proposed modulo multiplier is linear for single variable polynomial multiplications. The tensor matrices and input vectors employed in this method are sparse, which optimizes the space requirement of the proposed method.

The manuscript is organized as follows. Section II presents preliminary research works which lay the foundation of our proposed hardware architecture. Section III presents the proposed tensor-based multivariate polynomial multiplier with a suitable example. The time and space complexity are discussed here with an appropriate comparison. Section IV explains the software and hardware implementations of the proposed multiplier. Experimental and implementation results, along with the detailed discussion and comparison, are drawn in Section V. The application of the proposed modulo polynomial multiplier in a basic lattice-based homomorphic encryption scheme along with its detailed formulation and security analysis is illustrated in section VI. Finally, the conclusion of the proposed work is presented in Section VII.

2 Preliminaries

Many existing implementations are based on a single variable polynomial and are limited to only a few polynomial coefficients, such as trinomials and pentanomials. Gröbner basis is constructed with unique algorithmic properties to provide easy solutions for many fundamental problems in a polynomial ring over a field. Let us assume F is a set of polynomials in the ring R, i.e. $f_1(x_1, x_2, \cdots, x_n), f_2(x_1, x_2, \cdots, x_n), \ldots, f_n(x_1, x_2, cdot\cdot, x_n)$. The Gröbner basis G is constructed using the above-mentioned conditions by ordering [26] monomials in a set of polynomials. The proposed multivariate modulo polynomial multiplier is conceptualized based on the following three polynomial multiplications presented in [27], [28], [29] for single and multivariate polynomial expressions. The lexicographic ordering employed in the proposed method is based on monomial ordering [30], [31].

Polynomials over the field GF(2^m) have coefficients from domain $\{0, 1\}$ and the highest degree of the polynomial is $(m - 1)$. For $m = 16$, a GF (2^{16}) multiplier for a single variable polynomial is presented in [27]. Figure 1 exhibits the structural organization of a single variable polynomial operation.

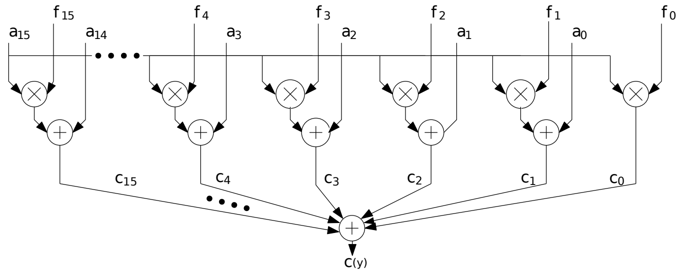

Fig. 1. The structural construction of GF (2^{16}) multiplier for a single variable polynomial

According to the Galois field, multiplication and addition are the primary operations in polynomial modular multiplication [27]. Therefore, if $a(y)$ and $b(y)$ be two polynomials in $GF(2^m)$, the product polynomial $c(y)$ can be expressed as equation 1.

$$c(y) = a(y).b(y) \mod f(y) \tag{1}$$

Initially, $a(y)$ is multiplied sequentially by the coefficients of $b(y)$. In this procedure, $a(y)$ is multiplied first with y; after that, $a(y)y$ modulo $f(y)$ is determined. Subsequently, $a(y)$ is multiplied by y^2, and $a(y)y^2$ modulo $f(y)$ is calculated. This process continues until the limit of the Galois field dimension is reached. Multiplying $a(y)$ with $b(y)$ and then dividing by $f(y)$ results in equation 2.

$$(a_{15}f_{15}+a_{14})y^{15}+(a_{15}f_{14}+a_{14})y^{14}\ldots\ldots \\ +(a_{15}f_1+a_0)y+a_{15}f_0 \tag{2}$$

A univariate *SISO* (serial-in-serial-out) modulo multiplier for polynomial modulo multiplication is proposed in [28]. The computational complexity of the SISO multiplier is $O(m)$. This algorithm is suitable for cryptographic applications due to less area utilization and higher output rate. All the relevant data and parameters related to the SISO multiplier over GF (2^m) can be found in [28].

The multivariate polynomial residue number system (MPRNS) based algorithm is presented in [29] to compute equation 3 mentioned below, where, $A(x)$, $B(x)$ and $C(x)$ $\in R[x]$.

$$C(x)=A(x).B(x).\ \bmod \prod_{i=1}^{L}(x_i^{N_i\pm 1}) \tag{3}$$

The quotient polynomial ring, $R[x]=Z_p/\prod_{i=1}^{L}(x_i^{N_i\pm1})$, has coefficients from the modular ring Z_p. The existence of an isomorphism between rings $R[x]$ and $Z_p^{N_1,N_2,\ldots,N_L}$ forms the basis of MPRNS(L) for L-variate polynomials.

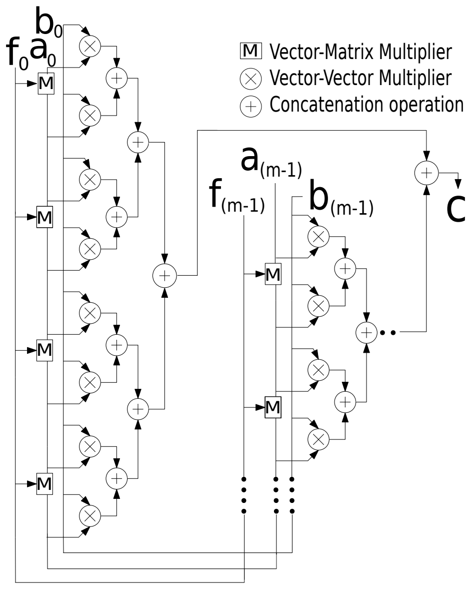

Fig. 2. The Structural Construction of GF (2^m) Multiplier for Multivariate Polynomial

Figure 2 exhibits the primary structural organization of the multiplication of two polynomials. This architecture can be theoretically scaled up for multiple polynomials by implementing identical blocks. Because of its sequential architecture, the throughput of this method is limited, and delay and power penalties are imposed. This methodology is scaled up for generic multivariate modulo multiplication by introducing tensor matrix multiplication techniques described in the following sections.

3 Tensor Based Polynomial Modulo Multiplication

3.1 Tensor Basics

A tensor, T, is a multi-linear map, such that $T: V_1\times V_2\times \ldots V_n\rightarrow W$, where $V_1,V_2,\ldots,V_n$ and W are the vector spaces of finite dimensions. The tensor can be expressed as a multidimensional array if vector spaces have a fixed base. For a bilinear map, one element from two vector spaces maps into an element in the third vector space. Let V_1 and V_2 be two vector spaces; a bilinear map B takes one element each from the vectors V_1 and V_2 and maps it to a third vector V_3 using the following steps.

1) $B: V_1\times V_2\leftarrow V_3$
2) If $v_1\in V_1$ is fixed, then $v_1\rightarrow B(v_1,v_2)$ is a linear function from $V_1\rightarrow V_3$
3) If $v_2\in V_2$ is fixed, then $v_2\rightarrow B(v_1,v_2)$ is a linear function from $V_2\rightarrow V_3$

Here, the tensor product of the linear maps is described briefly for completeness. Let $L_1: V_1\rightarrow V_2$ and $L_2: W_1\rightarrow W_2$ be two linear maps, then the tensor product of two linear maps is a linear map, $L_1\otimes L_2: V_1\otimes W_1\rightarrow V_2\rightarrow W_2$, which can be expressed by equation 4.

$$(L_1\otimes L_2)(v_1\otimes w_1)\rightarrow L_1(v_1)\otimes L_2(w_1) \tag{4}$$

A tensor product of two linear maps, A and B, can be represented as shown below.

$$A\otimes B=\begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B\\ a_{21}B & a_{13}B & \cdots & a_{2n}B\\ \cdot & \cdot & \cdot & \cdot\\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B\end{bmatrix}$$

Therefore, it can be stated that the polynomial multiplication is a tensor map because polynomials can be expressed as vectors in the polynomial space of n variables (i.e., $x_1,x_2,\ldots,x_n$ with the maximum degree of each variable $<d$). Thus it forms a vector space, V, of dimension d^n. Multiplication of polynomials in V *modulo* P can be considered as a function, f, which maps two elements of V into another element of V. Here, $p\in\mathbb{F}_2\ [x_1,x_2,\ldots,x_n]$, such that the degree of $x_1,x_2,\ldots,x_n$ in any monomial is d, i.e. $f: V\times V\rightarrow V$. Here, P is an irreducible polynomial in $\mathbb{F}_2$. This manifests f to a bilinear map enabling it to be employed for the tensor formulation.

The general modulo multiplication method requires multiplication of polynomials if the quotienting polynomial or reducible polynomial is changed. The main advantage of the tensor matrix method is to compute the tensor only once for a particular reducible polynomial to perform modulo multiplication. For a given reducible polynomial and the corresponding possible set of quotienting polynomials, a set of tensor coefficients is determined, which is employed to compute all the modulo multiplications until the reducible polynomial is changed. In the case of lattice pollycracker-based methods, the reducible polynomials do not alter much for a particular key pair and the security parameters. Therefore, the tensor matrix method can be of great advantage in reducing compute-intensive matrix operations. This method is elaborated below with a suitable example for completeness.

3.2 The Tensor Matrix Method

The proposed tensor based multiplication technique is derived from a fully homomorphic encryption scheme reported in [32]. The generic description of our proposed method is given below.

Let p_1 and p_2 be the polynomials in an ideal $\mathcal{I}_{\leq d}$ and $y \in \mathbb{Z}_q^n$, where $\mathbb{Z}$ is an integer and q is a prime number. $\mathbb{Z}_q$ denotes a finite field of cardinality q, and $p_1(y).p_2(y) = p_1p_2(y) = p(y)$. Therefore, $p_1.p_2 \in \mathcal{I}_{\leq 2d}$, which is a vector subspace of $\mathbb{Z}_q[x_1, \ldots, x_l]_{\leq 2d}$. Here, dimensions l and d are less than the dimension of subspace n. After the multiplication, let us assume that the dimension of this subspace is n', where $n' > n$. We now choose $(n' - n)$ additional points in $\mathcal{I}_{\leq 2d}$, which span the vector space $\mathbb{Z}_q^{n'}$ to evaluate the polynomial and are given below.

$$p(y_{n+1}) = \sum_{i=1}^{n} \beta_i.p(y_i) + \sum_{i=n+2}^{n'+1} \beta_i.p(y_i) \qquad (5)$$

There exists γ_i^j for $n+2 \leq i \leq n'+1$ and $1 \leq j \leq n$, such that for all $p \in \mathcal{I}_{\leq d}$. Here, $\lambda_{s,j}$ are the coefficients of each term, and γ_i^j are the constants. Thus, equation 5 can be represented as equation 6.

$$p(y_i) = \sum_{j=1}^{n} \gamma_i^j \lambda_{s,j}.p(y_j) \text{ for } n+2 \leq i \leq n'+1 \qquad (6)$$

From the steps mentioned above, we can observe that a linear transformation from $\mathcal{I} \leq 2d$ to $\mathcal{I} \leq d$ can be obtained, which is depicted as a matrix L in $\mathbb{Z}_q^{(n+1)\times(n'+1)}$ below.

$$L = \begin{bmatrix} L_1^{n\times n} & 0^{n\times 1} & L_2^{1\times(n'-n)} \\ L_3^{1\times n} & 1 & L_4^{1\times(n'-n)} \end{bmatrix} \in \mathbb{Z}_q^{(n+1)\times(n'+1)}$$

To formulate $p(y)$ of dimension n', an integer is added to the $(n+1)^{th}$ entry of $p(y)$, equivalent to $p_1p_2(y_{n+1}) \ mod \ q$, and is obtained using a transformation matrix, B, as shown below.

$$B = \begin{bmatrix} I_n & 0 & 0 \\ \beta_1 \ldots \beta_n & 1 & \beta_{n+2} \ldots \beta_{n'+1} \\ 0 & 0 & I_{(n'-n)} \end{bmatrix}$$

The vector $p(y)$ is multiplied by L, which generates $p' = \wedge_s LB.p(y)$. Further, modular multiplication of p' is obtained using $p_{mul} = \lfloor p' \rceil \ mod \ q \in \mathbb{Z}_q^{n+1}$. Here, $\wedge_s$ is a diagonal coefficient matrix.

In order to validate the correctness of the method mentioned above, a tensor $\mathcal{M}$ matrix-based scheme is formulated. Let us consider a bilinear map $\mathcal{B}_\mathcal{M} : \mathbb{Q}^{(n+1)} \times \mathbb{Q}^{(n+1)} \rightarrow \mathbb{Q}^{(n+1)}$, which can be represented by a tensor $\mathcal{M} \in \mathbb{Q}^{(n+1)(n+1)(n+1)}$. $\mathcal{B}_\mathcal{M}(p_1, p_2) = [p_1^T M_1 p_2, \ldots\ldots, p_1^T M_{n+1} p_2]^T$, where $\mathbb{Q}$ is a rational number. $M_1, \ldots\ldots, M_{n+1}$ are the slices of tensor $\mathcal{M}$ and are depicted as $\mathcal{M} = \mathcal{N} \times_1 (I)^T \times_2 (I)^T \times_3 \wedge_s LB$. I is an identity matrix, and $\mathcal{N}(i,i,i) = (\lambda_{s,i}^{-1})^2 \ \forall i \neq n+1$, $\mathcal{N}(n+1, n+1, n+1) = \frac{2}{q}$ and $\mathcal{N}(i,j,k) = 0$ are everywhere else. Further, $\mathcal{B}_\mathcal{M}(p_1p_2) = p_1p_2 \ mod \ q$, therefore, $p_{mul} = \lfloor \mathcal{B}_\mathcal{M}(p_1.p_2) \rceil$. The vector $\mathcal{B}_\mathcal{M}(p_1p_2)$ is the sum of two vectors, v, and v', where v is a vector with integer entries equivalent to $p(y) \ mod \ q$. v' is $(0, 0, \ldots, 0, \eta)$, where η can be described as follows.

$$\eta = \frac{2}{q}(p_1\lfloor\frac{q}{2}\rfloor + e_1 + qJ_1)(p_2\lfloor\frac{q}{2}\rfloor + e_2 + qJ_2)$$
$$= p_1p_2\lfloor\frac{q}{2}\rfloor - \frac{p_1p_2}{2q} + \frac{q-1}{q}(p_1e_2 + p_2e_1) + (2e_1 - p_1)J_2$$
$$+(2e_2 - p_1)J_1 + \frac{2}{q}e_1e_2 + q(p_1J_2 + p_2J_1 + 2J_1J_2)$$

Therefore, $\lfloor \mathcal{B}_\mathcal{M}(p_1p_2) \rceil \ mod \ q = \lfloor \eta \rceil \ mod \ q$, where e_1 and e_2 are residue error terms due to performing operations in $\mathbb{Q}$ space, and the values of $|J_1|$ and $|J_2|$ are less than the norm of an identity matrix.

In our proposed work, the modulo multiplication method is implemented over a binary field. Therefore, coefficients are represented as either 0 or 1 in the proposed method, and modulo multiplication is performed by multiplying input vectors with the tensor matrix directly. Note that all the other parameters during multiplication, i.e. e_1, e_2, J_1, J_2, are considered zero due to $GF(2^m)$. An example is given below depicting steps taken by the proposed modulo multiplier.

Let us consider two multivariate polynomials, $a = (xy + 1)$ and $b = (xy^2 + x)$, and a quotienting polynomial $f = x^2y + xy + 1$. For polynomial ring $\langle f_1 \rangle$, the reducible polynomial is taken as $y^3 + 1$, and $\{1,\ y,\ (1+y),\ (y^2),\ (1+y^2),\ (y+y^2),\ (1+y+y^2)\}$ are the elements of $\langle f_1 \rangle$. The tensor elements are computed by finding a remainder, which is obtained by dividing all the monomials in Table 2. The presence of a remainder element in the tensor matrix is marked by one or else as zero. The two main operations of tensor matrix method are described below.

TABLE 2
Table to Find the Product of Monomials

.	xy^2	x^2	xy	y^2	x	y	1
xy^2	x^2y^4	x^3y^2	x^2y^3	xy^4	x^2y^2	xy^3	xy^2
x^2	x^3y^2	x^4	x^3y	x^2y^2	x^3	x^2y	x^2
xy	x^2y^3	x^3y	x^2y^2	xy^3	x^2y	xy^2	xy
y^2	xy^4	x^2y^2	xy^3	y^4	xy^2	y^3	y^2
x	x^2y^2	x^3	x^2y	xy^2	x^2	xy	x
y	xy^3	x^2y	xy^2	y^3	xy	y^2	y
1	xy^2	x^2	xy	y^2	x	y	1

Multiply: Each vector element is multiplied with other elements in all possible ways, tabulated in Table 2.

Quotienting: After the multiplication, products are divided by the quotienting polynomial to get a modular output. This operation is a critical step in the tensor formation, followed by the multiplication of the next polynomial.

The remainders can be tabulated as shown in Table 3. As mentioned earlier, each of these steps can be calculated in parallel. For generating a tensor for a particular coefficient, if it appears in the remainder of a monomial after getting the monomial divided by the quotienting polynomial, that particular coefficient in the tensor is marked as "1", else "0". The tensor matrix for xy^2 is given below as a reference, and other tensor matrices for $x^2,\ xy,\ y^2,\ x,\ y$ and 1 can be calculated similarly.

TABLE 3
Remainders for Construction of Tensor Matrices

Matrix Indices	Check for Remainder	Remainder	Tensor Coefficients
a_{00}	$x^2y^4/\langle x^2y+xy+1\rangle$	$xy+1$	0
a_{01}	$x^3y^2/\langle x^2y+xy+1\rangle$	xy^2+xy+y	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{06}	$xy^2/\langle x^2y+xy+1\rangle$	xy^2	1
a_{10}	$x^3y^2/\langle x^2y+xy+1\rangle$	xy^2+xy+y	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{66}	$1/\langle x^2y+xy+1\rangle$	1	0

$$\mathcal{T}_{xy^2}: \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$\frac{(xy+1)*(xy^2+x)}{\langle x^2y+xy+1\rangle}$ is further analyzed using the tensors mentioned above. After lexicographic ordering, $a=(xy+1)$ and $b=xy^2+x$ can be expressed as,

$$a=\begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0 & 1\end{bmatrix};\ b=\begin{bmatrix}1 & 0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}.$$

In order to find whether terms are present in the final output, a and b are multiplied with each tensor, i.e. $a\times\mathcal{T}\times b^T$, where $\mathcal{T}$ is a tensor. While finding the presence of xy^2 in the final output, $a\times\mathcal{T}_{xy^2}\times b^T$ have to be computed. If its result is "1", it indicates that the term is present in the final output; otherwise not. The output of modulo multiplication is determined as xy^2+xy+y^2+1, using the above-mentioned steps, which matches the algebraic output. A detailed description of the polynomial multiplication with an example is presented in Appendix A.

3.3 Complexity Analysis

The proposed method can be divided into three steps for the complexity analysis. The first step is ordering random multivariate polynomial equations in graded lexicographic order. The second step is to generate a tensor matrix, and the last step is to multiply it with multivariate polynomials to get the modular output. The most recent works in the polynomial modulo multiplication domain only deal with single variable polynomial equations, whereas our proposed scheme can also be applied to multivariate polynomials. The dimension of the tensor matrix is related to the number of variables and the degree of the polynomials.

For lexicographic ordering, the time complexity for a single variable polynomial with n-terms is $(n-1)$ or $O(n)$, whereas the time complexity of the polynomial having n-terms of m-variables is $(nm-1)$ or $O(mn)$. For ordering, monomials are generated using coefficients taken from reducing and quotienting polynomials for generating tensor matrices. Later, every monomial is divided by a quotienting polynomial, and a table is generated for storing the remainders of the individual monomial. Further, a $n\times n$ binary tensor matrix for each coefficient is generated individually by finding its presence in the remainders. Monomials can be generated in parallel using n^2 processing elements in a single step. Similarly, remainders for each monomial are also computed in parallel using n^2 processing elements. The time complexities of finding monomials and remainders are $O(1)$ and O($\lceil log_2 n\rceil$) [33], respectively. It is to mention that for multivariate polynomials, the time complexity to find the remainder is O($m\lceil log_2 n\rceil$). Since each tensor can be computed in parallel for all the coefficients, total time complexity for generating all the tensors is O($\lceil log_2 n\rceil$) for $m=1$, i.e. single variable polynomials. As we know, each tensor and input polynomial vectors (a and b) are represented in binary. Therefore, the step $a\times\mathcal{T}\times b^T$ employed for finding the output of the modular multiplier can be performed in $(k+1)n+2$ steps. The multiplication of row vector a with a column of $\mathcal{T}$ takes $O(1)$ using n processing elements. Later, the addition of the binary partial products is also performed in $O(k)$, where k is the depth of LUTs mimicking XOR operations. Further, the resultant row vector after performing $a\times\mathcal{T}$ is multiplied with the column vector b, and it takes $O(1+k)$ to compute the final result of the modular multiplication. Thus, the worst-case time complexity of modular multiplication is $(n-1)+\lceil log_2 n\rceil+(k+1)n+(k+1)$ or $O(n)+\lceil log_2 n\rceil+(k+1)n+(k+1)$ or $O(n)+\lceil log_2 n\rceil+(k+1)(n+1)$. In our case, for a single variable with 256 terms input, where the maximum degree of the variable is 256, modulo multiplication is performed in $O(1)$, i.e., $k=1$. This is because of performing additions of 256 bits in a single clock cycle. It is to be noted that $k\ll n$ in general, thus, $(k+1)(n+1)\approx(n+1)$. This transforms the overall time complexity of modular multiplication to $O(n)+\lceil log_2 n\rceil+2n+2$, which can also be validated using Table 8 by analyzing the delay and number of variables.

As we know, a tensor matrix is sparse and stored using sparse matrix storage formats, such as Compressed Row Storage, Harwell-Boeing etc., having space complexity $O(n_{nz})$. Here n_{nz} is the total number of "1" in a given tensor matrix. Thus, space complexity is $O(n_{nz}n)$ for all the tensors. Similarly, for storing monomials, the space complexity is $O(n^2)$ and for storing lexicographically ordered inputs, the space requirement is $O(n)$. Thus, the worst case space complexity of modular multiplication is $O(n^2+n_{nz}n+n)$, i.e., $O(n^2)$ for a single variable polynomial.

Area complexity can be determined by the total number of gates utilized in designing a particular algorithm. The most resource-intensive part of the proposed multiplier is tensor generation operation, where the remainder is evaluated for every possible monomial by dividing it using an irreducible polynomial, taking $log_2 n$ steps to compute. Every step utilizes n AND and $(n-1)$ XOR operations to produce a remainder in $log_2 n$ steps, and there are n such monomial divisions computed simultaneously. Therefore, the total space required for a tensor computation is $A_A[n^2 log_2 n]+A_X[(n(n-1))log_2 n]$, where A_A and A_X are the area required for single 2-input AND and XOR gates. Subsequently, from Figure 3 in section 4, it can be observed that the space complexity of the input vector and tensor matrix multiplication is $A_A n(r+1)+A_X(n-1)(r+1)$ for a single multiplication, where r is the number of parallel column multiplications (r-PCM). Similar to tensor generation, this step is also implemented in parallel, and for n such blocks, the total space complexity is $A_A n^2(r+1)+A_X(n^2-n)(r+1)$.

TABLE 4
Comparison of Time and Space Complexities, and Latency

Test cases	Type	Proposed	Reyhani-Hasan [4]	Zhang-Parhi [25]	Others	
Trinomial $(P(x) = x^n + x^m + 1)$ $(n > m > 1)$ (# e is constant)	Time	$(n-1)+\lceil log_2 n\rceil+(k+1)(n+1)$	$n^2 + (2 + \lceil log_2(n-1)\rceil)(n^2-1)$	$n^2+(2+\lceil log_2 n\rceil)(n^2-1)$	$(\lceil(n-1)/e\rceil + 3 + \lceil log_2 e\rceil)^{\#}$	[34]
	Space	$A_A[n^2(log_2 n + (r+1))] + A_X[(n^2-n)(log_2 n + (r+1))]$	$A_A n^2 + A_X(n^2-1)$	$A_A n^2 + A_X(n^2-1)$	$A_{NA} n^2 + A_{XN}(n^2-1) + A_R(n^2+3n+1)$	
	Latency	$T_A + T_X[(\frac{n}{r} - log_2 n + 1)]$	$T_A + (2 + \lceil log_2(n-1)\rceil)T_X$	$T_A + (1 + \lceil log_2 n\rceil)T_X)$	$(T_{NA} + T_{XN})$	
Pentanomial $(P(x) = x^n + x^{k3} + x^{k2} + x^{k1} + 1, 1 < k1 < k2 < k3 \leq \frac{n}{2})$	Time	$(n-1)+\lceil log_2 n\rceil+(k+1)(n+1)$	$n^2 + (4 + \lceil log_2(n-1)\rceil)(n^2 + 2n-3)$	$n^2+(6+\lceil log_2 n\rceil)(n^2+2n-3)$	-	
	Space	$A_A[n^2(log_2 n + (r+1))] + A_X[(n^2-n)(log_2 n + (r+1))]$	$A_A n^2+A_X(n^2+2n-3)$	$A_A n^2+A_X(n^2+2n-3)$	-	
	Latency	$T_A + T_X[(\frac{n}{r} - log_2 n + 1)]$	$T_A + (4 + \lceil log_2(n-1)\rceil))T_X$	$T_A + (6 + \lceil log_2 n\rceil))T_X$	-	
Single variable s-ESP $(P(x)$=x^{ms}+$x^{(m-1)s}$+$\cdots + x^s + 1, n = ms)$	Time	$(n-1)+\lceil log_2 n\rceil+(k+1)(n+1)$	$n^2+(1+\lceil log_2 n\rceil)(n^2-s)$	$n^2+(1+\lceil log_2 n\rceil)(n^2-s)$	-	
	Space	$A_A[n^2(log_2 n + (r+1))] + A_X[(n^2-n)(log_2 n + (r+1))]$	$A_A n^2 + A_X(n^2-s)$	$A_A n^2 + A_X(n^2-s)$	-	
	Latency	$T_A + T_X[(\frac{n}{r} - log_2 n + 1)]$	$T_A + (1 + \lceil log_2 n\rceil)T_X$	$T_A + (1 + \lceil log_2 n\rceil)T_X$	-	
Generic Polynomial $(P(x) = x^n+x^{kt}+\cdots+x^{k2}+x^{k1}+1, 1 \leq k1 < k2 < \ldots kt \leq \frac{n}{2})$	Time	$(n-1)+\lceil log_2 n\rceil+(k+1)(n+1)$	$n^2 + (\lceil log_2(t+1)\rceil + \lceil log_2(\lceil \frac{t}{2}\rceil + 1)\rceil + \lceil log_2(n-1)\rceil)(n + t)(n-1)$	$n^2+(2t+\lceil log_2 n\rceil)(n+t)(n-1)$	$1.78n^{log_2 3} + (4.91n^{log_2 3} - 11n + 8.88)(2log_2 n - 1)$	[35]
	Space	$A_A[n^2(log_2 n + (r+1))] + A_X[(n^2-n)(log_2 n + (r+1))]$	$A_A n^2 + A_X(n+t)(n-1)$	$A_A n^2 + A_X(n+t)(n-1)$	$A_A(1.78n^{log_2 3}) + A_X(4.91n^{log_2 3} - 11n + 8.88)$	
	Latency	$T_A + T_X[(\frac{n}{r} - log_2 n + 1)]$	$T_A + (\lceil log_2(t+1)\rceil + \lceil log_2(\lceil \frac{t}{2}\rceil + 1)\rceil + \lceil log_2(n-1)\rceil)T_X$	$T_A + (2t + \lceil log_2 n\rceil)T_X$	$T_A + T_X(2log_2 n - 1)$	
Multivariate polynomial $P(x_1, x_2 \ldots x_m)$	Time	$(mn-1)+m\lceil log_2 n\rceil + (k+1)(n+1)$	-	-	-	
	Space	$A_A[n^2 m(log_2 nm + (r+1))]+A_X[(nm(n-1)(log_2 nm + (r+1))]$	-	-	-	
	Latency	$T_A + T_X[(\frac{nm}{r} - log_2 nm + 1)]$	-	-	-	

Therefore, the overall space complexity of our proposed modulo multiplication method for a single variable polynomial is $A_A[n^2(log_2 n+(r+1))]+A_X[(n^2-n)(log_2 n+(r+1))]$ and for multiple variable polynomials is $A_A[n^2 m(log_2 nm+(r+1))] + A_X[(nm(n-1)(log_2 nm + (r+1))]$, where m is the number of variables. Time delay in terms of T_A (2-input AND gate delay), and T_X (2-input XOR gate delay) can be calculated after aggregating the total time taken by all the AND and XOR gate delays. Here, generation of individual tensor matrices and vector-matrix multiplication of input polynomials with tensors are mutually exclusive processes. The time delay of a single and multiple variable polynomials are $T_A+T_X[(\frac{n}{r}-log_2 n+1)]$, and $T_A+T_X[(\frac{nm}{r}-log_2 nm+1)]$. Note that n is the maximum degree of the input polynomial.

The comparison of time and space complexities and latency of the proposed method with [4], [25], [34], and [35] is presented in Table 4. A_{NA}, A_{XN}, T_{NA}, and T_{XN} in [34] are area and gate delays of 2-input NAND and XNOR gates. The comparison of the methods reported in [4], [25], [34], and [35] for specific values of n are presented in Table 5. It can be seen that for $\{(r, n)\} = \{(16, 128); (32, 256); (32, 512)\}$ latencies are $(T_A + 2T_X)$, $(T_A + T_X)$ and $(T_A + 8T_X)$. These are the optimal latencies of the proposed multiplier for $n = 128,\ 256$ and 512, and are the least among all the other multipliers. It is to mention that the practical limit of latency cannot be reduced further even if more PCMs are employed. This would only increase dynamic power consumption without further reduction of the latency. Note that our proposed method does not depend on the polynomial expression of a single variable and can be easily extended to multivariate polynomials. It can be observed from Table 4 that the time complexity and latency are linear, whereas the existing methods are quadratic or subquadratic. Further, the space complexity

TABLE 5
Complexity Comparison for Specific Values of n

Trinomial ($x^n + x^m + 1$, where $n > m > 1$); r-PCM is used									
Method	$n = 128; r = 16;$			$n = 256; r = 32;$			$n = 512; r = 32;$		
	#AND	#XOR	Delay	#AND	#XOR	Delay	#AND	#XOR	Delay
[35]	3888	9145	$(T_A + 14T_X)$	11664	29360	$(T_A + 16T_X)$	34992	90124	$(T_A + 18T_X)$
[4]	16384	16383	$(T_A + 9T_X)$	65536	65535	$(T_A + 10T_X)$	262144	262143	$(T_A + 11T_X)$
[25]	16384	16383	$(T_A + 8T_X)$	65536	65535	$(T_A + 9T_X)$	262144	262143	$(T_A + 10T_X)$
Proposed	131072	130048	$T_A + (\frac{128}{r} - 6)T_X = (T_A + 2T_X)$	589824	587520	$T_A + (\frac{256}{r} - 7)T_X = (T_A + T_X)$	2621440	2616320	$T_A + (\frac{512}{r} - 8)T_X = (T_A + 8T_X)$

of many existing single variable polynomial multipliers is $O(n^{1+\varepsilon})$ [34] [35], whereas the space complexity of the proposed multiplier is $O(n^2 logn)$. Here, $0.5 < \varepsilon < 1$. This is due to not considering the sparsity of the tensors, which results in the quadratic space complexity. Further, the implementation details of the proposed multiplier are depicted in the following section.

4 Implementation Details

The proposed multiplier is implemented using MATLAB (software) and on the FPGA platform (hardware). MATLAB-based implementation is described in detail below.

4.1 MATLAB Implementation

A multivariate polynomial is represented as, $\begin{bmatrix} E_0 & E_1 & E_2 & \dots & E_n \end{bmatrix}$, where E_0 represents a constant term and E_i holds the information about the maximum degree of the i^{th} variable for $i = 1 \cdots n$. In the example discussed in Appendix A, E_0, E_1 and E_2 are equal to 1, 2 and 2, respectively. Thus, the program takes an input in the vector form $[E_0, E_1, E_2, \dots E_n]$. The lexicographic ordering of the polynomials follows the steps mentioned above. Subsequently, multiplication and quotienting operations are performed. The following modules are designed to meet the desired goal. Their brief descriptions and algorithmic descriptions are given below.

• **Monomial Ordering Module**: In the proposed work, the graded lexicographic ordering method is applied, where degree of a polynomial is given more weightage than the individual order of a variable. Algorithm 1 exhibits the steps of the module implementation. We have modified the standard lexicographic ordering algorithm [30], [31] and have employed it to generate ordered input polynomial vectors for our proposed tensor multiplication. Let us assume that the number of polynomial variables is m and the maximum degree of each variable is n. This results in the total number of possible coefficients $l = (n+1)^m$. $A(X_1, X_2 \dots, X_m)$ and $B(X_1, X_2 \dots, X_m)$ denote unsorted and sorted polynomials of degree n, respectively. The algorithm, which performs lexicographic ordering of $A(X_1, X_2 \dots, X_m)$, is described in Algorithm 1. The output of this algorithm is a binary array $S = \{s_1, s_2, \dots, s_l\}$, which is employed in all the subsequent operations in our proposed modulo multiplier. Here, in the proposed method, lexicographically ordered output exhibits the presence of any particular coefficient in the input polynomial vectors.

Correctness of lexicographic ordering: The final ordered polynomial, $B(X_1, X_2, \dots, X_m) = \{(X_1^n X_2^n \dots X_m^n), (X_1^n X_2^n \dots X_m^{n-1}), \dots, (X_1^n X_2^n \dots X_m^0), \dots, (X_1^0 X_2^0 \dots X_m^1), (X_1^0 X_2^0 \dots X_m^0)\}$, can be obtained using our lexicographic ordering algorithm on an unsorted m-variable and n-degree generic polynomial. In Algorithm 1, it can be observed that the initialization of $d[m]$ can be performed in $O(m)$. Steps mentioned in Lines $8 - 11$ extract the degree of individual variable in A_k and store it in $d[m]$. This can also be conducted in $O(m)$. Steps stated in Lines $12 - 13$ determine the location of A_k in $B(X_1, X_2, \dots, X_m)$ in $O(1)$. Also, $d[m]$ re-initialization is performed in $O(m)$. The whole process, i.e., Lines $6 - 15$, repeats p times, taking $O(pm)$ steps. Thus, the total time complexity of Algorithm 1 is $O(pm) + O(m)$ or $O(pm)$. For single variable polynomials, $m = 1$ enables Algorithm 1 to perform in $O(p)$ or linear time. In reality, a significantly less number of coefficients are present in the polynomial, i.e., $p \ll l$, which reduces computation time drastically. It is to be noted that $B(X_1, X_2, \dots, X_m)$ is an ordered polynomial. A one-to-one map exists between B and S for all of its ordered terms. It can be proved inductively that for p terms, if a linear mapping exists between B and S, $B_{p+1} \uplus B$, S_{p+1} is uniquely determined, assuming that no two terms in A or B are the same, and each term $\in B$ has a unique image in S.

Algorithm 1: Graded Lexicographic Ordering

```
1. Input: $A(X_1, X_2, \dots, X_m)$, $m$, $n$
2. Output: S=$\{s_1, s_2, \dots, s_l\}$, where $l = (n+1)^m$
3. Initialize: $p$=number of coefficients in
   $A(X_1, X_2, \dots, X_m)$; S= $\{0, \dots, 0\}$; $temp = 0$
4. $A_k = \prod_{j=1}^{m}(X_j^i)$ and $0 \le i \le n$, $1 \le k \le p$
5. Initialize $d[m]$: An integer array of size $m$ to hold
   degree of each variable in $A_k$;
6. for $k = 1$ to $p$
7. begin
8.     for $j = 1$ to $m$
9.     begin
10.        $d_j = \ degree\ of\ X_j \in A_k$;
11.    end
12.        $temp = \sum_{i=1}^{m}\{(n+1)^{m-i}.(n - d_i)\} + 1$;
13.        $s_{temp} = 1$
14.        Re-initialize $d[m]$
15. end
16. return S
```

For instance, we take two variables (x and y) and the maximum degree of a variable in the polynomial as three, i.e., $m = 2$, $n = 3$ and $l = (n+1)^m = (3+1)^2 = 16$. Here, all the possible coefficients ordered lexicographically can be expressed as B=$\{x^3y^3$, x^3y^2, x^3y, x^3, x^2y^3, x^2y^2, x^2y, x^2, xy^3, xy^2, xy, x, y^3, y^2, y, $1\}$. These coefficients are uniquely mapped from 1 to 16 in S in an ordered manner. Now consider an input polynomial $(x^2y^3 + y^2 + x^3 + 1)$, where $p = 4$. Thus, in the output vector, there should be four "1s" only. Using Algorithm 1, four array indices can be computed using the expression $temp = ((n+1)^{m-1}.(n - d_1) + (n+1)^{m-2}.(n - d_2) + 1)$, where $\{d_1, d_2\}$ for $p = 1 \rightarrow 4$ are ($\{2, 3\}$, $\{0, 2\}$,$\{3, 0\}$, and $\{0, 0\}$), respectively. The array indices, thus calculated, are $\{4, 5, 14, \text{and } 16\}$ for $x^3 + x^2y^3 + y^2 + 1$, which constitute the final binary output in graded lexicographic format as $\{0001100000000101\}$.

• **Monomial Multiplication Module**: This module multiplies two monomials to generate their product, as shown in Table 2. The details of this module have already been explained in section 3.

• **Quotienting Module**: We employ a well-known Euclid's polynomial division algorithm to obtain remainders to generate tensor matrices. Algorithm 2 depicts the steps taken to perform the quotienting operation. Since Euclid's polynomial division method is widely known, its detailed mathematical description and hardware implementation can be found in [36], [37], [38]. In Algorithm 2, $D(x)$, $R(x)$, and $Divi(x)$ denote quotienting, reducible polynomial and

Algorithm 2: Polynomial Quotienting Operation

```
1. Input: $D(x)$, $R(x)$, $Divi(x)$
2. Output: Remainder $Rem$ = $Divi(x)/D(x)$ mod $R(x)$
3. Initialize: $A = D(x), B = R(x), C = Divi(x), E = 0, Rem = 0$;
4. while ($B \neq 0$) do
5.     while($a_0 == 0$) or ($b_0 == 0$) do
6.         if($a_0 == 0$) then
7.             $A = A/x, C = (C + c_0.R(x))/x$;
8.         end if
9.         if $b_0 == 0$ then
10.            $B = B/x, E = (E + e_0.R(x))/x$;
11.        end if
12.    end while
13.    if $A > B$ then
14.        $A = B + A, C = C + E$;
15.    else
16.        $B = B + A, E = C + E$;
17.    end if
18. end while
19. $Rem = C - A.E$;
20. Return Rem
```

input polynomial to be quotient, respectively. Here, Rem is the final remainder of the quotienting operation.

The ordering of the *input* vectors is determined using the steps of Algorithm 1. Thereafter, the multiplier module is called recursively, followed by the quotienting module (Algorithm 2), until all the input polynomials are evaluated,and tensor matrices are formulated.

4.2 FPGA Implementation

Modulo multiplication of multivariate polynomials is designed using Verilog HDL. All the modules are synthesized and analyzed using the Xilinx Vivado platform [39] and are implemented on Zynq-7000-Z020 and Artix-7-200T Xilinx FPGA boards. A detailed description of these modules is given below.

• **Top module**: This module receives two multivariate polynomials as input, and is the central control unit of the proposed multiplier. The top module calls the following modules in the hierarchy: *multipoly*, *divpoly*, *tensorPoly* and *mulVectMat*, which are described below. The finite state machine (FSM) of the multivariate polynomial modulo multiplication method is presented in Figure 4.

• **MultiPoly**: This module takes two monomials simultaneously as inputs and generates their product. Iteratively, it produces product of every combination of the monomials.

• **Divpoly**: This module receives the product obtained from the MultiPoly module as input and divides it first by the quotienting polynomial. If the product is not divisible by the quotienting polynomial, either reducible or irreducible polynomials are chosen to divide the product. In each iteration, this module generates a division of every product obtained from the *MultiPoly* module.

• **TensorPoly**: This module receives an output of the *DivPoly* module and generates tensor matrices.

• **MulVectMat**: Finally, modulo multiplication of polynomials employing tensors is performed by this module.

Vector-Tensor-Vector Multiplication

The architecture of the multiplication unit of the input polynomial vector and a tensor matrix is presented in Figure 3. Initially, it is needed to generate tensor matrices $T = \{T_1, T_2, \ldots, T_x, \ldots, T_n\}$ specific to an irreducible polynomial, which is generated once, if this polynomial is unchanged while performing different polynomial multiplications. The first polynomial $A = \{A_1, A_2, \ldots, A_n\}$ is multiplied with a tensor matrix T_x of dimension $(n \times n)$ column-wise and then multiplied with a second polynomial $B^\top = \{B_1, B_2, \ldots, B_n\}^\top$ to determine the presence of the corresponding coefficient in the output. Here, each column of T_x is multiplied with all the coefficients of polynomial A, which takes two clock cycles to compute the result. This implementation is noted as one parallel column multiplication (1-PCM) and can be performed up to r-PCM, where $r \leq n$. As all the coefficients of A, B, and a tensor matrix T_x are computed and stored in BRAM, all the operations related to PCM can be performed simultaneously. Therefore, by increasing r, vector-tensor-vector computation time can be reduced drastically with a minimal increase in resource utilization. As an example, in Figure 3, 2-PCM is marked in gray.

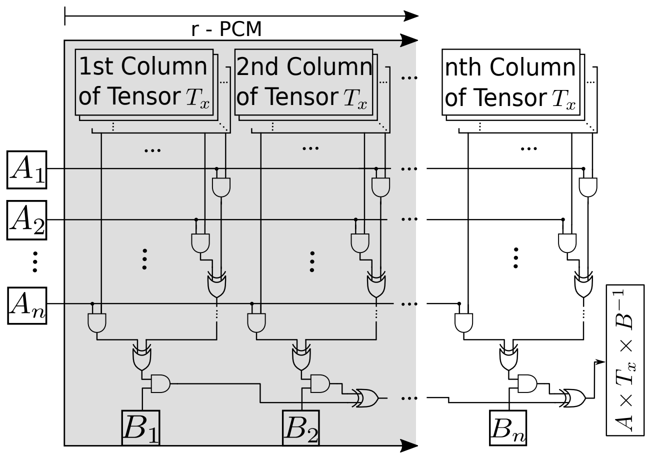
r - PCM
1st Column of Tensor T_x
2nd Column of Tensor T_x
nth Column of Tensor T_x
A_1
A_2
A_n
B_1
B_2
B_n
$A \times T_x \times B^{-1}$

Fig. 3. r-Parallel Column Multiplication (r-PCM) Architecture

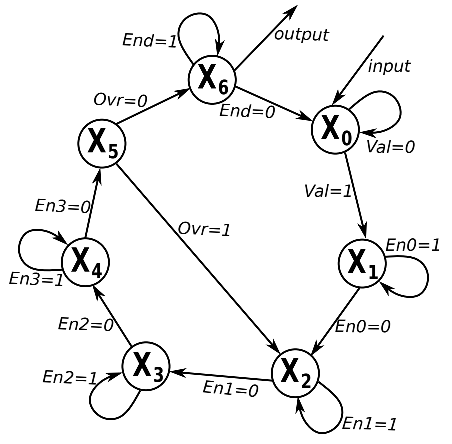

Fig. 4. Finite State Machine of Main Module

Tables 6 and 7 present descriptions of all the probable states and transitions to a 7-bit flag register F={Val, En0,

TABLE 6
State Description of the FSM and Dependency on Flag Registers

State	Specific Description	Flag
X_0	Check for valid data	Val, End
X_1	Compute monomial multiplication	Val, En0
X_2	Quotienting operation to generate remainders	En0, En1, Ovr
X_3	Formulate tensors from X_3	En1, En2
X_4	Vector tensor matrix	En2, En3
X_5	Check for further modulo division	En3, Ovr
X_6	Produce multiplication output	Ovr, End

TABLE 7
State Transition of the FSM of Proposed Modulo Multiplier

	Flag register		Present State						
			X_0	X_1	X_2	X_3	X_4	X_5	X_6
Next State	Val	0	X_0	x	x	x	x	x	x
		1	X_1	x	x	x	x	x	x
	En0	0	x	X_2	x	x	x	x	x
		1	x	X_1	x	x	x	x	x
	En1	0	x	x	X_3	x	x	x	x
		1	x	x	X_2	x	x	x	x
	En2	0	x	x	x	X_4	x	x	x
		1	x	x	x	X_3	x	x	x
	En3	0	x	x	x	x	X_5	x	x
		1	x	x	x	x	X_4	x	x
	Ovr	0	x	x	x	x	x	X_6	x
		1	x	x	x	x	x	X_2	x
	End	0	x	x	x	x	x	x	X_0
		1	x	x	x	x	x	x	X_6

"x" is Don't Care condition

En1, En2, En3, Ovr, End}. It can be observed that there are no ambiguous state transitions, and for all the possible values of the flag register, there is a unique transition from the present state to the next state. All the inputs to the main multiplication module are assumed to be lexicographically ordered; therefore, ordering is not included in the main module of the FSM. The X_0 remains in the same state until it gets lexicographically ordered inputs. As per the state diagram of the top module described in Figure 4, state X_0 receives input, checks for valid data, and forwards it to the next state. If data is not valid, the flag Val is set to "0", and it waits for a valid input until the valid input Val flag is set and enters into state X_1. State X_1 computes the multiplication of elements and stores it in a vector if $En0$ is set. After completion of the multiplication, the $En0$ flag resets to zero, and state X_2 is activated. Based on flag $En1$, the division operation is performed in state X_2. Thereafter, if $En2$ is set, state X_3 formulates tensors and advances the FSM into the next state. Subsequently, when $En3$ is set, state X_4 multiplies tensor with input polynomials. State X_5 checks whether modulo division is needed to constrain the output in the desired range or not. If affirmative, it sets flag Ovr to "1", advancing the FSM to state X_2 or moving to state X_6. The flag End is set to "1", which implies X_6 is ready to generate and produce an output. If an external circuit is ready to take the output, the flag End is reset from "1" to "0" and the FSM re-enters into an initial state X_0. Additionally, an overview of the hardware implementation is shown in Figure 5.

There are two ways to prove the correctness of a finite state machine (FSM), as mentioned in [40]. In the first method, an FSM can be split at two abstraction levels, i.e., high and low. The high-level abstraction specifies the abstract behavior of an FSM, while the low level illustrates a detailed hardware implementation of the FSM. Therefore, an FSM's correctness can be addressed by employing behavioral equivalence between these two processes, i.e., the abstract specifications and hardware implementation. The second method is based on algebraic computational methodology, in which FSM algebraic techniques (based on semantic transitions or syntactic transformations) can be analyzed for their correctness. The intermediate results obtained through hardware implementation match Tables 2 and 3 for the same polynomial inputs. It is to be noted that the FPGA implementation outcome is ratified step by step with the theoretical analysis of our proposed modulo multiplier. This fulfills the equivalence criterion of the correctness of the FSM, as stated in [40]. In the next section, experimental details of the proposed multiplier are presented.

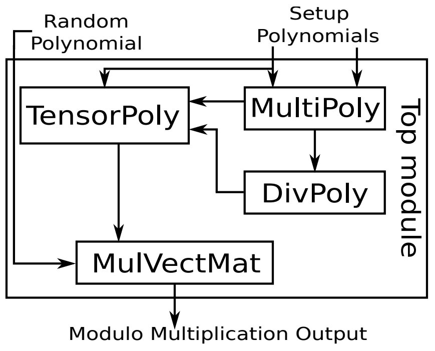

Fig. 5. Block-level view basic process flow of the FSM

5 Results and Discussion

In this section, the results of FPGA implementation are presented. The proposed modulo multiplier is realized on Zynq-7000-Z020 and Artix-7-200T Xilinx FPGA boards. Each slice of the FPGA contains two types of LUTs; 6-input and 2-output, and 4-input and 2-output LUTs. Every slice contains four such types of LUTs. The Artix and Zynq architecture has eight flip-flops, one MUX, and one look-a-head carry logic cell per slice. Since most of the logic is implemented using pass-transistor logic, the total estimated transistor count is around 222 per slice. The proposed multiplier implemented on FPGA is validated with the vast range of inputs. The number of variables in inputs varies from two to nine, and the maximum degree for each variable ranges from 16 to 128. Note that the experiments are performed over field sizes ranging from 32 to 1152. The resource, power, area, and delay information is presented in Table 8. A comparison of the proposed multiplier with the recent multipliers is depicted in Table 9 and Table 10. From the basic structural diagram for multivariate polynomial multiplication shown in Figure 2, the total functional blocks required for the operations can be calculated using equation 7. The main polynomial multiplier block, a vector-matrix multiplier block, and vector addition block are denoted by $Mult_P$, $Mult_V$ and Add_V, respectively. If the total number of variables is V and the maximum degree of a variable is D, then the arithmetic functional block utilization can be formulated as equation 7.

$$Mult_P \times D + Mult_V \times D \times V + Add_V \times (D-1) \quad (7)$$

TABLE 8
Resource Utilization For Various Inputs

Variables	Degree	Slice Count	Transistor Count	Area (μm^2)	Delay (ns)	Power (mW)	PDP ($10^{-12}J$)	PDAP	ADP($\mu m^2.ns$)
2	16	781	173382	3120.9	1.84	205.96	378.97	1182691	5742.42
	32	1565	347430	6253.8	2.736	291.25	796.84	4983197	17110.24
	64	3133	695526	12519.5	4.528	461.81	2091.07	26179027	56688.16
	128	6269	1391718	25051.0	8.112	802.95	6513.47	163168293	203213.10
3	16	793	176046	3168.9	1.94	207.27	402.09	1274152	6922.76
	32	1789	397158	7148.9	2.836	315.61	895.07	6398664	20274.13
	64	3581	794982	14309.7	4.628	510.55	2362.79	33810682	66225.19
	128	7165	1590630	28631.4	8.212	900.41	7394.16	211704601	235120.57
4	16	1005	223110	4016.0	2.04	230.33	469.87	1886952	8192.60
	32	2013	446886	8044.0	2.936	339.98	998.17	8029180	23617.04
	64	4029	894438	16099.9	4.728	559.28	2644.26	42572125	76120.26
	128	8061	1789542	32211.8	8.312	997.88	8294.35	267175315	267744.12
5	16	1117	247974	4463.6	2.14	242.51	518.97	2316420	9551.96
	32	2237	496614	8939.1	3.036	364.35	1106.14	9887833	27138.97
	64	4477	993894	17890.1	4.828	608.01	2935.47	52515702	86373.37
	128	8957	1988454	35792.2	8.412	1095.35	9214.03	329789817	301083.76
6	16	1229	272838	4911.1	2.24	254.70	570.51	2801808	11000.83
	32	2461	546342	9834.2	3.136	388.71	1218.99	11987709	30839.92
	64	4925	1093350	19680.3	4.928	656.75	3236.43	63693759	96984.52
	128	9853	2187366	39372.6	8.512	1192.81	10153.20	399757489	335139.47
7	16	1341	297702	5358.7	2.34	266.88	624.49	3346389	12539.21
	32	2685	596070	10729.3	3.236	413.08	1336.71	14341893	34719.89
	64	5373	1192806	21470.6	5.028	705.48	3547.13	76158641	107953.72
	128	10749	2386278	42953.1	8.612	1290.28	11111.86	477287716	369911.28
8	16	1453	322566	5806.2	2.44	279.06	680.90	3953433	14167.10
	32	2909	645798	11624.4	3.336	437.45	1459.31	16963473	38778.88
	64	5821	1292262	23260.8	5.128	754.21	3867.59	89962694	119280.96
	128	11645	2585190	46533.5	8.712	1387.75	12090.02	562589881	405399.16
9	16	1565	347430	6253.8	2.54	291.25	739.76	4626213	15884.50
	32	3133	695526	12519.5	3.436	461.81	1586.78	19865534	43016.90
	64	6269	1391718	25051.0	5.228	802.95	4197.78	105158264	130966.24
	128	12541	2784102	50113.9	8.812	1485.21	13087.68	655873366	441603.13

Maximum cumulative degree is $(2 \times 128) = 256$, $(3 \times 128) = 384$, $(4 \times 128) = 512$, $(5 \times 128) = 640$, $(6 \times 128) = 768$, $(7 \times 128) = 896$, $(8 \times 128) = 1024$, $(9 \times 128) = 1152$, PDP = power-delay-product, PDAP = power-delay-area-product, ADP = area-delay-product.

Table 8 presents a detailed physical aspect of the proposed multiplier. In this table, overall transistor count, area, delay, power, *power-delay product* (PDP), *power-delay-area product* (PDAP), and *area-delay product* (ADP) are stated. Transistor count can easily be estimated using the total slice count and other resource utilization indicators. For the total area estimation of the proposed multiplier, first, we consider the area of a single 22-nm transistor based on the Intel Standard library [41], and then multiply it by the total number of transistors. This provides a close estimation of actual ASIC implementation. Power and delay are estimated by employing synthesis reports of the proposed multiplier. It is to mention that delays shown in Table 8 are the cumulative delay of the logic path and net delays only. Due to hardware resource constraints, the HDL analyzer may generate different delays for different underlying hardware architectures because of the distinctive performance of its synthesis, placement, and routing algorithms.

For an FPGA implementation, block-level logic path delays can be estimated using $T_{MP} = 0.9ns$, $T_{MS} = 0.1ns$ and $T_{AS} = 0.056ns$, obtained using a synthesis report of FPGA realization. These can also be used to compute generic delays. The total delay can be formulated from Figure 2 as mentioned in equation 8, where T_{MP}, T_{MS} and T_{AS} are the total delay of the individual polynomial multiplier, delay of individual vector-matrix multiplier, and delay of vector addition unit, respectively. It can be observed that FPGA-based implementation matches the analytical observations.

$$T_{MP} + (V-1) \times T_{MS} + [log_2^D] \times T_{AS} \qquad (8)$$

In Table 9, a comparison with recent multipliers is presented. It is found that our proposed multiplier is faster as well as power efficient and utilizes lesser resources as compared to other multipliers. The experimental results presented in [9] are for a 4-bit power array of two variables. They have implemented single variable NIST polynomials to showcase the performance of their proposed multiplier. In [16], the vector-matrix multiplier is explicitly implemented for DSP and communication applications, whereas in [17], the Toeplitz matrix-vector product-based method is proposed. In [17], a single digit is represented by 2-bits to perform polynomial multiplication in every clock cycle. In [18], flexible architecture for the large binary polynomial multiplier is demonstrated by combining iterative Karatsuba and Comba algorithms. Lastly, in [13], number theoretic transform (NTT) and inverse-NTT (INTT) based multiplication techniques are presented, which exploit the concept of the n^{th} root of unity to find multiplication and are only optimized for the NIST finalist CRYSTALS-KYBER PQC scheme. It can be observed that the proposed multiplier's experimental results are better than the methods mentioned earlier. The multiplier proposed in this paper can be efficiently utilized to design state-of-the-art Learning-with-Errors (LWE) and Ring-LWE (RWE) cryptographic homomorphic encryption algorithms.

Figure 3 shows the depiction of vector-tensor-vector multiplier architecture in the proposed method. This architecture enables the multiplication of r columns of T_x in parallel, where $r \leq n$, and can be performed in $((2 \times \frac{n}{r}) + 2)$ steps. The cumulative computational complexity employing

TABLE 9
Comparison of Existing Multipliers with the Proposed Multiplier

Work		Platform	Resource LUT/REG/MUX/DSP/BRAM	Comparative Utilization	Clock (MHz)	Power (W)	Delay (nS)	Latency (μs/CC)
Proposed*	1-PCM	Zynq-XC7Z020	8993/6479/6/-/6	**1.345X**	100	0.205	8.734	5.22/522
		Artix-7-200T	8982/2497/7/-/25	**X**	197	0.492	4.52	2.65/522
	4-PCM	Zynq-XC7Z020	9099/6544/6/-/6	**1.360X**	100	0.216	8.734	1.38/138
		Artix-7-200T	9088/2562/7/-/25	**1.015X**	194	0.511	4.61	0.72/138
	16-PCM	Zynq-XC7Z020	9659/6496/6/-/6	**1.405X**	98	0.227	8.82	0.43/42
		Artix-7-200T	9648/2514/7/-/25	**1.059X**	192	0.518	4.65	0.22/42
Toeplitz [17]**		Virtex-5	14701/∼/∼/∼/∼	**1.277X**	301.41	∼	23.065	13.791/∼
Alshawi et al.-I [16]*		Virtex-5	16207/12760/∼/∼/∼	**2.516X**	81.71	∼	12.24	∼/∼
Alshawi et al.-II [16]*		Virtex-5	8594/4700/∼/∼/∼	**1.155X**	65.47	∼	15.27	∼/∼
Karatsuba [9]**		Stratix II	∼/∼/∼/∼/∼	∼	86.80	3.216	41.52	∼/∼
LBPM [18]*		Artix -7	10768/21536/-/-/183	**2.822X**	143	∼	4000	27/∼
Yaman et al. [13]***	16 BTFs	Spartan-6	9898/3688/-/16/70	**1.188X**	115	∼	∼	∼/256
		Artix-7-200T	9508/2684/-/16/35	**1.064X**	172	∼	∼	

Polynomial coefficients in the form of * $x^{256} + \cdots + 1$ and, ** denotes pentanomial and trinomial polynomials $x^{163} + x^7 + x^6 + x^3 + 1$ and $x^{233} + x^{74} + 1$, ***$x^{256} + 1$, respectively. "∼" and "−" denote "Data not specified" and "Resource not utilized", respectively.

tensor generation and r-PCM is $log_2 n + (2 \times \frac{n}{r}) + 2$. Along with a sequential implementation, two parallel implementations, 4-PCM and 16-PCM, are realized to compare the performance of the proposed modulo multiplier with state-of-the-art methods available in the literature. This comparison is presented in Table 9 for reference. It can be observed that the overall latency of the multiplier proposed in [13] are 3359, 864, and 256 clock cycles using 1-BTFs, 4-BTFs, and 16-BTFs, respectively. The multiplier proposed here takes 522, 138, and 42 clock cycles only while employing 1-PCM, 4-PCM and 16-PCM, respectively. BTF is the butterfly architecture proposed for polynomial multiplication in [13]. The proposed implementations are 6.43×, 6.26× and 6.09× faster than the hardware realizations proposed in [13] for 1-BTFs, 4-BTFs and 16-BTFs, respectively.

For an irreducible input polynomial mentioned in Table 9, tensor matrix generation consumes an average {8649-LUT, 6367-REG, 6-DSP, 6-BRAM} and {8638-LUT, 2412-REG, 7-DSP, 25-BRAM} resources in Zynq-XC7Z020 and Artix-7-200T FPGAs, respectively, which is approximately 94% of the total resource utilization. Further, the vector-tensor-vector multiplier consumes fewer resources than the tensor generation module. Its parallel implementation, i.e. r-PCM blocks, slightly increases overall resource utilization but reduces computation steps by a factor of r. It can be observed that the resource utilization increases by $0.059\times$ for r-PCM in our proposed multiplier, whereas resource utilization escalates by $9.39\times$ for r-BTF in [13] when r changes from 1 to 16. It can be stated that the performance of our proposed multiplier increases with an increase in r at the minimal expense of resources. It is to mention that the work presented in [13] and our proposed work report latencies (computational steps) excluding the steps taken in loading data into BRAM. Note that each generated tensor is sparse when the multiplier is implemented practically on the hardware platform. Its sparsity is exploited by the EDA tools during design synthesis. Thus, the complete design becomes not only resource optimized but comparable to other multipliers as well. For a single variable, the space complexity of the proposed multiplier for a particular polynomial realization on the hardware is comparable to the existing methods and is showcased in Table 9.

As shown in Table 9, total LUT logic slices, flip-flops, MUX, and BRAM utilized during implementing our proposed multiplier on the Zynq-7000-Z020 board are 8993, 6479, 6, and 6, respectively, at 100 MHz. Similarly, total LUT logic slices, total flip-flops, MUX, and BRAM utilized realizing the same design on Artix-7-200T FPGA board are 8982, 2497, 7, and 25, respectively, at 197 MHz. Comparative resource utilization of all the multipliers is presented in Table 9. It can be observed that the resource utilization of our proposed multiplier is optimal, and its average critical path is also reduced, resulting in higher operating frequency. Its total power consumption is estimated to be approximately $0.2W$ and $0.492W$ at 100 MHz and 197 MHz, respectively, which are a minimum 6.5× less compared to the existing multipliers. For further power reduction in ASIC, input clock frequency and supply voltage can be reduced.

The static power consumption is higher in our proposed implementation because of storing initial polynomial coefficients and tensors. It is also found that on the Artix-7 series

TABLE 10
Comparison of Delay and Area and Cycle Requirement of Proposed Multiplier With Existing Multipliers

Multiplier	m	Delay (ns)	Area (μm^2)	Cycle (no.)
Proposed	163	3.046	9029	334
	233	3.266	10998	474
	283	3.426	12430	574
	409	3.816	15921	826
	571	4.326	20486	1150
	1223	6.366	38747	2454
Chen at el. [7]	163	439	13544	4458
	233	4032	19448	6108
	283	4576	19692	7596
	409	7392	20670	12908
	571	21216	127450	22100
	1223	50656	270214	91762
Hua at el. [8], [15]	163	90	4481	1203
	233	161	4715	2337
	283	204	4883	3331
	409	425	5442	6923
	571	816	6186	13163
	1223	3832	9195	60181
Chiou at el. [19]	163	5	61637	32
	233	7	78893	45
	283	7	100112	53
	409	11	133779	77
	571	14	202323	105
	1223	34	262175	227

Polynomial expression applied is $x^m + x^{m-1} \cdots + 1$

FPGA board with XC7A200T, the latencies of our multiplier and the multiplier based on [13] are $2.65\mu s$ and $2.31\mu s$ at the clock frequencies of 197 MHz and 172MHz. It can be seen in Table 9 the multiplier proposed in [13] employs DSP blocks, while our proposed design does not use it. Also, the multiplier proposed in [13] utilizes the FFT type of operations, which can be performed easily using DSP blocks [42]. This helps in reducing the overall latency of the design. Thus, there is a marginal difference in the latencies of both the multipliers. It can also be observed in Table 9 that the power consumption of the proposed multiplier is the least among all the other multipliers, even in the worst-case scenario. Thus, our proposed multiplier can be considered area and power optimal with respect to other multipliers and can be utilized in a wide range of applications, including lattice-based PQC algorithms, drone-based surveillance systems, etc.

Table 10 compares three more well-known multiplication schemes over parameters, such area delay and latency in clock cycles with varying polynomial degrees. A multiplier based on the dual systolic basis is depicted in [7]. In [8] and [15], low complexity systolic multipliers based on the Hankel matrix and Karatsuba algorithm are presented, respectively. In [19], the multiplication unit is decomposed into four mutually independent sub-multiplication units, which improves the performance compared to the prior methods. In Table 10, a comparison of specific polynomial multipliers having polynomial degrees $m = \{163, 233, 283, 409, 571, 1223\}$, is presented. It is observed that the reduction of area-delay product (ADP) with respect to Chiou's method [19] are 91%, 93.5%, 93.9% 95.8%, 96.8%, 95.5% for different m, while average reduction in ADP is 94.42%. It can also be seen that the delayed improvement for different m is 39.1%, 53.3%, 51.1%, 65.3%, 69.1%, 81.3%, and an average delay improvement is 59.87%.

The subsequent section demonstrates an application of our proposed polynomial modulo multiplier. A polynomial-based lattice homomorphic encrypting scheme utilizing the proposed tensor method to calculate modulo operations is realized. The detailed formulation and implementation with security parameter analysis are elaborated in the next section.

6 Application of Proposed Multiplier

Lattice-based PQC is one of the alternatives to building a strong resistance against various malicious intrusions. Lattices are subgroups of $\mathbb{R}^m$. A lattice L is a set of linearly independent basis vectors $(b_1, \ldots, b_n)$ in $\mathbb{R}^m$, such that $L(b_1, \ldots, b_n) = \sum_{i=1}^{n} x_i b_i, x_i \in \mathbb{Z}, \forall$ integer linear combinations of b_i. Here, n is the dimension of L. If ϕ represents the function evaluated over plaintexts m_1 and m_2, then a homomorphic encryption scheme computes $\phi(m_1, m_2)$ using encryption $Enc(m_1)$ and $Enc(m_2)$, without the knowledge of m_1 and m_2.

The modulo polynomial operations are employed to formulate a lattice homomorphic encryption scheme [43] and are summarized in the following steps.

- First, a message $m \in \mathbb{F}$, is mapped to $p(m)$, i.e., a polynomial in the polynomial ring $\mathbb{F}_q[x_1, ..., x_n]$.
- Second, the coefficient vector of the polynomial obtained in the above step is mapped to a ciphertext matrix C, where $\mathbb{F}$ is a field, and $\mathbb{F}_q$ is a field of cardinality q and a prime number.

Considering $\mathbb{F}_q$ as a plaintext space, $n \in \mathbb{N}$ is the number of variables and $c = f(\lambda)$, where λ is the security parameter. $m \in \mathbb{F}_q$ is mapped to $p(m)$ as depicted in equation 9.

$$p(m) = m + \sum_{i=1}^{n} r_i g_i \tag{9}$$

where $r_i \in \mathbb{F}_q[x_1, \ldots, x_n]$ and g_i are chosen randomly from $\mathbb{F}_q[x_1, \ldots, x_n]$. The coefficients of $p(m)$ are mapped to matrix C as described below.

- A random matrix $\tilde{C}$ of dimension $N \times N$ is chosen, where $N = (d+1)^n$, and d is the size of the vector space. One column is chosen randomly at a time, using a column shift operator δ. Thereafter, δ^{th} column of the matrix $\tilde{C}$ is replaced with a coefficient vector.
- Later, two random invertible matrices (M, R) are chosen to construct the final ciphertext $C = M.\tilde{C}.R$

The homomorphic scheme employing the same value of δ is used to evaluate ciphertexts. While encrypting zeros with this δ, the resulting ciphertext can span an entire subspace of $\mathbb{F}_q^{N \times N}$. The upper limit of ciphertexts during homomorphic encryption is equal to the dimension of this subspace. For homomorphic multiplication, the product of ciphertexts is computed using a bilinear map. The public evaluation key and linear maps, M and R, comprise this bilinear map. The product of two ciphertexts, C_1, and C_2, is determined using a bilinear map B and is represented by equation 10.

$$B[p(m_1), p(m_2)] = p(m_1).p(m_2) \quad mod \quad f_{pk} \tag{10}$$

To calculate f_{pk}, a chain of rings is constructed. $R_0 \subseteq R_1 \subseteq \ldots R_{n-1}$ and the i^{th} term can be expressed as equation 11.

$$R_i = R_{i-1}[x_i]/\langle f_i \rangle, \quad 1 < i < n-1 \tag{11}$$

The polynomials, $p_i(m)$, in all the stages are the members of a ring. Considering all symbols with their usual meaning, the product of two ciphertexts, C_1 and C_2, based on the proposed tensor matrix method, can be expressed as equations 12 and 13. c_i^1 and c_i^1 are the columns vectors of C_1 and C_2.

$$T(C_1, C_2) = [T_1(C_1, C_2), T_2(C_1, C_2) \ldots T_N(C_1, C_2)] \tag{12}$$

$$T_i(C_1, C_2) = [{c_1^1}^T \ldots {c_N^1}^T] T_i \begin{bmatrix} c_1^2 \\ . \\ . \\ . \\ c_N^2 \end{bmatrix} \tag{13}$$

The security of such encryption schemes depends on an arithmetic field in which ciphertexts are mapped and perform operations. If the number of variables increases along with their degree of polynomials, i.e., field, it enhances the security. The relationship between field and security is illustrated in Table 11. The relationship of security parameters is expressed in equation 14.

$$q \approx \lambda^{b+\mu}; \alpha = 1/(\lambda^{\mu} log_b^{\lambda\sqrt{\lambda}}) \tag{14}$$

The description of various parameters in Table 11 is as follows. λ is the parameter to generate polynomials over a

TABLE 11
Security Parameters Comparison [44] for Different Field Sizes (for $b = 2; k = \sqrt{2log(100)})$

λ	μ	n	N	α	q	$\alpha q > 2\sqrt{N}$
32	1	16	140	0.0041666666667	3277	False
32	3	28	344	4.069010416667E-06	3355444	False
64	1	18	160	0.0017361111111	26215	True
64	3	26	575	4.238552517361E-07	107374183	False
128	1	24	310	0.0007440476191	209716	True
128	3	28	378	4.541306268602E-08	3435973837	True
256	1	39	790	0.0003255208333	1677722	True
256	3	45	1003	4.967053731283E-09	109951162778	True
512	1	66	2225	0.000144675926	13421773	True
512	3	70	2598	5.518948590314E-10	3518437208884	True

finite field of prime size $q(\lambda)$, where q is a prime number, n is the number of variables, α is the parameter to generate discrete Gaussian distribution ($\alpha > 0$), μ states levels of multiplication, and N is the number of elements in the polynomial ring. *False* in the last column states that the scheme based on the parameters in a particular row is not strong enough against quantum computer-based attacks, whereas *True* shows a good measure of resistance against such attacks.

All the polynomial multiplications mentioned above are modulo operations and integral parts of lattice cryptosystems. Therefore, polynomial modulo multipliers proposed in this paper can be utilized to implement a secure and efficient lattice homomorphic encryption scheme. Therefore, the proposed multivariate polynomial modulo multiplier can be used efficiently to realize lattice-based homomorphic post-quantum cryptography algorithms [43], which can be utilized in cryptographic hardware accelerators [45].

7 Conclusion

The paper presents a novel design methodology of multivariate polynomial modulo multiplication based on tensors. The proposed multiplier is realized on a hardware platform and is compared with other state-of-the-art multipliers. Our proposed multiplier outperforms other multipliers in terms of resource utilization and power consumption. An embarrassingly parallel and scalable architecture of the proposed multiplier is described in this paper, which performs linearly for single variable polynomial multiplication and quadratically for multivariate polynomial multiplication in the worst case. However, its average and best case computational complexity are linear for both the single variable and multivariate polynomials. The proposed multiplier consumes $6.5\times$ less power for single variable polynomial multiplication and is more than $6\times$ faster than other polynomial modulo multipliers. As per our knowledge, a multivariate polynomial modulo multiplier is realized for the first time on the hardware platform. Its performance is analyzed with the multivariate polynomial of nine variables, where the maximum degree of each variable is 128. For this polynomial, chip area, power, and delay estimated by our proposed multiplier are $50113.9\mu m^2$, $1.485\ W$, and $8.812\ nS$, respectively. It is found that the performance of our multivariate multiplier is linear with respect to the parameters mentioned above. The proposed multiplier is validated using an LWE-based lattice multivariate encryption scheme to measure its efficacy. This indicates that our proposed multiplier may be ideal for efficiently implementing homomorphic encryption algorithms on various platforms.

References

[1] W. . Huang, C. H. Chang, C. W. Chiou, and S. . Tan, "Non-xor approach for low-cost bit-parallel polynomial basis multiplier over $gf(2^m)$," *IET Information Security*, vol. 5, no. 3, pp. 152–162, 2011.

[2] C. Y. Lee and C. W. Chiou, "Efficient design of low-complexity bit-parallel systolic hankel multipliers to implement multiplication in normal and dual bases of $gf(2^m)$," *IEICE Trans. Fund. Electron. Commun. Comput. Sci.*, vol. E88-A, no. 11, pp. 3169–3179, 2005.

[3] J. . Wang, H. W. Chang, C. W. Chiou, and W. . Liang, "Low-complexity design of bit-parallel dual-basis multiplier over $gf(2^m)$," *IET Information Security*, vol. 6, no. 4, pp. 324–328, 2012.

[4] A. Reyhani-Masoleh and M. Hasan, "Low complexity bit parallel architectures for polynomial basis multiplication over gf(2^m)," *IEEE Transactions on Computers*, vol. 53, no. 8, pp. 945–959, 2004.

[5] Chiou-Yng Lee, Erl-Huei Lu, and Jau-Yien Lee, "Bit-parallel systolic multipliers for $gf(2^m)$ fields defined by all-one and equally spaced polynomials," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 385–393, 2001.

[6] S. Talapatra, H. Rahaman, and J. Mathew, "Low complexity digit serial systolic montgomery multipliers for special class of $gf(2^m)$," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 5, pp. 847–852, 2010.

[7] L. H. Chen, P. L. Chang, and C. Y. Lee, "Scalable and systolic dual basis multiplier over $gf(2^m)$," *Int. J. Innovat. Comput. Inf. Control*, vol. 7, no. 3, pp. 1193–1208, 2011.

[8] Y. Hua, J.-M. Lin, C. Chiou, C.-Y. Lee, and Y. Liu, "A novel digit-serial dual basis systolic karatsuba multiplier over $gf(2^m)$," *Chinese Journal of Computers*, vol. 23, pp. 80–94, 07 2012.

[9] J. Xie, P. K. Meher, M. Sun, Y. Li, B. Zeng, and Z. H. Mao, "Efficient fpga implementation of low-complexity systolic karatsuba multiplier over $f(2^m)$ based on nist polynomials," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 7, pp. 1815–1825, July 2017.

[10] P. L. Chang, Y. K. Yang, L. H. Chen, F. H. Hsieh, and C. Y. Lee, "Low-complexity dual basis digit serial $gf(2^m)$ multiplier," *ICIC Express Letters*, vol. 3, no. 4, pp. 1113–1118, 2009.

[11] Po-Lun Chang, F. Hsieh, L. Chen, and C. Lee, "Efficient digit serial dual basis $gf(2^m)$ multiplier," in *2010 5th IEEE Conference on Industrial Electronics and Applications*, 2010, pp. 166–170.

[12] E. Ozcan and A. Aysu, "High-level synthesis of number-theoretic transform: A case study for future cryptosystems," *IEEE Embedded Systems Letters*, vol. 12, no. 4, pp. 133–136, 2020.

[13] F. Yarman, A. C. Mert, E. ztrk, and E. Sava, "A hardware accelerator for polynomial multiplication operation of crystals-kyber pqc scheme," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1020–1025.

[14] G. B. Agnew, R. C. Mullin, I. M. Onyszchuk, and S. A. Vanstone, "An implementation for a fast public-key cryptosystem," *J. Cryptology*, vol. 3, pp. 63–79, 1991.

[15] Y.-H. Ying, L. Jim-Min, C. Che-Wun, L. Chiou-Yng, and L. Yong-Huan, "Low space-complexity digit-serial dual basis systolic multiplier over galois field $gf(2^m)$ using hankel matrix and karatsuba algorithm," *IET Information Security*, vol. 7, pp. 75–86(11), June 2013. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/iet-ifs.2012.0227

[16] T. Alshawi, A. Bentrcia, and S. Alshebeili, "Design and low-complexity implementation of matrix-vector multiplier for iterative methods in communication systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 12, pp. 3099–3103, Dec 2015.

[17] M. A. Hasan, A. H. Namin, and C. Negre, "Toeplitz matrix approach for binary field multiplication using quadrinomials," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 3, pp. 449–458, March 2012.

[18] D. Zoni, A. Galimberti, and W. Fornaciari, "Flexible and scalable fpga-oriented design of multipliers for large binary polynomials," *IEEE Access*, vol. 8, pp. 75 809–75 821, 2020.

[19] C. W. Chiou, C. Lee, J. Lin, Y. Yeh, and J. Pan, "Low-latency digit-serial dual basis multiplier for lightweight cryptosystems," *IET Information Security*, vol. 11, no. 6, pp. 301–311, 2017.

[20] A. C. Mert, E. Karabulut, E. ztrk, E. Sava, M. Becchi, and A. Aysu, "A flexible and scalable ntt hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 346–351.

[21] T. Pöppelmann and T. Güneysu, "Towards practical lattice-based public-key encryption on reconfigurable hardware," in *Selected Areas in Cryptography – SAC 2013*, T. Lange, K. Lauter, and P. Lisoněk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 68–85.

[22] T. Fritzmann, G. Sigl, and J. Seplveda, "Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography," Cryptology ePrint Archive, Report 2020/446, 2020, https://ia.cr/2020/446.

[23] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-lwe cryptoprocessor," in *Cryptographic Hardware and Embedded Systems – CHES 2014*, L. Batina and M. Robshaw, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 371–391.

[24] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng, "Vpqc: A domain-specific vector processor for post-quantum cryptography based on risc-v architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2672–2684, 2020.

[25] T. Zhang and K. Parhi, "Systematic design of original and modified mastrovito multipliers for general irreducible polynomials," *IEEE Transactions on Computers*, vol. 50, no. 7, pp. 734–749, 2001.

[26] Y. H. Chang, J. S. Wang, and R. C. T. Lee, "Generating all maximal independent sets on trees in lexicographic order," in *Proceedings ICCI '92: Fourth International Conference on Computing and Information*, May 1992, pp. 34–37.

[27] L. Carlitz, "The arithmetic of polynomials in a galois field," *American Journal of Mathematics*, vol. 54, no. 1, pp. 39–50, 1932.

[28] P. Scott, S. Tavares, and L. Peppard, "A fast vlsi multiplier for gf(2^m)," *IEEE Journal on selected Areas in Communications*, vol. 4, no. 1, pp. 62–66, 1986.

[29] B. Singh and M. U. Siddiqi, "Multivariate polynomial products over modular rings using residue arithmetic," *IEEE transactions on signal processing*, vol. 43, no. 5, pp. 1310–1312, 1995.

[30] S. Zaks, "Lexicographic generation of ordered trees," *Theoretical Computer Science*, vol. 10, no. 1, pp. 63–82, 1980. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397580900730

[31] J. Wiedermann, "The complexity of lexicographic sorting and searching," in *Mathematical Foundations of Computer Science 1979*, J. Bečvář, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1979, pp. 517–522.

[32] U. Dowerah and S. Krishnaswamy, "A new multiplication technique for lwe based fully homomorphic encryption," *IEEE Letters of the Computer Society*, vol. 3, no. 2, pp. 62–65, 2020.

[33] S. Ivasiev, M. Kasianchuk, I. Yakymenko, R. Shevchuk, M. Karpinski, and O. Gomotiuk, "Effective algorithms for finding the remainder of multi-digit numbers," in *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*, 2019, pp. 175–178.

[34] P. Chen, S. N. Basha, M. Mozaffari-Kermani, R. Azarderakhsh, and J. Xie, "Fpga realization of low register systolic all-one-polynomial multipliers over $gf(2^m)$ and their applications in trinomial multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 725–734, 2017.

[35] M. Cenk, M. A. Hasan, and C. Negre, "Efficient subquadratic space complexity binary polynomial multipliers based on block recombination," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2273–2287, 2014.

[36] J.-H. Guo and C.-L. Wang, "Systolic array implementation of euclid's algorithm for inversion and division in gf(2^m)," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1161–1167, 1998.

[37] H. Brunner, A. Curiger, and M. Hofstetter, "On computing multiplicative inverses in gf(2^m)," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 1010–1015, 1993.

[38] M.-S. Kang and K.-H. Lee, "Hardware implementation of fast division algorithm for gf(2^m)," in *2006 8th International Conference Advanced Communication Technology*, vol. 1, 2006, pp. 117–119.

[39] V. Xilinx, "Design suite," 2018. [Online]. Available: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html

[40] W. Mao and G. J. Milne, "An automated proof technique for finite-state machine equivalence," in *Computer Aided Verification*, K. G. Larsen and A. Skou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 233–243.

[41] E. Karl, Z. Guo, Y. Ng, J. Keane, U. Bhattacharya, and K. Zhang, "The impact of assist-circuit design for 22nm sram and beyond," in *2012 International Electron Devices Meeting*, 2012, pp. 25.1.1–24.1.4.

[42] F. Dinechin and B. Pasca, "Large multipliers with less dsp blocks," 08 2009.

[43] U. Dowerah and S. Krishnaswamy, "A new symmetric key homomorphic encryption scheme," in *23rd International Symposium on Mathematical Theory of Networks and Systems*, 2018, pp. 885–892.

[44] M. R. Albrecht, P. Farshim, J.-C. Faugere, and L. Perret, "Polly cracker, revisited," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2011, pp. 179–196.

[45] B. Paul, T. K. Yadav, B. Singh, S. Krishnaswamy, and G. Trivedi, "A resource efficient software-hardware co-design of lattice-based homomorphic encryption scheme on the fpga," *IEEE Transactions on Computers*, pp. 1–14, 2022.

Bikram Paul completed his MTech degree from National Institute of Technology, Agartala in Department of Electronics and Electrical Engineering, in 2014. Currently, he is pursuing PhD degree from Indian Institute of Technology, Guwahati. His area of interests are cryptographic processor design for quantum secure application, post-quantum cryptography, next-gen quantum superconductive electronics circuit design and VLSI circuits design for IoT applications.

Angana Nath completed her MTech degree from Indian Institute of Technology, Guwahati in Department of Electronics and Electrical Engineering. She was working as senior engineer at Qualcomm Inc and staff engineer at Western Digital. Currently, she is pursuing MBA from University of Wisconsin-Madison, United States.

Srinivasan Krishnaswamy received Ph.D. degree in Department of Electrical Engineering, Indian Institute of Technology Bombay, Mumbai. He joined as an Assistant Professor, Department of Electronics and Electrical Engineering, Indian Institute of Technology Guwahati in 2012. His research interests are Cryptography and Control Systems.

Jan Pidanic received the Ph.D. degree on Methods for computing cross ambiguity function from University of Pardubice, Pardubice in the year of 2012. Then he joined as an Assistant Professor, Department of Electrical Engineering and Informatics, University of Pardubice, Pardubice, Czech Republic. His research interests are Control Systems, communication, information security etc.

Zdenek Nemec received the Ph.D. degree on Application of a data communication to a Mobile Station Localization from University of Pardubice, Pardubice in the year of 2007. He joined as an Assistant Professor, Department of Electrical Engineering and Informatics, University of Pardubice, Pardubice, Czech Republic in the year of 2008. His research interests are mobile communication, navigation systems, Informatics etc.

Gaurav Trivedi received Ph.D. degree in Department of Electrical Engineering, Indian Institute of Technology Bombay, Mumbai in 2007. He joined Department of Electronics and Electrical Engineering, Indian Institute of Technology Guwahati in 2011. He is currently an Associate Professor, Department of Electronics and Electrical Engineering, Indian Institute of Technology Guwahati. His research interests include circuit simulation (Analog & Digital) and VLSI CAD, high performance computing, VLSI cryptographic circuits design and hardware security. He is a member of IEEE.