IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 22, 2021, accepted January 10, 2022, date of publication January 14, 2022, date of current version January 21, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3143804

Efficient Homomorphic Encryption Accelerator With Integrated PRNG Using Low-Cost FPGA

INFALL SYAFALNI[1,2], (Member, IEEE), GILBERT JONATAN[2], NANA SUTISNA[1,2], (Member, IEEE), RAHMAT MULYAWAN[1,2], (Member, IEEE), AND TRIO ADIONO[1,2], (Member, IEEE)

[1]Electrical Engineering Department, School of Electrical Engineering and Informatics, Bandung Institute of Technology, Bandung 40132, Indonesia
[2]University Center of Excellence on Microelectronics, Bandung Institute of Technology, Bandung 40132, Indonesia

Corresponding author: Infall Syafalni (infall@ieee.org)

This work was supported by the Indonesian Ministry of Research and Technology/National Agency for Research and Innovation (Kemenristek/BRIN) under National Competition Research Grant (Program Penelitian Kompetitif Nasional).

ABSTRACT With recent development in internet speed and reliability, cloud computing has become a more reliable solution for the user. In many cases where data privacy is critical, fully homomorphic encryption (FHE) can be a security solution for securing cloud computing. FHE enables computation on encrypted data, hence it ensures data privacy in case of cloud computing. One popular scheme of FHE is the BFV homomorphic encryption scheme, which is based on ring learning with error (RLWE) computation. The BFV scheme uses ring polynomials as the main object, hence its encryption, decryption, and evaluation require high-degree polynomial multiplication. In this paper, we present comprehensive design and implementation of a hardware architecture to accelerate encryption and decryption in BFV scheme. Our accelerator uses convolution approach for calculating a polynomial multiplication. To implement the convolution, we use a systolic array to calculate polynomial convolution followed by a simple delayed subtraction to calculate polynomial modulo reduction inside our accelerator's core. Moreover, we use a built-in Gaussian pseudo-random number generator (PRNG) to generate Gaussian noise in the encryption operations. Finally, we implement the 1024 degrees BFV accelerator on the Xilinx PYNQ Z1 board and compare the encryption and decryption performances to other methods as well as a software implementation on Intel Core i7 with 8GB memory. Experimental results show that our accelerator outperforms the clock cycles of other methods with the same polynomial degrees 1024 up to 22×. Moreover, our proposed Gaussian PRNG has better 2× correlation compared to the rotation-only-based PRNG. Finally, our accelerator accelerates up to 9× for encryption and 3.5× for decryption as well as 6.8× for overall compared to Microsoft SEAL on Intel Core i7 processor with 8GB memory. The proposed design is scalable for higher degrees polynomial multiplication and useful for security technology such as high-speed secure cloud computing, blind computing, and secure communication.

INDEX TERMS BFV scheme, fully homomorphic encryption, Gaussian PRNG, hardware accelerator, systolic array.

I. INTRODUCTION

Recent development in internet speed and reliability around the world enables cloud computing to be a reliable and convenient solution for our software and computation needs. By relying on the internet connection, users can run software that requires more computing power by using services from cloud computing providers, instead of resources on their device. Cloud computing has many uses [1] and is considered more secure in terms of data recovery in an unwanted event. In most cloud computing, the user data privacy is only limited by term and agreement between the two parties. In some cases where data confidentiality is critical even towards the cloud service provider, cloud computing can be inconvenient for clients. One solution to this problem is fully homomorphic encryption.

Fully homomorphic encryption is a new technology in cryptography that enables computation on encrypted data.

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

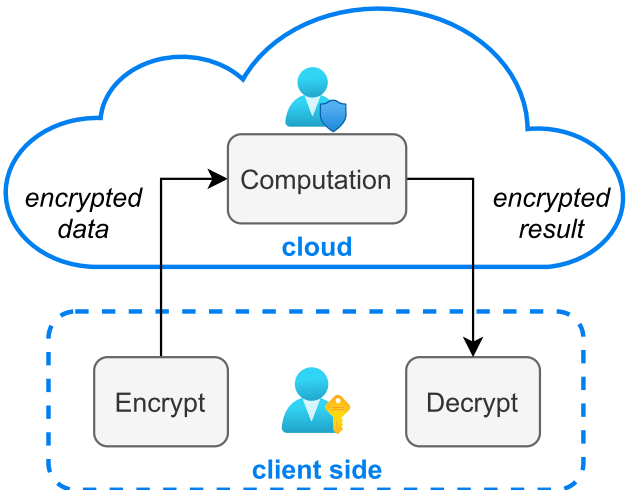


FIGURE 1. Cloud computing with FHE.

The computation can be performed without the need of decryption, hence FHE can ensure the security of the data not only from an attacker, but also from the cloud service provider. Although the concept has been around for a while, the first FHE scheme was proposed in 2009 by Gentry [2]. His breakthrough work enables arbitrary computation in FHE with expense of computing power. After Gentry's scheme, new FHE schemes keep emerging ever since [3]–[8]. One of the well known schemes was proposed by Fan [7] by extending the Brakerski's LWE scheme [4] into its ring variation.

Figure 1 shows a cloud computing scheme using FHE as the encryption method choice of both sides. The encryption and decryption operation only happens on the client side hence the actual data only visible to the client. The cloud service provider only performs computation in the ciphertext domain, theoretically with arbitrary computation depth, most operations can be performed in the ciphertext domain. The result of the computation will still be encrypted with the user's secret key. All of these computations can be achieved at the expense of computing power.

In this work, we focus on the BFV homomorphic encryption scheme, one of the most well known schemes that exist today. The main object of the BFV scheme is ring polynomial and a lot of its operations are based on polynomial multiplication and addition on a predefined ring. The polynomial multiplication usually took longer to calculate compared to polynomial addition hence most of the existing work focuses on accelerating this operation using a hardware accelerator.

There are several existing works related to hardware accelerator design to accelerate polynomial multiplication in the BFV homomorphic encryption scheme. To the best of our knowledge, the first one came in 2016 [9] they use the Karatsuba algorithm [10] to calculate the polynomial multiplication in the homomorphic multiplication. At the time one of the most promising schemes was YASHE [6], but its security was reduced due to a recent attack [11], especially on decision small polynomial ratio (DSPR) assumption [12]. Hence some recent works start to focus more on a less efficient BFV scheme. The authors in [9] show that Karatsuba can be an alternative to FFT, despite their design having some constraint and requiring large amounts of resources. Another work to accelerate BFV homomorphic multiplication came in 2018 [13], they use CRT and NTT for fast polynomial multiplication.

On the high performance side, the NTT method is often used to calculate polynomial multiplication. Instead of using real numbers like the FFT algorithm, NTT algorithm only uses integer number arithmetic, hence making it easier to implement in hardware. In 2019, in more recent work from [14], they designed a custom co-processor for the BFV homomorphic encryption scheme. Lastly at the time of writing this paper, the most recent work came from Mert et al [15], they utilized their NTT-based polynomial multiplier architecture in [16] to accelerate encryption and decryption of BFV scheme. The hardware accelerator is designed to accelerate the encrypt and decrypt function in Microsoft SEAL [17].

In this work, we use optimized polynomial multiplication is proposed. Our work aims to accelerate encryption and decryption operation of BFV homomorphic encryption, especially for the python environment using Xilinx's Python productivity for Zynq (PYNQ). Our contribution in this paper is listed as follows:

1) We design and implement comprehensively a hardware accelerator for encryption and decryption of the BFV homomorphic scheme on a low cost FPGA. The hardware accelerator is implemented as an overlay for the Xilinx PYNQ environment.
2) We propose an efficient systolic array to calculate polynomial convolution with all processing elements are optimized for the BFV scheme.
3) We propose a ring polynomial multiplication core with a FIFO and a subtractor to perform the polynomial modulo reduction.
4) We design and implement a hardware of Gaussian pseudo-random number generator for encryption mode by using hybrid rotation and split methods.
5) Our accelerator outperforms other methods with the polynomial degrees 1024 up to 22×. Our proposed PRNG has better 2× randomness factor compared to rotation-only PRNG. Moreover, our accelerator accelerates up to 9× for encryption and 3.5× for decryption compared to Microsoft SEAL on Intel Core i7 processor with 8GB memory.

This paper is organized as follows: Section I introduces the work. Section II explains the definitions and basic properties for FHE. Section III shows the proposed BFV scheme and the methods used in the accelerator. Section IV shows the hardware architecture of our accelerator. Section V shows memory configuration for encryption and decryption mode. Section VI shows implementation results and comparison for our hardware Gaussian pseudo-random number generator and our accelerator. Finally, Section VII summarizes our conclusion.

II. DEFINITIONS AND BASIC PROPERTIES

A. FULLY HOMOMORPHIC ENCRYPTION (FHE)

Fully Homomorphic Encryption is a new technology in cryptography that enables computation to be performed on encrypted data without decryption. This concept has been around for a while since its first introduction by Rivest [18] under the name privacy homomorphisms. However, the operation is limited to only addition or multiplication in the plaintext space. This limitation remains unsolved until a breakthrough result from Gentry [2] in 2009, in which he presented a scheme that allows arbitrary computation on the ciphertext by showing how to modify his scheme to make it bootstrappable or self evaluating its own decryption circuit. In his work he also shows that any bootstrappable somewhat-homomorphic encryption scheme can be transformed into a fully homomorphic encryption.

A fully homomorphic encryption (FHE) scheme theoretically can perform a limitless number of additions or multiplications in the plaintext space, this is done cleverly by ''refreshing'' the noise or using a self-referential property to evaluate its own decryption circuit. There are several types of homomorphic encryption based on supported operation and evaluation depth:

- **Partially Homomorphic Encryption:** Only one operation can be done, either addition or multiplication in the plaintext domain.
- **Somewhat Homomorphic Encryption:** Both addition and multiplication in the plaintext domain, but have a limited number of operations depending on the operation and noise.
- **Leveled Homomorphic Encryption:** Both addition and multiplication in the plaintext domain on a limited evaluation circuit depth.
- **Fully Homomorphic Encryption:** Both addition and multiplication in the plaintext domain with arbitrary circuit depth.

Fully homomorphic encryption scheme can be considered ring homomorphic. A ring in mathematics is a set R equipped with two operations $+$ and $\times$ satisfying the ring axioms [19]. Given R and S are two rings, then a ring homomorphism can be explicitly express as a function

$$f : R \to S$$

such that

$$f(a+b) = f(a) + f(b)$$
$$f(a \times b) = f(a) \times f(b)$$

for all a and b in R. Figure 2 shows graphical interpretation of ring homomorphism.

B. BFV HOMOMORPHIC ENCRYPTION SCHEME

In this section, we presented the BFV scheme from Fan and Vercauteren [7], they extended Brakerski's scheme [4] to the ring learning with error (RLWE) variation, which is an algebraic variant of learning with error (LWE) [20].

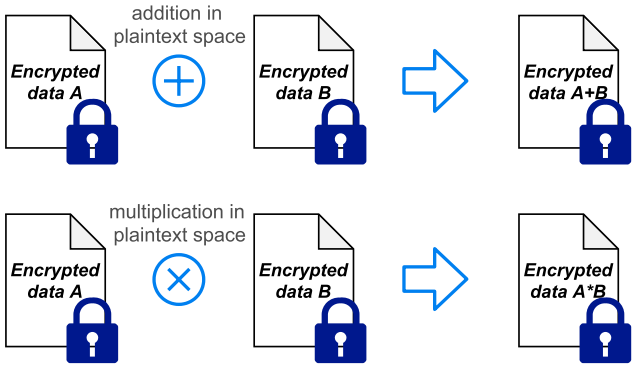


FIGURE 2. Ring homomorphism in fully homomorphic encryption.

Before presenting the proposed scheme, we start with a brief explanation for the basic notations and processes in the BFV Homomorphic Encryption scheme as presented in [7].

Definition 2.1: A ring of modulo polynomial is represented as $R = Z[x]/(f(x))$, *where* $Z[x]$ *is a polynomial and* $f(x)$ *is a cyclotomic polynomial* $\phi_m(x)$, i.e. *the minimal polynomial of the primitive m-th roots of unity, and* x *is a variable. In this work, we use* $f(x) = x^n + 1$ *with* $n = 2^d$.

Definition 2.2: Let a be uniformly sampled from the set S that is indicated by

$$a \xleftarrow{\chi} S,$$

where χ *indicates the discrete Gaussian distribution.*

Definition 2.3: Let n, q, and t be integers representing the degrees of polynomial modulus, the ciphertext coefficient modulus, and the plaintext coefficient modulus, respectively. Suppose that Z_q *denotes the set of integers* $(-q/2, q/2]$, *thus* R_q *is the set of polynomials in R with coefficient* Z_q. *The plaintext and ciphertext spaces are in the rings* R_t *and* R_q, *respectively, for* $q > t > 1$. *Neither q nor t have to be prime or coprime.*

Definition 2.4: The ratio between the ciphertext coefficient modulus q and the plaintext coefficient modulus t is defined as $\Delta = q/t$.

Definition 2.5: Let $\lfloor \cdot \rfloor$, $\lfloor \cdot \rceil$, *and* $[\cdot]_q$ *be the flooring operation, the rounding operation to the nearest integer, and the coefficient reduction operation by modulo q, respectively.*

Definition 2.6: Secret key generation is defined by n samples of spaces in the rings R_2 *as follows*

$$s \leftarrow R_2^n.$$

In this case, we can see a secret key s is in the form of binary numbers in n arrays.

Lemma 2.1: Public key generation is represented by the following

$$(p_0, p_1) = ([-(a \cdot s + e)]_q, a),$$

where p_0 *and* p_1 *are the public keys,* $a \leftarrow R_q^n$ *and* $e \leftarrow \chi$.

Lemma 2.2: Encryption is represented by the following

$$(c_0, c_1) = ([\Delta \cdot m + p_0 \cdot u + e_1]_q, [p_1 \cdot u + e_2]_q),$$

TABLE 1. BFV parameter selection in this work.

Parameter	Symbol	Value
Polynomial modulus	n	2^{10}
Plaintext coefficient modulus	t	2^8
Ciphertext coefficient modulus	q	2^{32}

where c_0 and c_1 are the ciphertexts, p_0 and p_1 are the public keys, $m \in R_t$, $u \leftarrow R_2^n$, $e_1, e_2 \leftarrow \chi$, and $\Delta = q/t$.

Lemma 2.3: Decryption is represented by the following

$$m = \left[\left\lfloor \frac{[c_0 + c_1 \cdot s]_q}{\Delta} \right\rceil \right]_t,$$

where c_0 and c_1 are the ciphertexts, s is the secret key, q is the ciphertext coefficient modulus, t is the plaintext coefficient modulus, and $\Delta = q/t$.

Fan and Vercauteren [7] remark that neither q or t have to be a prime, nor that t and q are coprime. In this work, we propose a hardware architecture to accelerate encryption and decryption operation of BFV scheme. Our parameter selection is shown in Table 1. To optimize the hardware implementation, we choose q and t as a power of 2, where $q = 2^{32}$ (32-bit) and $t = 2^8$ (8-bit). Lastly, we also choose $n = 2^d = 1024$, where $d = 10$.

III. PROPOSED BFV HOMOMORPHIC ENCRYPTION USING CONVOLUTION WITH INTEGRATED PRNG

This section explains our exploration of the proposed BFV homomorphic encryption and decryption using convolution. Moreover, we optimize the design by considering the pattern on the encryption and decryption calculations, minimizing the multiplication operations on the convolution, applying polynomial modulo optimization, and finally designing the Gaussian pseudo-random number generator (PRNG) using linear feedback shift registers (LFSR).

A. POLYNOMIAL MULTIPLICATION PATTERN ON ENCRYPTION-DECRYPTION IN THE BFV SCHEME

The BFV scheme has several polynomial multiplications. In the encryption process, to produce the ciphertexts (c_0, c_1), two polynomial multiplications are required. Moreover, in decryption process, one polynomial multiplication is required. These polynomial multiplications follow a simple computation pattern as shown in Figure 3, which is a multiplication between an R_q polynomial and an R_2 polynomial, followed by addition with an R_q polynomial. Note that R_q and R_2 can be seen as arrays of the polynomial coefficients in q-field and 2-field (binary), respectively.

In this work, we optimize our proposed accelerator's core by performing the computation pattern as shown in Figure 3. The accelerator's core computes the multiplication followed by the polynomial reduction by the polynomial modulus that defines the ring and performs polynomial addition at the end. We use convolution approach for the polynomial multiplication operations while we implement a simple subtractor

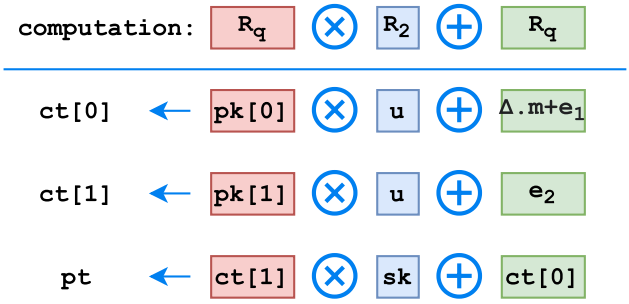


FIGURE 3. Computation pattern.

for the polynomial reduction. Both techniques will be further discussed in the next subsections.

B. POLYNOMIAL MULTIPLICATION AND CONVOLUTION

One way to evaluate polynomial multiplication is through a convolution method. It can be proven intuitively by multiplying, sliding, and adding two polynomials or formally with mathematical proof. In this section, we will show a formal proof from [21] that a polynomial multiplication can be seen as a convolution, which is necessary to understand our proposed accelerator.

Lemma 3.4: Polynomial multiplication can be seen as a convolution [21]. Supposed that $G(x)$ and $H(x)$ are polynomial of degrees $n-1$ in x, where x is the polynomial variable. We have

$$G(x) = \sum_{i=0}^{n-1} g_i x^i, \tag{1}$$

and

$$H(x) = \sum_{i=0}^{n-1} h_i x^i, \tag{2}$$

such that

$$\begin{aligned} Y(x) &= G(x) \cdot H(x) \\ &= \sum_{k=0}^{2n-2} y_k x^k, \end{aligned} \tag{3}$$

where $y_k = \sum_{i=0}^{k} g_i h_{k-i}$.

Proof: The multiplication results of Eq. (3) will be a polynomial degree of $2n-2$. In polynomial multiplication, each coefficient x is obtained by summing all products $g_i h_{k-i}$, where k is the degree of x. Hence, by multiplying Eq. (1) and Eq. (2), we form a discrete convolution for finding coefficients of the multiplied polynomials y_k as follows:

$$\begin{aligned} y_k &= g_0 h_k + g_1 h_{k-1} + \ldots + g_{k-1} h_1 + g_k h_0 \\ &= \sum_{i=0}^{k} g_i h_{k-i} \\ &= (g * h)[k], \end{aligned} \tag{4}$$

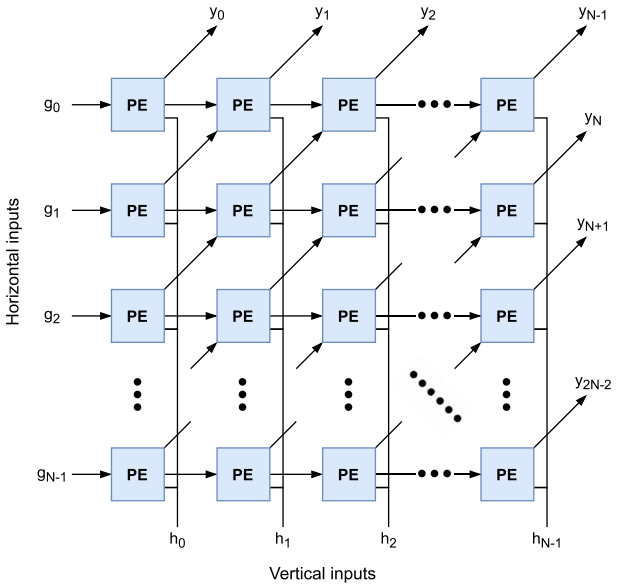


FIGURE 4. 2D systolic array for $(N-1)$-degree polynomial multiplication.

where $[*]$ is the convolution operation, and k is the degree of the corresponding variable x in the multiplied polynomials. This means that the multiplication of two polynomials can be treated as the convolution of two polynomials and vice versa. Thus, we have the lemma. □

By using this method, in the next subsection, we will explain a 2-dimensional systolic array to calculate polynomial multiplication. Later, we optimize the design into one dimensional systolic array and the systolic cells for BFV scheme.

C. 2D SYSTOLIC ARRAY FOR POLYNOMIAL CONVOLUTION

This subsection previews the basic of polynomial convolution to accelerate the polynomial multiplication presented in [22] that uses systolic arrays as the core architecture. The illustration of a 2-dimensional systolic arrays is shown in Figure 4.

A set of cells in each diagonal line (*i.e.* lines pointing to northeast) is responsible to calculate the sum of all products of $g_i h_{k-i}$ as described in Eq. (4) in the previous subsection. In other words, the output of each diagonal line is coefficient y_k for x^k. At every clock cycle, the input and output of the systolic cells are moving as indicated by the arrows. The right arrows forward the input to the next right cell and the diagonal arrows send the output diagonally to the next cell. The horizontal input will change, while the vertical input is static for the entire computation.

The output function of the systolic cell or processing element is out $= (a \times b) + sum$ as shown by Figure 5. The processing element has 3 inputs, consisting of 2 polynomial coefficients and 1 sum input from another cell. Moreover, it consists two registers to store the sum of the product and the forward of the horizontal coefficient. The processing element multiply the 2 polynomial coefficients and then sum the result

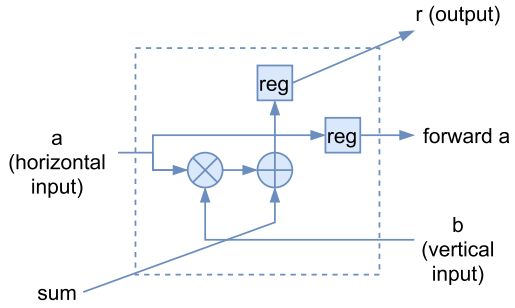


FIGURE 5. Processing element or systolic cell of 2D systolic array.

with the sum input from another cell, before producing the output in the next clock cycle. In the diagonal line, every cell will calculate one product of $g_i h_{k-i}$ and sum it with the result of the previous cells. The top-most or right-most cells will generate all the coefficients of y after N clock cycles.

Theorem 3.1: Suppose that we have integers i and j with $[0, 1, \ldots, N]$, where $N > 0$, and (i, j) is the row and column of the systolic array as shown in Figure 4. The first row is denoted by $i = 0$ and the first column is denoted by $j = 0$. The output of a systolic cell that is located in ith row and jth column is denoted as $out(i, j)$. The output of each cell can be obtained from the following recursion

$$out(i, j) = g_i h_j + out(i + 1, j - 1), \tag{5}$$

where g is the horizontal input coefficient and h is the vertical input coefficient. The $out(0, j)$, for all $j < N$, and the $out(i, N-1)$, for all $i < N$, calculate the coefficient y_k according to Lemma 3.4, where $k = i + j$.

Proof: It is clear that the summed coefficient is found from the systolic index $(i+1, j-1)$ by maintaining $k = i+j$, while the current product is found by coefficient g_i and h_j. The output of the calculation out(i, j) is indicated by $i = 0$ or $j = N - 1$. Thus, by maintaining $k = i + j$, we have the following:

$$\text{out}(i, j) = g_i h_j + g_{i+1} h_{j-1} + \ldots$$
$$\text{out}(i, j) = \sum_{i=0}^{k} g_i h_{k-i}.$$

And, we have the theorem. □

Example 3.1: In Figure 6, we provide an example of how the 2D systolic array works for 3-degree polynomial multiplication. The diagram simulates a multiplication between $G(x) = 1 + 2x + x^2 + 3x^3$ and $H(x) = 2 + 5x + 3x^2 + x^3$ in 4 clock cycles. In this case, we have 4 coefficients for each polynomial, one polynomial assigned as the horizontal input while the other as the vertical input. Each blue cell shows the computation result of the corresponding cell, while the green cells show the multiplication result which is the polynomial $Y(x) = 2 + 9x + 15x^2 + 18x^3 + 20x^4 + 10x^5 + 3x^6$.

Each blue column shows the computation result depicted for different clock cycles. The computation propagates to the

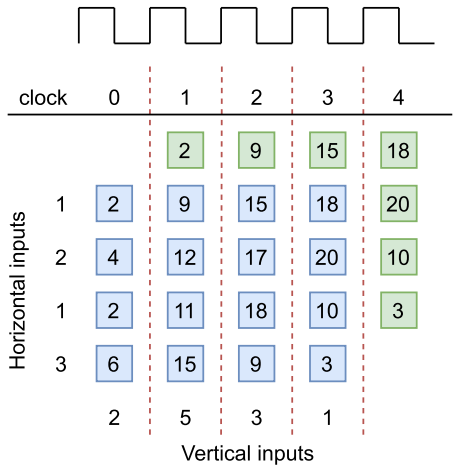


FIGURE 6. 2D systolic array for 3-degree polynomial multiplication.

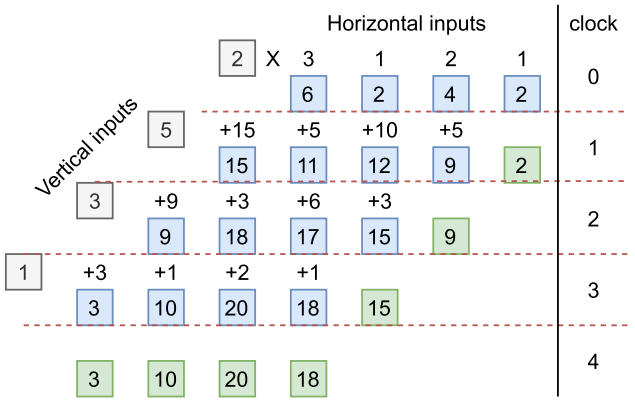


FIGURE 7. Stretched 2D systolic array to show convolution.

right as the clock increases. It produces one output each clock cycle with 3 additional outputs at the final clock cycle. Figure 7 shows the systolic array that has been rotated and stretched. In this figure, the systolic array is performing convolution between two polynomials. ■

D. POLYNOMIAL MODULO REDUCTION

The ring polynomial $R = Z[x]/(f(x))$ as described in the previous subsection is specified by the polynomial modulus $f(x) = x^n + 1$. In our design, we utilize a simple pattern occurring on polynomial reduction by the polynomial modulus. The following lemma is to show the reduction:

Theorem 3.2: The polynomial modulus $f(x) = x^n + 1$ of $Y(x)$ with polynomial degrees of $2n-1$ can be represented in $n-1$ multiplications as follows

$$Y(x)/f(x) = y_{n-1}x^{n-1} + \sum_{i=0}^{n-2} (y_i - y_{n+i})\, x^i, \tag{6}$$

where $Y(x)$ is the polynomial with coefficient y_i.

Proof: Suppose we have a $2n-2$ degree polynomial $Y(x)$ with polynomial degrees of $2n-1$, which is a multiplication result between two $n-1$ degrees polynomials.

$$Y(x) = \sum_{i=0}^{2n-2} y_i x^i \tag{7}$$

Now, we reduce the result by modulo the polynomial modulus. We subtract $Y(x)$ with $y_i x^{i-n}(x^n+1)$ for every i larger than $n-1$. Hence, the polynomial reduction can be written as follows.

$$Y(x)/f(x) = \sum_{i=0}^{2n-2} y_i x^i - \sum_{i=n}^{2n-2} y_i x^{i-n}\left(x^n+1\right) \tag{8}$$

From $i = 0$ to $i = n-2$, the coefficient y_i will be reduced by y_{n+i} and by rearranging (8), we have the theorem. □

Equation (6) shows we can perform polynomial reduction by simply subtracting the i-th coefficient with the $(n+i)$-th coefficient while ignoring the n-th coefficient.

After the polynomial reduction, we can reduce the coefficient by modulo q. In this case, we have $q = 2^d$, where d is a positive integer. We can simplify this by only taking d least significant bits (LSB) of the coefficient.

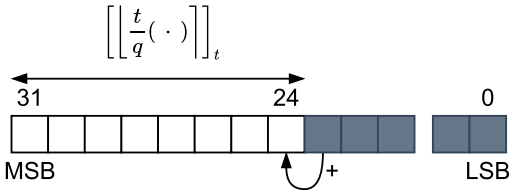


FIGURE 8. Extracted bit from $\left[\left\lfloor \frac{t}{q}(\cdot)\right\rceil\right]_t$ operation in the decryption operation in Lemma 2.3.

E. COEFFICIENT MODULO REDUCTION AND DIVISION

The encryption and decryption in BFV homomorphic encryption scheme require reduction by modulo q or t. This operation is denoted by $[\cdot]_q$ as explained in BFV scheme subsection. In our design, we use a power of 2 as q and t, thus the reduction can be easily performed by only taking a certain number of bits and ignoring the rest. For $[\cdot]_q$ and $[\cdot]_t$, we only care for 32-bit and 8-bit results, respectively.

As a consequence, q/t is a power of 2. Therefore, we have the following:

Lemma 3.5: The bit extraction for the coefficient modulo division

$$\left[\left\lfloor \frac{t}{q}(a)\right\rceil\right]_t$$

can be calculated by $[a] \gg log_2(q/t)$, where a is an integer, q is the ciphertext coefficient modulus, and t is the plaintext coefficient modulus.

Proof: As Definition 2.3 and Table 1, we have $t = 2^8$ and $q = 2^{32}$ thus $\frac{t}{q}(a)$ can be performed by $[a] \gg log_2(q/t) = [a] \gg 24$. □

In practice, we just take the 8-bits of the most significant bits as shown in Figure 8. The similar idea also was applied

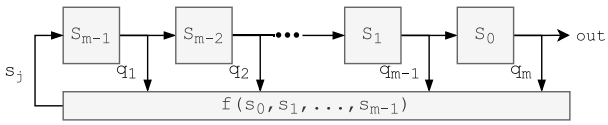


FIGURE 9. Fibonacci linear feedback shift registers.

in an existing work [9]. They use $t = 2$, while in our case, we use $t = 2^8$.

F. LINEAR FEEDBACK SHIFT REGISTERS

In the encryption operation, the ring polynomial e is a polynomial with all of its coefficients uniformly sampled from a discrete Gaussian distribution as indicated by $e \leftarrow \chi$. In order to improve the encryption time, we also design a stream Gaussian pseudo-random number generator (PRNG) that will be integrated with our accelerator. The Gaussian pseudo-random number generator (PRNG) was implemented using linear feedback shift registers (LFSR) and central limit theorem (CLT) as presented in [23].

In this subsection, we will provide a basic knowledge for linear feedback shift registers (LFSR) as described in [24]. LFSR is the most common method to generate pseudo-random numbers. It requires low resources to implement, yet it can produce large pseudo-random period sequences with good statistical properties and can be analyzed using algebraic techniques.

An LFSR of length m has m stages indexed from 0 to $m-1$ as shown in Figure 9, each stage is a register containing one bit value with an input and an output controlled by a clock.

In an LFSR in Figure 9, the following operations are performed at each clock cycle:

1) The content of stage 0 (s_0) forms the output sequence.
2) The content of stage i moves to stage $i-1$ for each i in $[1, m-1]$; and
3) The new content of stage $m-1$ is the feedback bit s_j which is modulo 2 the addition result of the previous contents where $q_{m-i} = 1$.

Lemma 3.6: Let the initial state of the LFSR is $[s_{m-1}, \ldots, s_1, s_0]$ *and* $s = s_0, s_1, s_2, \ldots, s_{m-1}$ *is the LFSR's output sequence, as a consequences, the following recursion will uniquely determined the output sequence s:*

$$s_j = \left(q_1 s_{j-1} + q_2 s_{j-2} + \ldots + q_m s_{j-m}\right) mod\ 2, \qquad (9)$$

where j is an integer and $j \geq m$. *The LFSR is usually denoted as* $\langle m, c(x)\rangle$, *where* $c(x) = 1 + q_1 x + q_2 x^2 + \ldots + q_m x^m \in \mathbb{Z}_2[x]$ *is the connection polynomial.*

Proof: As shown in Figure 9, it is clear that the output s_j is determined by the values of q_i based on the connection polynomial $c(x)$. □

It is important to acknowledge that s_j and $c(x)$ are different. s_j represents the recursion function for an output at a certain time, while $c(x)$ represents the hardware or physical connection in Figure 9.

The LFSR, that we show in Figure 9, is a Fibonacci LFSR. There also exists a different LFSR architecture called Galois LFSR which can generate the same output bit sequence s when the same connection polynomial $c(x)$ is used. Despite generating the same output bit stream, Galois LFSR has a different internal state sequence and theoretically have a lower critical delay path by only applying an XOR gate for an output s_i.

In this work, we implement a 20-bit LFSR with $c(x) = 1 + x^{17} + x^{20}$, thus the feedback connections only formed for q_{17} and q_{20}, which belong to stage s_0 and s_3. The maximum output sequence period is $2^m - 1$, and can be achieved if and only if the connection polynomial is a primitive polynomial [25]. In this case, the LFSR will always produce the maximum length sequence despite the non-zero initial state. The LFSR will simply cycle throughout all possible non-zero states.

G. CENTRAL LIMIT THEOREM

The LFSR explained in the previous subsection will be used to generate several uniformly distributed pseudo-random samples, to modify this into Gaussian or normally distributed numbers we use central limit theorem.

According to the central limit theorem (CLT) the mean μ of a random sample of size n drawn from independent random variables will approach a Gaussian distribution as the sample size increases, regardless of the original shape of the sample distribution [26]. The formal mathematics definitions and proof are presented in [27]. If the sample distribution is not terribly skewed, a general rule of thumb is having a sample size greater than 30.

Lemma 3.7: By using Lindeberg-Levy CLT [28], the following:

$$\frac{\sqrt{n}}{\sigma}(\tilde{X}_n - \mu),$$

converges to the normal distribution, where n is the number of inserted samples, μ *is the overall samples mean,* σ *is the standard deviation, and the average of inserted samples* $\tilde{X}_n$ *is calculated as follows*:

$$\tilde{X}_n = \frac{X_1 + \ldots + X_n}{n}.$$

where $X_1, \ldots, X_n$ *represented the samples.*

Proof: The lemma can be derived by using characteristic functions [29] and levy's continuity theorem [30]. □

According to the work in [31] that compares three conventional algorithms for generating Gaussian distributed random numbers, CLT is an extremely efficient method by simply sampling enough identical and independent uniform distributions. However, the Gaussian random numbers from CLT are lower quality compared to other conventional algorithms. Thus, in this work, we improve the quality of PRNG by applying rotation instead of only applying CLT in the LFSR.

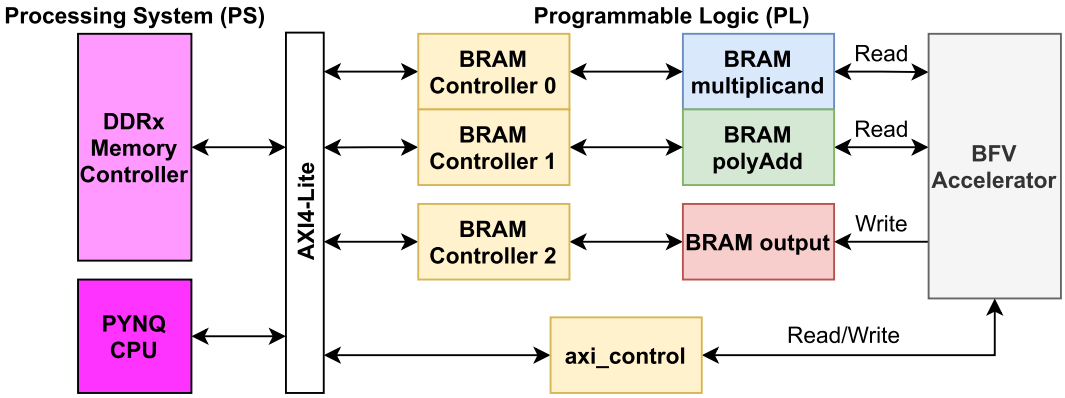


FIGURE 10. Processing system and programmable logic interface.

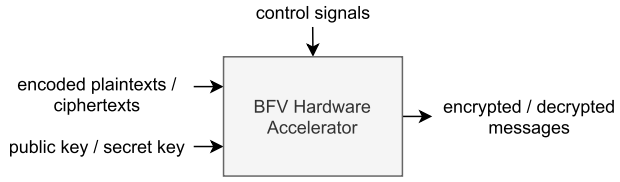


FIGURE 11. Hardware accelerator abstraction.

IV. PROPOSED HARDWARE ARCHITECTURE OF BFV HOMOMORPHIC ENCRYPTION ACCELERATOR USING INTEGRATED PRNG

This section explains the proposed hardware architecture of the BFV homomorphic encryption accelerator with integrated Gaussian pseudo-random number generator (PRNG) in details. First, we will explain about the accelerator architecture. After that, ring polynomial multiplication core, systolic array module and optimization, polynomial modulus reduction, Gaussian PRNG architecture, and control of the architecture will be presented.

A. PROPOSED ACCELERATOR ARCHITECTURE

The accelerator has several inputs namely plaintext or ciphertext, public key or secret key, and control signals. The accelerator will produce encrypted or decrypted messages depending on the operation mode. The control signal consists of start signal, number of inputs, reset signal, initial seed, and accelerator mode signal.

Figure 10 shows the diagram block of the processing system (PS) and the programmable logic (PL). The yellow colored modules are the interface between the accelerator and the processing system of PYNQ Z1 CPU. Our accelerator requires two input BRAMs to store public key and plaintext in encryption mode or ciphertexts in decryption mode. Moreover, it has one output BRAM to store ciphertexts in encryption mode, or plaintext in decryption mode. All the BRAMs store the R_q polynomials values. The R_2 polynomials have 3 possible values $(-1, 0, 1)$. These values will be connected directly from the axi control module to the accelerator.

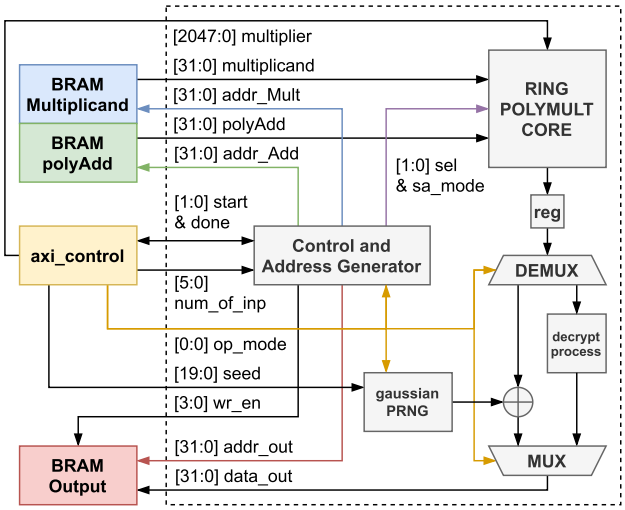


FIGURE 12. The proposed hardware accelerator architecture.

The axi control module is also responsible for sending the start signal and other control signals. The abstraction of our accelerator is shown in Figure 11.

The more detail architecture of the proposed accelerator is depicted in Figure 12. The polynomial inputs are stored in BRAM multiplicand and BRAM polyAdd, which can store multiple plaintexts or ciphertexts. The lowest degree coefficient is stored in the lowest address. The memory configuration for each mode can be found in the memory configuration section. The computation result is stored in the BRAM output. All read and write addresses are controlled by the accelerator's address generator when the start signal has been given by the axi control module.

The accelerator is constructed from several modules with the ring polymult core as the main module for calculating the ring polynomial multiplication and addition. The polymult core produces a stream output at constant period. This stream has two possible operations before being stored into the BRAM output depending on the accelerator mode. In

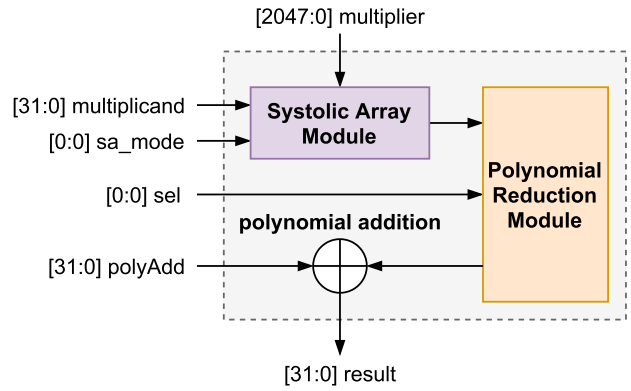


FIGURE 13. The ring polymult core.

encryption mode, the output stream is added with the Gaussian PRNG output. However, in decryption mode, the division and rounding are carried out in the coefficient modulo reduction and division as presented in Figure 8.

The control and address generator modules produce the address and control signals for the accelerator. These modules start producing periodic signals after the start signal has been given by the axi control module. After the computation is finished, the control module sends a done signal to indicate that the computation is finished for all plaintexts or ciphertexts. The outputs are stored in the memory (BRAMs).

B. RING POLYMULT CORE

Algorithm 1 explains the inner workings of the ring polymult core module as depicted in Figure 13. The stream multiplicand input will first enter the systolic array module, where the R_2 polynomial multiplier (*i.e.* 1024 2-bit weights) is already connected to the module. The multiplication output is then reduced inside the polynomial reduction module as described in Equation (6). The reduced polynomial will be added with the additional input stream (*i.e.* polyAdd) before exiting the accelerator core.

The polynomial reduction module causes the first 1024 outputs to wait for the higher degrees stream output, thus lowering the module throughput by almost half. This introduces 1024 clock cycles output latency to the polymult core module. The systolic array and the polynomial reduction module will be explained in detail on the next subsections.

C. SYSTOLIC ARRAY MODULE

The systolic array presented in the previous section has a two-dimensional array architecture that requires a lot of resources. Naively, to implement 1024-degree polynomial multiplication, we need a 1024×1024 systolic array, which is not practical. In this subsection, we present the reduced systolic array by using a systolic array with the number of arrays is $n = 1024$. The implementation is reduced to 1×1024 arrays equipped with processing element that has been optimized for multiplication between R_q and R_2 polynomials.

Algorithm 1: Ring Polymult Algorithm

Input: Multiplicand, $A(x) \in \mathbb{Z}_q[x]/(x^n + 1)$, where a_i is the i-th element of matrix A.
Input: PolyAdd, $B(x) \in \mathbb{Z}_q[x]/(x^n + 1)$, where b_i is the i-th element of matrix B.
Input: Multiplier, $C(x) \in \mathbb{Z}_q[x]/(x^2 + 1)$, where c_i is the i-th element of matrix C.
Output: $P(x) \in \mathbb{Z}_q[x]/(x^n + 1)$
begin
  $P = [p_0, p_1, \ldots, p_{n-1}] = [0, \ldots, 0]$;
  $Y = [y_0, y_1, \ldots, y_{2n-2}] = [0, \ldots, 0]$;
  for $i = 0$ *to* $n - 1$ **do**
    for $j = 0$ *to* $n - 1$ **do**
      $y_{i+j} \leftarrow y_{i+j} + a_j c_i$;
  for $i = 0$ *to* $n - 2$ **do**
    $p_i \leftarrow y_i - y_{n+i} + b_i$;
  $p_{n-1} \leftarrow y_{n-1} + b_{n-1}$;
  return P;

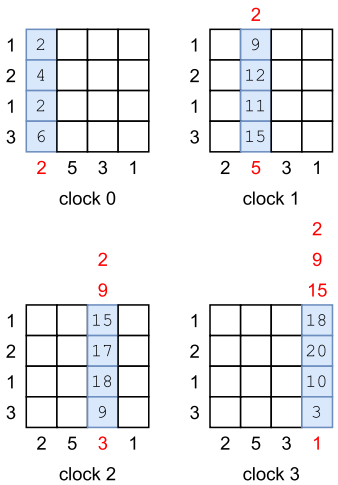


FIGURE 14. Polynomial multiplication from the moving input perspective.

The optimization can be obtained from the Algorithm 1. If we parallel the inner loop of the multiplication of $a_j c_i$, we have the following theorem:

Theorem 4.3: The run time of ring polynomial multiplication in Algorithm 1 can be reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ by paralleling the additions and the multiplications using n processing elements.

Proof: As shown in Algorithm 1, we can get the proof directly. By paralleling the n processing elements for additions and multiplications of $A \times c_i$, we can reduce the inner n-loop executions becomes only an execution. □

Figure 14 highlights the computation in Figure 6 that shows the propagating of input perspective. Notice that as the horizontal input propagates, the vertical input is changing while the horizontal input is fixed. If we ignore the column shifting,

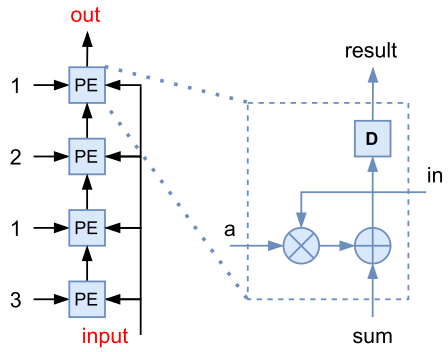


FIGURE 15. One dimensional systolic array for 3-degree polynomial multiplication in Figure 14.

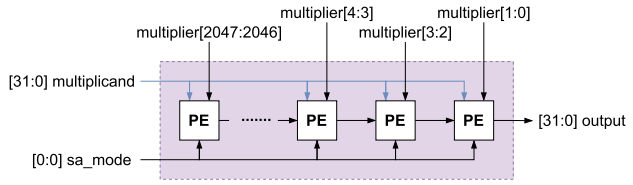


FIGURE 16. One dimensional (1D) systolic array.

the result is only forwarded to the cell above. Thus, the only modification needed for the systolic array is each cell output needed to be connected to the cell above it. After applying the modifications by changing the vertical input starting from the lowest degree coefficient and fixing the horizontal input, we can achieve the same multiplication result by using only one modified systolic array shown in Figure 15.

The same principle is scalable for larger systolic array sizes. Therefore, the systolic array in Figure 4 can be reduced into a one dimensional systolic array. Instead of diagonally connected output, the output is rerouted to the systolic cell above it. By reducing into a one dimensional array and rotating the reduced array, we obtained a design presented in Figure 16.

D. OPTIMIZED PROCESSING ELEMENT

The optimized version of the systolic cell presented in the previous section is shown in Figure 17. In order to significantly reduce the resources utilization, we replace the multiplier with two multiplexers, with each multiplexer basically performing an if-else operation. As discussed before, the R_2 polynomial $(-1, 0, 1)$ can be represented with a 2-bit number. This is why the multiplication operation can be substituted with only two if-else statements.

The mode signal will control the operation of the cell. The systolic cell generates multiplication results or forwards the output from the previous cell. The output table for the processing element is shown in Table 2. The table shows the cell output for every possible multiplier input and it's input mode. The value of multiplier input represents one of possible values in R_2, 00 represents 0. Any multiplication result with

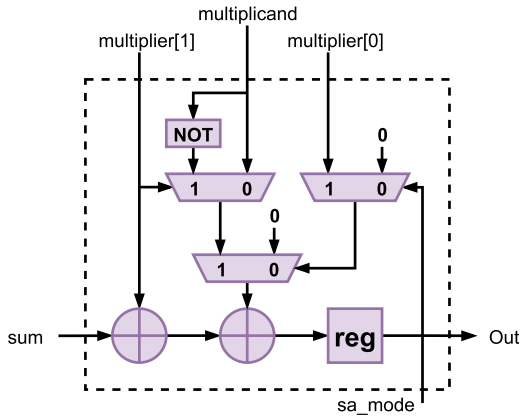


FIGURE 17. Optimized processing element for multiplication between R_q and R_2.

TABLE 2. Processing element's output table.

Input Mode	Multiplier Input	Output
0	XX	sum
1	00	sum+0
	01	sum + multiplicand
	10	sum + 1
	11	sum - multiplicand

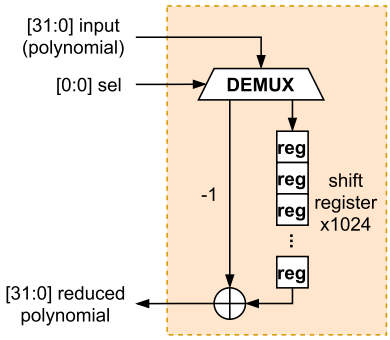


FIGURE 18. Hardware for polynomial reduction in Theorem 3.2.

0 is 0, thus the output is the sum input. 01 represents 1, hence the output is sum + multiplicand. 10 is not used as an input but it will produce sum + 1. Lastly, 11 represents −1, hence the output is sum − multiplicand.

E. POLYNOMIAL REDUCTION MODULE

The output of the systolic array module is the unreduced polynomial multiplication result between R_2 and R_q. To reduce the multiplication result, we simply subtract two output coefficients according to Theorem 3.2 presented previously. The systolic module generates an output stream starting from lowest coefficient, thus there is a time delay before we can start subtracting between two coefficients. To reduce the polynomial output, we design a polynomial reduction module shown in Figure 18.

The module has shift registers to delay the output and a subtractor to perform subtraction as described in Theorem 3.2.

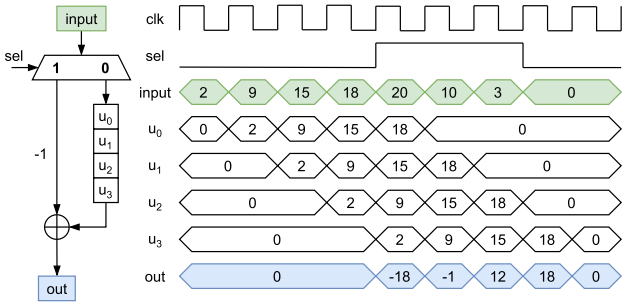


FIGURE 19. Timing diagram example for 3-degree polymod with the multiplication result from Figure 14.

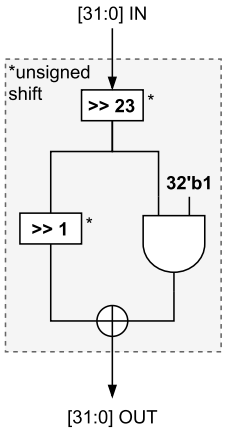


FIGURE 20. Hardware representation for polynomial modulus division (decryption process) in Lemma 3.5.

The first 1024 output will be stored inside the shift registers. In Figure 18, the datapath is determined by the sel signal. The left path is a direct bypass to the subtractor while the right one has shift registers for storing or delaying the output. In the first 1024 clock cycles, this module will not produce any output until the high degree coefficient is ready for reduction.

Figure 19 shows the reduction illustration for the polynomial multiplication result for 3-degree polynomials as shown before in Figure 6. The input is the output stream from the systolic array module. u_i represents the output of the stage i shift register. In Figure 19, only the first 4 inputs are stored into the shift registers. As the sel signal changes to 1, the subtraction process begins resulting the reduced polynomial modulus in the output signal. This module can be scaled to reduce larger degree polynomial multiplication results by adding more shift registers and adjusting the sel signal timing. The coefficient modulo q will transform the negative results into positive results by adding 2^q. The addition does not change the 32-bit binary result, thus this operation can be ignored.

F. POLYNOMIAL MODULUS DIVISION PROCESS

Polynomial modulus division is used for decryption process as described in Lemma 3.5. It involves division, rounding to the nearest integer, and coefficient modulo reduction by t. Our parameter selection enables the division to be performed by using right shifting, to round to the nearest integer we only need to add the 23^{rd} bit to the shifting result, then reduction by modulo t is simply taking the first 8-bit number from the result. This series of actions has been shown before in Figure 8 and the hardware representing this process is shown in Figure 20. The data will only enter this module when the accelerator is performing decryption.

G. GAUSSIAN PRNG

The encryption of the BFV scheme requires a discrete random Gaussian number to be added to the plaintext as a noise. We have tried using software to generate Gaussian random numbers, which increases the pre-processing time and the total encryption time. This is also worsened by the data transfer time from PS to PL that is required to store the pre-processed inputs. Therefore, it is necessary implement a low-cost Gaussian pseudo-random number generator using LFSR and CLT as presented in [23] to eliminate data transfer from PS to PL while the accelerator is running. The generator used in this work is presented in Figure 21.

In [23], we combine two existing methods to generate a stream of low variance Gaussian distributed pseudo-random numbers with longer pseudo-random periods compared to the existing works [32], [33]. The splitting internal state method is similar to the decimator presented in [32] and the rotation method is similar to the rotation presented in [33]. The split method is performed by split adder module as shown in Figure 21(b). This module will calculate the mean of two split numbers from Lemma 3.7 as follows:

$$\tilde{X}_n = \frac{X_1 + X_2}{2},$$

where $X_1 = X/2^N$, $X_2 = X \mod 2^N$, and X is the split adder input. The division by 2 is obtained from a right shift denoted by $\gg$. Moreover, we perform the following processes to randomize the samples as shown in Figure 21(a):

Lemma 4.8: The cyclic rotation method can be expressed as follows

$$\begin{aligned} Rot^{(k)}(s^{(t)}) &= Rot^{(k)}([s_{m-1+t}, \ldots, s_{1+t}, s_t]) \\ &= [s_{k-1+t}, s_{k-2+t}, \ldots, s_t, s_{m-1+t}, \ldots, s_{k+t}], \end{aligned}$$

where k is the number of cyclic rotation and $s^{(t)} = [s_{m-1+t}, \ldots, s_{1+t}, s_t]$ is the LFSR internal state.

Definition 4.7: The concat operator inside the generator joins two previous N-bit outputs into $2N$-bit numbers or simply

$$concat(a, b) = (2^N \times a) + b,$$

where a and b are N-bit outputs from previous stage.

Figure 21(a) shows the Gaussian PRGN used in this work. It utilizes rotation and split of a 20-bit LFSR internal state to generate 8 samples, then it applies CLT by simply adding and shifting by 2. The split adder module that is shown in Figure 21b, receives $2N$-bit input from the previous stage. After

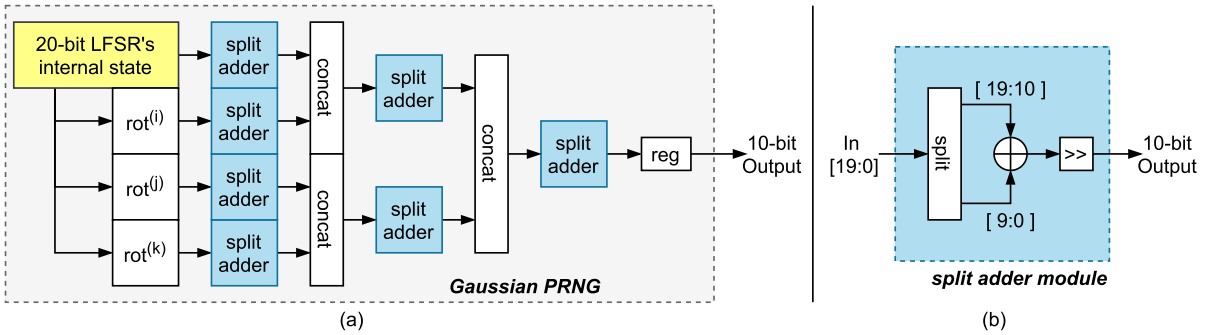


FIGURE 21. Gaussian pseudo-random number generator (PRNG) design.

that, it splits the input into two equal sizes and adds them together. The module was created to simplify the Verilog implementation and code reusability. The generator requires a 20-bit non-zero initial state and generates a sequence with a pseudo-random period of $2^{20} - 1$.

H. CONTROL AND ADDRESS GENERATOR

The accelerator can read and store the I/O from and to the memory. As explained before in previous sections, the timing of addresses and control signals is crucial to ensure the computation is correct. In this section, we present our control and address generator, which generates a set of periodic signals to meet all the timing requirements in the accelerator. We split the explanation into two parts. The first one is the equivalent hardware architecture for this module and the second one is the signal and address finite state machine (FSM).

1) CONTROL AND ADDRESS GENERATOR EQUIVALENT HARDWARE ARCHITECTURE

Figure 22 shows equivalent hardware architecture for the control and address generator. Only four main signals are generated from the finite state machine in Figure 23, namely 17-bit main address, 1-bit mode, 1-bit BRAM reset and 1-bit done signal, the rest of the signals are a delayed or shifted version of the main signals. The 16-bit address has a minimal bit size to access 128K memory.

The address signal is used to generate three BRAM addresses (*i.e.* two input BRAMs and an output BRAM). The address signal requires to be zero padded into 32-bit before being used for the BRAM address. The number of zero needed is 15 zeros for all BRAM addresses, except for the multiplicand BRAM in encryption mode. In encryption mode, the multiplicand BRAM only stores the public key, thus the accelerator only requires to access periodically from 0×0 to 0×2000 (8188 in decimal) with an increment of 0×4. A simple solution for this is only by using the first 13-bit of the main 17-bit address as the multiplicand address and zero padding the 13-bit address to 32-bit. Therefore, the range of output addresses is between 0 and 8191.

The unshifted main address will be used as the multiplicand BRAM address with the number of zero padding

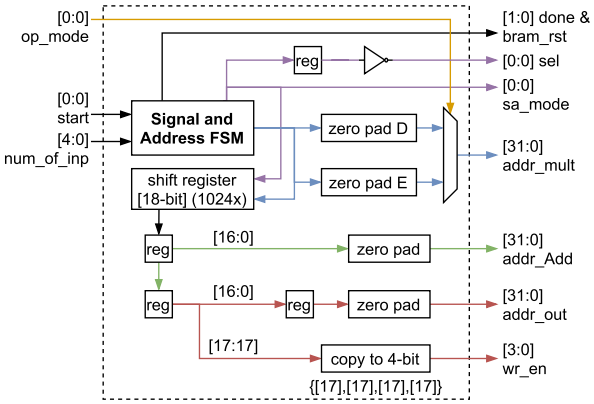


FIGURE 22. Control and address generator equivalent hardware architecture.

depending on the accelerator operation mode. In Figure 22, there is a selector to select which padded address will be used for the multiplicand BRAM address, where *E* stands for encryption and *D* stands for decryption. The polyAdd BRAM address requires to be shifted by 1025 clock cycles to adjust for the polynomial reduction module latency. On the other side, the output BRAM address needs to be shifted 1027 clock cycles. Two additional clock cycles were added to account for one clock cycle memory read delay and one clock cycle from the polymult core output register between the core output and the demultiplexer.

The mode signal is used to generate systolic array mode signal, selector signal inside the polynomial reduction module, and output BRAM's write enable signal. The sel signal is a one clock cycle delayed inverted mode signal and the write enable signal is 1025 clock cycles delayed mode signal. The write signal requires to be copied into a 4-bit signal where each bit is responsible for each Byte in the memory.

2) CONTROL AND ADDRESS GENERATOR FINITE STATE MACHINE (FSM)

The FSM in Figure 23 has four states and is responsible for generating the main signals. The first state is idle state. At this state, the accelerator is not working and waiting for the start

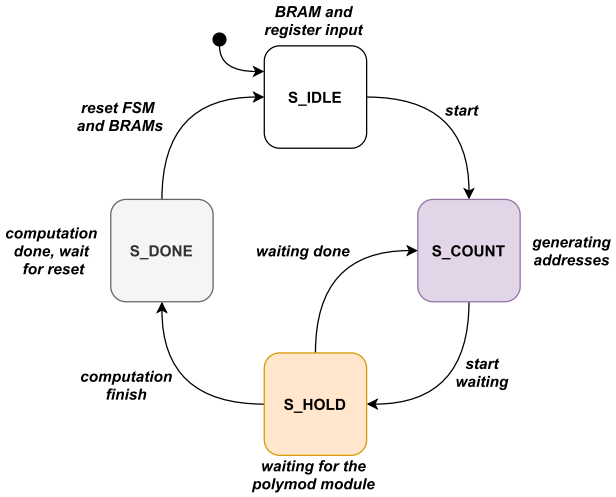


FIGURE 23. Control and address generator finite state machine (FSM).

31:29	28	27:8	7	6	5	4:0
*not used	done	initial seed	reset	start	mode	number of input

FIGURE 24. Control register bitmap.

signal. The input memories also can be written before the start signal is given. As the start signal changes to 1, the current state changes to count state. In this state, the address generator will generate addresses for the first 1024 coefficients, thus it generates an input stream for the systolic array module.

After generating addresses for the first 1024 coefficients, the FSM current state changes to hold state and waits for the polynomial reduction process to finish. Once the polynomial reduction is finished, the FSM current state will return to the count state to generate addresses for the next 1024 coefficients. The cycle of counting and waiting will repeat for the number of inputs available in the BRAM.

When the counting and the waiting cycles completed, the FSM state moves to the done state and triggers the finish flag or done signal. For the next encryption or decryption, the PS must send the reset signal to reset the FSM back to the idle state and empty the BRAMs before reusing the accelerator.

I. CONTROL REGISTER

The control register is the first register inside the axi control module. The axi control module stores control signals and the R_2 polynomial for multiplication weight. The control register is a 32-bit register storing control signals such as start signal, number of inputs, operation mode (*i.e.* encryption or decryption), reset signal, LFSR's initial seed, and done signal. The bitmap for the control register is shown in Figure 24.

The detail for each field in the control register is explained as follows:

- **Number of input:** Number of data stored inside the input memory. For decryption mode, the number of input is equal to the number of ciphertext inside the memory. However, for encryption mode, the number of input doubles for each plaintext.
- **Start signal:** The rising edge of start signal will trigger the FSM to start the accelerator.
- **Operation mode:** Choosing between encryption (0) or decryption (1) mode.
- **Reset:** To reset the accelerator and the memory modules.
- **Initial seed:** LFSR's initial state. Any non-zero 20-bit number can be used.
- **Done:** Only turns into 1 when the encryption or decryption has finished for every input.

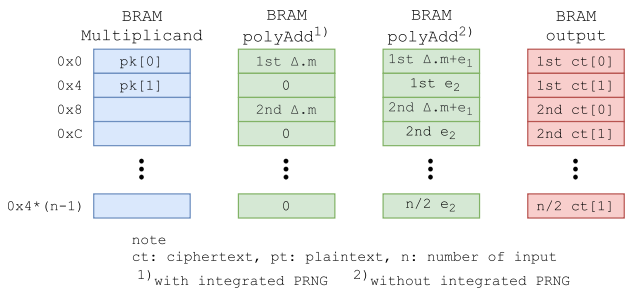


FIGURE 25. BRAM configuration for encryption mode.

V. MEMORY (BRAM) CONFIGURATION AND DATAFLOW

The proposed accelerator works by reading the input that is already stored inside the input memory and then storing the output inside the output memory. All memories have the same sizes. However, they are used and arranged differently depending on the operation modes. The datapath also has a slight difference in each mode. In this section, we provide the memory configuration and datapath for both encryption and decryption modes.

A. ENCRYPTION CONFIGURATION

Figure 25 shows the memory configuration for encryption mode. The multiplicand memory stores the public keys starting from $pk[0]$ and then followed by $pk[1]$. In encryption mode, the rest of the memory is empty. The address generator will generate only 13-bit addresses, thus only the first 2048 32-bit data will be cycled. The polyAdd memory stores the scaled encoded message and the Gaussian noise. Finally, the output memory stores ciphertexts as shown in Figure 25.

Figure 26 shows dataflow for encryption mode. The red arrows indicate path the data will take in encryption mode. Each I/O arrow is also labeled with the input or output name (*e.g.*, public key, plaintext, etc). In encryption mode, the data will take the left path and will be added with Gaussian noise generated from the Gaussian PRNG module.

B. DECRYPTION CONFIGURATION

Figure 27 shows memory configuration for decryption mode. The multiplicand memory stores the ciphertext element 1. The polyAdd memory stores the ciphertext element 0, and

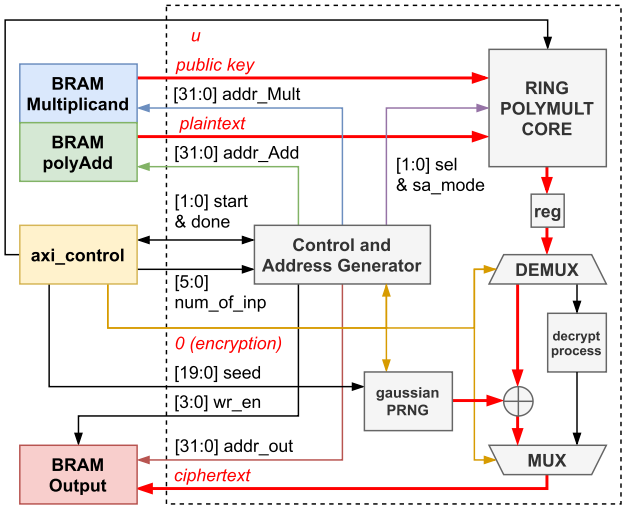


FIGURE 26. Datapath in encryption mode.

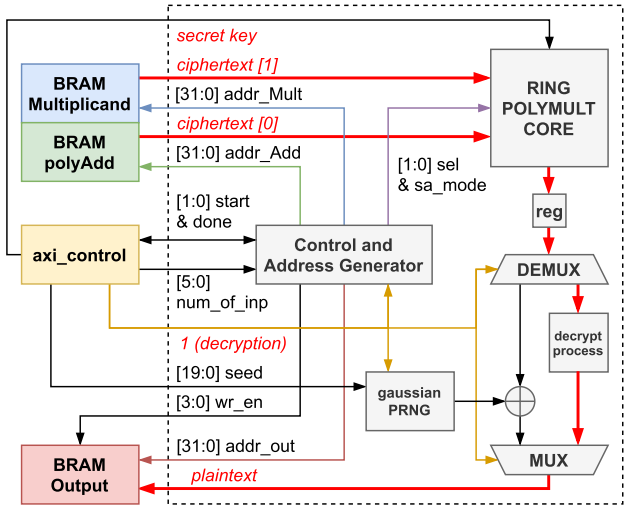


FIGURE 28. Datapath in decryption mode.

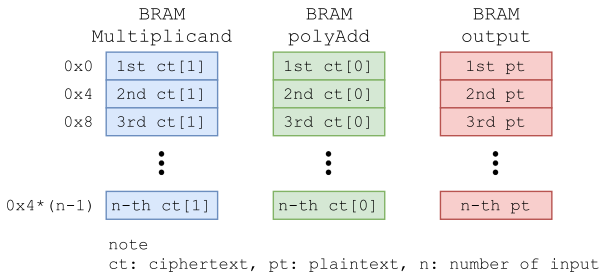


FIGURE 27. BRAM configuration for decryption mode.

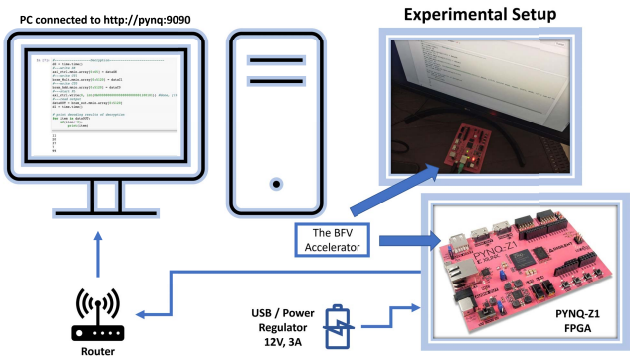


FIGURE 29. Experimental Setup.

the output memory stores encoded plaintexts. In this configuration, all memories use the same sizes.

Figure 28 shows dataflow for decryption mode. The red arrows indicate path the data will take in decryption mode. Each I/O arrow is also labeled with the input or output name (*e.g.*, secret key, ciphertext, etc.). In decryption mode, the data will take the right path and will enter the decryption process until it produces the plaintext.

VI. IMPLEMENTATION RESULTS AND COMPARISONS

We implemented our proposed accelerator on a low-cost xilinx's PYNQ Z1 FPGA (price $199, in 2021). The available resources are 53,200 LUTs, 106,400 flip-flops, and 630KB memory (BRAM). We use xilinx's Vivado IDE to synthesize our project. Our project consists of multiple Verilog modules that each module represents as described in the hardware architecture section previously. As a consequence of using xilinx's PYNQ environment, the processing system was implemented using embedded Python Jupyter Notebook IDE.

Figure 29 shows our experimental setup. We can use a 5V USB connection to our computer or a 12V and 3A voltage regulator to power up the FPGA PYNQ-Z1. The FPGA and the PC is connected to the same router. In this case, we use a PC with an Intel Core i7 processor with 8GB memory. The PS of the FPGA can be accessed by connecting to http://pynq:9090/ via browser.

A. BUILT-IN GAUSSIAN PRNG RESULT

The built-in generator was implemented and simulated using Verilog HDL. The simulation generates a pseudo-random number period or 1,048,575 data. Next, we plot a histogram of the result in Figure 30. Clearly, the histogram in Figure 30 resembles normal distribution, to show this statistically we plotted a quantile-quantile (QQ) plot of our result compared to a normal distribution plot. The QQ plot is shown in Figure 31. The blue marks represent our result and the green line is the normal distributed numbers. Majority of the blue marks match the green line which means our result fit the normal distribution.

To observe randomness, we perform auto-correlation test for $k = 1$ to $k = 30$. Figure 32 shows the performance of randomness by auto-correlation. In this case, since the *corr* values are small, we use auto-correlation factor $10 \log |corr|$ to measure the correlation between two random variables.

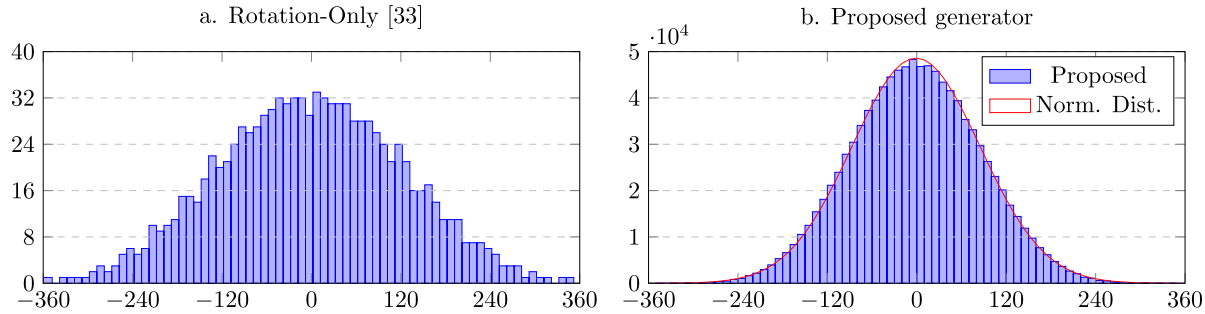


FIGURE 30. Comparison of PRNG output distribution.

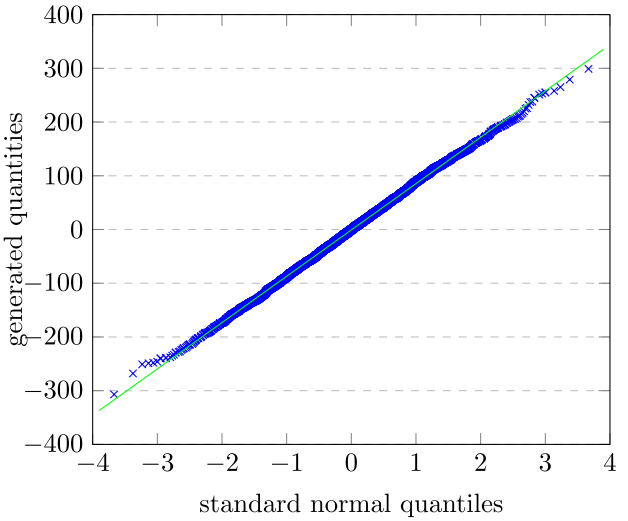


FIGURE 31. Generator output quantile-quantile plot compares to normal distribution.

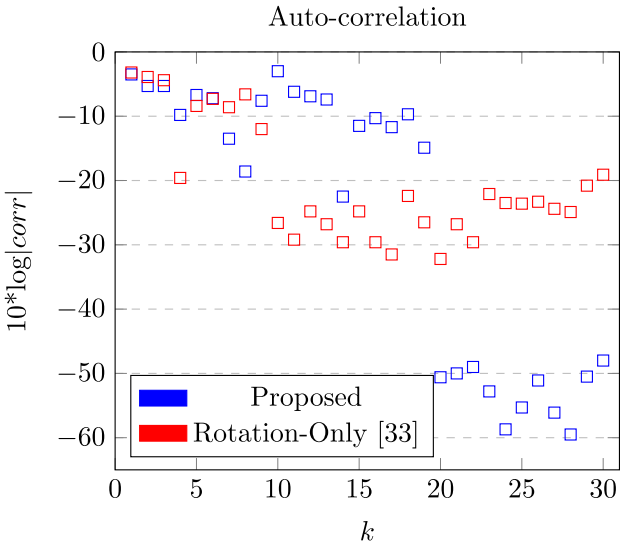


FIGURE 32. Auto-correlation comparison.

Moreover, the correlation *corr* is calculated by [34]:

$$corr(X, Y) = \frac{E((X - \mu_x)(Y - \mu_y))}{\sigma_x \sigma_y},$$

where E is expected value, X and Y are the random variables, and σ_x and σ_y are the standard deviations of X and Y, respectively.

For $k < 10$, our proposed PRNG has similar correlation with the rotation-only design. However, for $10 < k < 20$, auto-correlation of the rotation-only has better performance. Finally, our proposed Gaussian PRNG outperforms $2\times$ (around -30 vs. around -60 of auto-correlation factor) better randomness factor from that of the rotation-only [33].

For the implementation, the proposed PRNG requires a 20-bit LFSR with an XOR gate. The extension for 60-bit LFSR is also provided in Table 3. The number of XOR gates depends on the connection polynomial. The more coefficient used in the connection polynomial, the more XOR gates are required. The more XOR gates will increasing delay and critical path area. The problem is solved by implementing the Galois LFSR in the architecture. In fact, the Galois LFSR generates the same bit pattern in certain period. However, the different internal state by applying bit splitting and bit rotation also change the pseudo-random in the Galois LFRS.

TABLE 3. FPGA resource utilization and performance comparison for Gaussian PRNG using LFSR.

Work	LUTs	FFs	Freq. (MHz)	#Bits	Area/ #Bits	Clock Cycle
LFSR113 [35]	79	162	443	32	60.3 (1.4×)	2
Taus88 [36]	68	130	448	32	49.5 (1.1×)	2
LFSR258 [35]	171	386	396	64	69.6 (1.6×)	2
LUTSR [37]	64	64	609	32	32.0 (0.7×)	2
Rotation [33]	40	11	568	20	20.4 (0.5×)	1
Our's	81	41	292	20	48.8 (1.1×)	1
Our's	241	91	179	60	44.3 (1×)	1

Table 3 shows the resource utilization and performance comparison for some PRNGs using linear LFSR methods. The area is defined as [38]:

$$(\text{Number of LUTs} + \text{Number of FFs}) \times 8.$$

The clock cycle indicates the number of clocks to generate a random number in certain bits and the #Bits indicates the size of the LFSR in bits. Some implementations of 32 and

TABLE 4. NIST test for proposed Gaussian PRNG.

No.	Test Type	P-Value	Result
1	Frequency (Monobit)	0.717	Passed
2	Frequency Test within Block	0.404	Passed
3	Run	~ 0	Not Passed
4	Longest Run of Ones in a Block	~ 0	Not Passed
5	Binary Matrix Rank	0.694	Passed
6	DFT (Spectral)	~ 0	Not Passed
7	Non-Overlap. Template Match.	~ 0	Not Passed
8	Overlap. Template Match.	0.296	Passed
9	Maurer's Universal Statistical	0.000	Not Passed
10	Linear Complexity	0.809	Passed
11	Serial	0.000	Not Passed
12	Approximate Entropy	0.797	Passed
13	Cumulative Sums (Forward)	0.634	Passed
14	Cumulative Sums (Reverse)	0.774	Passed
15	Random Excursions	0.673	Passed
16	Random Excursions Variant	0.658	Passed

64 bits of extended period of LFSR are shown in [35], [36]. FPGA implementation of an LUT-based LFSR is presented in [37]. The rotation-only has a smallest area because it does not require additional XORs and registers to perform the method [33]. As shown in Table 3, compared to the other LFSRs, the proposed Gaussian PRNGs (20-bit and 60-bit LFSRs) have relatively smaller area ratios up to 1.6× and have a faster clock cycle to generate a random number that can output a random number in a clock.

Moreover, we tested the proposed Gaussian PRNG using the NIST test suite [39] as shown in Table 4. First, we generated 1 million random bits using the proposed PRNG as an input of the NIST test suite. The P-values, which indicates how likely it is that the data could have occurred under the null hypothesis, is then measured in the range from 0 to 1 [39]. The proposed Gaussian PRNG passed 10 of the NIST tests. There are a few scenarios which are not passed, due to the proposed Gaussian PRNG uses only linear method *i.e.,* LFSR that is combined by rotation and split.

Some previous PRNG methods also use nonlinear circuits or algorithms to increase the degree of randomness such as chaotic PRNGs. For instance, the works in [40] and [41] use analog approaches for generating chaotic maps. However, using a special analog circuit for a system-on-chip in FPGA implementation is expensive.

Although nonlinear, such as chaotic, PRNGs can achieve higher degree of randomness, they suffer from additional latency due to much higher number of iterations, compared to the linear PRNGs. Moreover, nonlinear PRNGs require special circuits or higher complexity algorithms such as chaotic logistic map, negotiation, permutation, reseeding, and mixing algorithms as described in [38], [40]–[43].

Since our aim is to achieve a satisfactory degree of randomness with the limitation of targeted FPGA resources, we focus on the linear type of efficient Gaussian PRNG. The main objective of the proposed work is to have a high-throughput Gaussian PRNG that can output a random number in a clock. Thus, the Gaussian noise on the BFV, as explained in Section III-F can be implemented efficiently.

B. PROPOSED BFV ACCELERATOR EXPERIMENTAL RESULTS AND COMPARISONS

In our design, we are dealing with 1023-degree polynomials which have 1024 coefficients in each polynomial. Thus, it requires 1024 × 32 × 3 bit memory for each ciphertext. Our FPGA board has 630KB memory size. This means we can perform decryption up to 51 ciphertexts, with each BRAM having the size of 204KB. The closest block memory size available is 128KB or 256KB, for this implementation we limit our use to 128KB per block memory. As a consequence, we can only store a maximum of 32 ciphertexts.

First, we test our accelerator for both encryption and decryption and compare to well-known homomorphic encryption library project called Microsoft SEAL [44] on Intel Core i7 CPU with 8GB memory. Tabel 5 shows the performance of our accelerator compared to the BFV on Microsoft SEAL (with newest update ver 3.7 on Sept. 2021). As shown in the table, the Microsoft SEAL's BFV has average time executions; 371.85 μs for encryption, 73.27 μs for decryption and 419.83 μs for the both encryption and decryption for the same parameters in Table 1. We record the time execution in the accelerator and it turns out that our accelerator accelerates up to 9× for encryption, 3.5× for decryption, and 6.8× for both than to that of the Microsoft SEAL's BFV [44]. This is to show that by the proposed design, the hardware implementation of BFV accelerator is very promising for embedded platform in the form of FPGA or ASIC implementation in the future.

Table 6 shows the comparison to the other previous works with similar schemes in FPGAs. The table shows the polynomial multiplication method, the platform (FPGA), the BFV parameters, the resources utilization, the clock speed, and the polynomial multiplication latency. We compare with similar algorithms such as iterative NTT, Karatsuba, FFT-based, and somewhat-homomorphic encryption (SHE). We do not compare our design to the parallel implementation of NTT algorithm such as in [15], because our proposed accelerator is targeted for a low-cost FPGA. It is possible to further parallel our proposed design for the future research, because it has iterative pattern in the polynomial calculation as shown in Algorithm 1. However, the parallel implementation will require much larger FPGA that consume more areas of utilization.

The polynomial multiplication time is the time of the accelerator doing a polynomial multiplication operation inside the accelerator. In [9], Karatsuba algorithm is used to calculate homomorphic multiplication. The rest of the work presented in Table 6 implements various similar methods implementing the polynomial multiplication such as iterative NTT-based polynomial multiplication [13],

TABLE 5. Running time comparison to Microsoft SEAL [17], [44].

Project	Platform	Encryption (μs)	Decryption (μs)	Both (μs)
Microsoft SEAL's BFV [17], [44]	CPU i7	371.85	73.27	419.83
Our's	FPGA PYNQ-Z1	40.97 (9×)	20.50 (3.5×)	61.47 (6.8×)

TABLE 6. Resource and polynomial multiplication comparison to other methods in FPGA.

Ref.	Pol. Mul. Method	Platform (FPGA)	Price	n	q	LUT / DSP / BRAM	# DSP to LUT [45]	Total LUT	Clock (MHz)	Pol. Mul. Lat. (μs)	Clock Cycle
[13]	Iter. NTT	VIRTEX-6	$3494	65536	30-bit	72K/250/106	10667	82.6K	100	3376	337.6K
[9]	Karatsuba	STRATIX-6	$6995	2560	125-bit	30K/100/-	4267	34.2K	331	583	192.0K
[14]	Iter. NTT	UltraScale	$2495	4096	30-bit	64K/200/400	8534	72.5K	225	171	38.5K
[46]	FFT-based	SPARTAN-6	$695	**1024**	30-bit	1644/1/6.5	43	1.6K	200	110	**22.0K(22×)**
[47]	SHE	SPARTAN-6	$695	**1024**	31-bit	6689/4/8	171	6.8K	241	33	**7.9K(7.9×)**
[16]	Iter. NTT	SPARTAN-6	$695	**1024**	32-bit	1208/14/14	598	1.7K	212	37	**7.8K(7.8×)**
Our's	Conv.	PYNQ Z1	$199	**1024**	32-bit	37K/ - /96	0	37K	100	10.24	**1.0K(1×)**

[14], [16], FFT-based polynomial multiplication [46], and somewhat-homomorphic encryption (SHE) [47]. As shown in Table 6, for $n = 1024$, our proposed accelerator outperforms the polynomial multiplication clock cycle up to 22× compared to the other methods. This means that the proposed design can finish the multiplication with smaller clock cycle and suitable for ASIC implementation. For future work, the design can be implemented in a larger FPGA by implementing more optimizations and parallelizations.

VII. CONCLUSION

In this paper, we presented comprehensive design and implementation of a hardware architecture to accelerate encryption and decryption in BFV scheme. Our accelerator used convolution approach with for calculating a polynomial multiplication. To implement the convolution, we used a systolic array to calculate polynomial convolution followed by a simple delayed subtraction to calculate polynomial modulo reduction inside our accelerator's core. Moreover, we used a built-in Gaussian pseudo-random number generator (PRNG) to generate Gaussian noise in the encryption operations. Finally, we implemented the 1024 degrees BFV accelerator on the Xilinx PYNQ Z1 board and compared the encryption and decryption performances to other methods as well as a software implementation on Intel Core i7 with 8GB memory. Experimental results showed that our accelerator outperforms the clock cycles of other methods for the polynomial multiplication operation with degrees 1024 up to 22×. Moreover, our proposed PRNG has better 2× correlation compared to the rotation-only-based PRNG. Furthermore, our accelerator accelerates up to 9× for encryption and 3.5× for decryption as well as 6.8× for overall compared to Microsoft SEAL on Intel Core i7 processor with 8GB memory. The proposed design is scalable for higher degrees polynomial multiplication and useful for security technology such as high-speed secure cloud computing, blind computing, and secure communication.

REFERENCES

[1] (2020). *IBM Top 7 Most Common Uses of Cloud Computing*. Accessed: Aug. 2, 2021. [Online]. Available: https://www.ibm.com/cloud/blog/top-7-most-common-uses-of-cloud-computing

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Symp. Theory Comput. (STOC)*, 2009, pp. 169–178.

[3] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2010, pp. 24–43.

[4] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-LWE and security for key dependent messages," in *Proc. Annu. Cryptol. Conf.*, 2011, pp. 505–524.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, pp. 1–36, Jul. 2014.

[6] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme," in *Proc. IMA Int. Conf. Cryptogr. Coding*, 2013, pp. 45–64.

[7] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," IACR Cryptol. ePrint Arch., Tech. Rep. 2021/144, 2012, p. 144. [Online]. Available: https://ia.cr/2012/144

[8] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2017, pp. 409–437.

[9] V. Migliore, M. M. Real, V. Lapotre, A. Tisserand, C. Fontaine, and G. Gogniat, "Hardware/software co-design of an accelerator for FV homomorphic encryption scheme using Karatsuba algorithm," *IEEE Trans. Comput.*, vol. 67, no. 3, pp. 335–347, Mar. 2018.

[10] A. Karatsuba and Y. Ofman, "Multiplication of multidigit numbers on automata," *Sov. Phys. Doklady*, vol. 7, no. 7, pp. 595–596, Jan. 1963.

[11] M. Albrecht, S. Bai, and L. Ducas, "A subfield lattice attack on overstretched NTRU assumptions," in *Proc. Annu. Int. Cryptol. Conf.*, 2016, pp. 153–178.

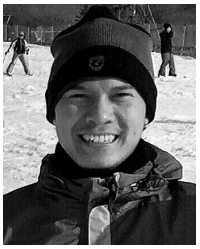
[12] D. Stehle and R. Steinfeld, "Making NTRUEncrypt and NTRUSign as secure as standard worst-case problems over ideal lattices," Cryptol. ePrint Arch., Tech. Rep. 2013/004, 2013. [Online]. Available: https://ia.cr/2013/004

[13] S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018.

[14] S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 387–398.

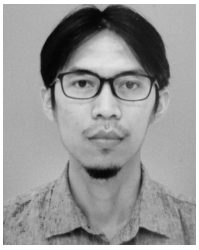
[15] A. C. Mert, E. Ozturk, and E. Savas, "Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 2, pp. 353–362, Feb. 2020.

[16] A. C. Mert, E. Ozturk, and E. Savas, "Design and implementation of a fast and scalable NTT-based polynomial multiplier architecture," in *Proc. 22nd Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2019, pp. 253–260.
[17] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library–SEAL v2.1," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.*, 2017, pp. 3–18.
[18] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
[19] E. W. Weisstein. *'Ring' Mathworld–A Wolfram Web Resource*. Accessed: Jul. 9, 2021. [Online]. Available: https://mathworld.wolfram.com/Ring.html
[20] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, 2010, pp. 1–23.
[21] H. J. Nussbaumer, "Elements of number theory and polynomial algebra," in *Fast Fourier Transform and Convolution Algorithms*. Berlin, Germany: Springer-Verlag, 1982, pp. 4–31.
[22] N. Sutisna, G. Jonatan, I. Syafalni, R. Mulyawan, and T. Adiono, "Polynomial multiplication systolic array for homomorphic encryption in secure network communications," in *Proc. IEEE Int. Conf. Commun., Netw. Satell. (Comnetsat)*, Dec. 2020, pp. 390–394.
[23] G. Jonatan, I. Syafalni, N. Sutisna, R. Mulyawan, and T. Adiono, "Gaussian pseudo-random number generator using LFSR's rotation and split," in *Proc. Int. Symp. Electron. Smart Devices (ISESD)*, Jun. 2021, pp. 1–5.
[24] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, 2018.
[25] S. W. Golomb, *Shift Register Sequences: Secure and Limited-Access Code Generators, Efficiency Code Generators, Prescribed Property Generators, Mathematical Models*. Singapore: World Scientific, 2017.
[26] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability and Statistics for Engineers and Scientists*, vol. 5. New York, NY, USA: Macmillan 1993.
[27] E. W. Weisstein. *Central Limit Theorem*. Mathworld–A Wolfram Web Resource. Accessed: May 28, 2021. [Online]. Available: https://mathworld.wolfram.com/CentralLimitTheorem.html
[28] S. D. Chatterji, "Lindeberg's central limit theorem à la Hausdorff," *Expo. Math.*, vol. 25, no. 3, pp. 215–233, 2007.
[29] D. S. Lemons, *An Introduction to Stochastic Process in Physics*. Baltimore, MD, USA: Johns Hopkins Univ. Press, 2002.
[30] D. Williams, *Probability With Martingales*. Cambridge, U.K.: Cambridge Univ. Press, 1991.
[31] Y. Hu, Y. Wu, Y. Chen, G. C. Wan, and M. S. Tong, "Gaussian random number generator: Implemented in FPGA for quantum key distribution," *Int. J. Numer. Model., Electron. Netw., Devices Fields*, vol. 32, no. 3, p. e2554, May 2019.
[32] M. Kang, "FPGA implementation of Gaussian-distributed pseudo-random number generator," in *Proc. 6th Int. Conf. Digit. Content, Multimedia Technol. Appl.*, Aug. 2010, pp. 11–13.
[33] G. Cotrina, A. Peinado, and A. Ortiz, "Gaussian pseudorandom number generator based on cyclic rotations of linear feedback shift registers," *Sensors*, vol. 20, no. 7, p. 2103, Apr. 2020.
[34] D. C. Howell, *Statistical Methods for Psychology*, 7th, ed. Wadsworth, OH, USA: Cengage Learning, 2010.
[35] P. L'Ecuyer, "Tables of maximally equidistributed combined LFSR generators," *Math. Comput.*, vol. 68, no. 225, pp. 261–269, 1999.
[36] P. L'Ecuyer, "Maximally equidistributed combined tausworthe generators," *Math. Comput.*, vol. 65, no. 213, pp. 203–213, 1996.
[37] D. B. Thomas and W. Luk, "The LUT-SR family of uniform random number generators for FPGA architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 4, pp. 761–770, Apr. 2013.
[38] M. Bakiri, J.-F. Couchot, and C. Guyeux, "CIPRNG: A VLSI family of chaotic iterations post-processings for F_2-linear pseudorandom number generation based on Zynq MPSoC," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 5, pp. 1628–1641, May 2018.
[39] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," NIST Special Publication, Gaithersburg, MD, USA, Tech. Rep. 800-22, 2010.
[40] P. S. Paul, M. Sadia, and M. S. Hasan, "Design of a dynamic parameter-controlled chaotic-PRNG in a 65 nm CMOS process," in *Proc. IEEE 14th Dallas Circuits Syst. Conf. (DCAS)*, Nov. 2020, pp. 1–4.
[41] C.-Y. Li, Y.-H. Chen, T.-Y. Chang, L.-Y. Deng, and K. To, "Period extension and randomness enhancement using high-throughput reseeding-mixing PRNG," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 2, pp. 385–389, Feb. 2012.
[42] M. Bakiri, C. Guyeux, J.-F. Couchot, L. Marangio, and S. Galatolo, "A hardware and secure pseudorandom generator for constrained devices," *IEEE Trans. Ind. Informat.*, vol. 14, no. 8, pp. 3754–3765, Aug. 2018.
[43] J. Černák, "Digital generators of chaos," *Phys. Lett. A*, vol. 214, nos. 3–4, pp. 151–160, 1996.
[44] (Sep. 2021). *Microsoft SEAL (Release 3.7)*. Redmond, WA, USA. [Online]. Available: https://github.com/Microsoft/SEAL
[45] X. Li, A. Jain, D. Maskell, and S. A. Fahmy, "An area-efficient FPGA overlay using DSP block based time-multiplexed functional units," 2016, *arXiv:1606.06460*.
[46] T. Pöppelmann and T. Güneysu, "Towards efficient arithmetic for lattice-based cryptography on reconfigurable hardware," in *Proc. Int. Conf. Cryptol. Inf. Secur. Latin Amer.*, 2012, pp. 139–158.
[47] D. D. Chen, N. Mentens, F. Vercauteren, S. S. Roy, R. C. C. Cheung, D. Pao, and I. Verbauwhede, "High-speed polynomial multiplication architecture for ring-LWE and SHE cryptosystems," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 62, no. 1, pp. 157–166, Jan. 2015.



INFALL SYAFALNI (Member, IEEE) received the B.Eng. degree in electrical engineering from the Institut Teknologi Bandung (ITB), Bandung, Indonesia, in 2008, the M.Sc. degree in electronic engineering from the University of Science Malaysia (USM), Penang, Malaysia, in 2011, and the Dr. (Eng.) degree in engineering from the Kyushu Institute of Technology (KIT), Iizuka, Fukuoka, Japan, in 2014. From 2014 to 2015, he has held a research position with KIT. From 2015 to 2018, he has held an ASIC Engineer with the ASIC Development Group, Logic Research Company Ltd., Fukuoka. In 2019, he joined the ITB, where he is currently an Assistant Professor with the School of Electrical Engineering and Informatics and a Researcher with the University Center of Excellence on Microelectronics, ITB. His current research interests include logic synthesis, logic design, VLSI design, and efficient circuits and algorithms.

GILBERT JONATAN received the B.Eng. degree *(cum laude)* in electrical engineering from the Institut Teknologi Bandung (ITB), Indonesia, in 2020. He is currently pursuing the M.S. degree with the Graduate School of Electrical Engineering, Korean Advanced Institute of Science and Technology (KAIST), South Korea. From 2020 to 2021, he worked as a Researcher with the University Center of Excellence on Microelectronics, ITB. His current research interests include computer architecture, hardware accelerator, homomorphic encryption, and artificial intelligence.



NANA SUTISNA (Member, IEEE) received the B.Eng. degree in electrical engineering and the M.Eng. degree in microelectronics from the Bandung Institute of Technology, Indonesia, in 2005 and 2011, respectively, and the Ph.D. degree in computer science and electronics from the Kyushu Institute of Technology, in 2017. From 2017 to 2020, he was a Postdoctoral Fellow with the Department of Computer Science and System Engineering, Kyushu Institute of Technology. He is currently with the Institut Teknologi Bandung as a Lecturer. His research interests include VLSI design, baseband wireless system design, AI processor design, and HW/SW co-design and co-verification.

RAHMAT MULYAWAN (Member, IEEE) received the B.Eng. degree in EE from ITB, Indonesia, in 2008, and the M.Sc. degree in EE from TU Delft, The Netherlands, in 2011. He is currently a member of the Microelectronics Center, ITB. His research interests include intelligent signal processing, MIMO systems, and transceiver design for optical wireless communications.

TRIO ADIONO (Member, IEEE) received the B.Eng. degree in electrical engineering and the M.Eng. degree in microelectronics from the Institut Teknologi Bandung, Indonesia, in 1994 and 1996, respectively, and the Ph.D. degree in VLSI design from the Tokyo Institute of Technology, Japan, in 2002. He holds a Japanese Patent on a high quality video compression system. He is currently a Professor with the School of Electrical Engineering and Informatics and also works as the Head of the IC Design Laboratory, Microelectronics Center, Institut Teknologi Bandung. His research interests include VLSI design, signal and image processing, VLC, smart cards, and electronics solution design and integration.