IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 16, 2020, accepted September 7, 2020, date of publication September 10, 2020,
date of current version September 24, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3023255

FPGA-Based Hardware Accelerator for Leveled Ring-LWE Fully Homomorphic Encryption

YANG SU[1,2], BAILONG YANG[1], CHEN YANG[3], (Member, IEEE), AND LUOGENG TIAN[1,4]

[1]School of Operational Support, PLA Rocket Force University of Engineering, Xi'an 710025, China
[2]School of Cryptography Engineering, Engineering University of People's Armed Police, Xi'an 710086, China
[3]School of Microelectronics, Xi'an Jiaotong University, Xi'an 710049, China
[4]School of Xi'an Communication, National University of Defense Technology, Xi'an 710106, China

Corresponding authors: Yang Su (wj_suyang@126.com) and Chen Yang (chyang00@xjtu.edu.cn)

This work was supported in part by the National Natural Science Foundation (NNSF) of China under Grant 61704136, in part by the China Postdoctoral Science Foundation under Grant 2018M631163, in part by the Shaanxi Postdoctoral Research Project Foundation under Grant 2017BSHEDZZ29, in part by the Fundamental Research Funds for the Central Universities under Grant Z201805200, in part by the Guangdong Province Key Research and Development Project under Grant 2019B010154002, in part by the National Key Research and Development Program of China under Grant 2017YFB1102903, in part by the Key Research and Development Plan of Shaanxi Province under Grant 2019ZDLGY03-07-01, and in part by the Basic Research Foundation of Engineering University of People's Armed Police (PAP) of China under Grant WJY201916.

ABSTRACT Fully homomorphic encryption (FHE) allows arbitrary computation on encrypted data and has great potential in privacy-preserving cloud computing and securely outsource computational tasks. However, the excessive computation complexity is the key limitation that restricting the practical application of FHE. In this paper we proposed a FPGA-based high parallelism architecture to accelerate the FHE schemes based on the ring learning with errors (RLWE) problem, specifically, we presented a fast implementation of leveled fully homomorphic encryption scheme BGV. In order to reduce the computation latency and improve the performance, we applied both circuit-level and block-level pipeline strategies to improve clock frequency, and as a result, enhance the processing speed of polynomial multipliers and homomorphic evaluation functions. At the same time, multiple polynomial multipliers and modular reduction units were deployed in parallel to further improve the hardware performance. Finally, we implemented and tested our architecture on a Virtex UltraScale FPGA platform. Runing at 150MHz, our implementation achieved $4.60\times\sim9.49\times$ speedup with respect to the optimized software implementation on Intel i7 processor running at 3.1GHz for homomorphic encryption and decryption, and the throughput was increased by $1.03\times\sim4.64\times$ compared to the hardware implementation of BGV. While compared to the hardware implementation of FV, the throughput of our accelerator also achieved $5.05\times$ and $167.3\times$ speedup for homomorphic addition and homomorphic multiplication operation respectively.

INDEX TERMS Privacy-preserving, ring-LWE, leveled fully homomorphic encryption, BGV scheme, hardware accelerator, polynomial multiplication, modular reduction, KeySwitch, ModSwitch.

I. INTRODUCTION

The fully homomorphic encryption (FHE) [1] provides a theoretical and practical solution for cloud computing security and privacy-preserving, which can directly perform the arbitrary computations over ciphertext without disclosing the personal sensitive information. Concretely, users can encrypt the data and upload it in the form of ciphertext to the cloud server, and yet perform computations on encrypted data (hidden from cloud owner). Unless the private key of the FHE is obtained, no one can get the plaintext. Because FHE has the property that any computation on ciphertext is equivalent to performing the same computation on plaintext, the users can obtain the final result by decrypting the ciphertext with private keys. Some interesting applications of FHE include: secure outsourcing matching computation on genomic data [2], private information retrieval (PIR) and privacy-preserving data mining (PPDM) [3], multi-party computation (MPC) and privacy-preserving prediction from consumption data in smart electronic instrumentation [4], training neural networks over encrypted data [5], [6] etc.

The concept of FHE was first introduced by Rivest, Adleman, and Dertouzos in 1978 [7]. But constructing FHE schemes proved to be a difficult problem that cannot be

The associate editor coordinating the review of this manuscript and approving it for publication was Yiming Huo.

solved until Gentry proposed the first FHE based on ideal lattice in 2009 [8]. Despite the groundbreaking work, Gentry's scheme was not the practical solution for the low performance. Since then, many researchers and cryptographers have introduced more efficient schemes to improve the performance of FHE, such as DGHV10 [9], BV11 [10], BGV12 [11], FV12 [12], GSW13 [13], FHEW [14], TFHE [15] and so on. Despite great progress in the performance of the algorithms, these FHE schemes are still too slow to be used in practical scenarios. Even the somewhat homomorphic encryption (SWHE) schemes that can realize limited number of operations on encrypted data are slow yet. Speed has become one of the main factors limiting the practical application of FHE.

At present, there are mainly software and hardware two kinds of implementations to accelerate the homomorphic encryption operations. However, the efficiency of software implementations [16]–[21] is still too slow for practical applications. For instance, a homomorphic evaluation of AES-128 in [16] takes over 36 hours based on the NTL C++ library, while the homomorphic evaluation of lightweight block cipher SIMON-32/64 in [17] takes around 20 min and 50 min using C++ implementation of FV and YASHE respectively. Even the GPU-based implementation of matrix-vector multiplication of BGV leveled FHE in [20] needs at least 2 seconds when evaluated on NVIDIA Tesla K20, which has 2,496 cores, 5GB DDR5 memory. On the other hand, the existing hardware implementations [22]–[33] just realize limited several FHE functions and the performance is still low. For instance, Cao *et al.* [23] only proposed a FPGA-based large-integer FFT multiplier and a Barrett modular reduction for accelerating the FHE operations over integers, Wang *et al.* [24] merely presented a 768K-bit multiplier based on 64K-point FFT processor targeting for Gentry-Halevi FHE primitives. Although Pöppelmann *et al.* [28] and Roy *et al.* [30] proposed the homomorphic evaluation architectures for YASHE with different parameter sets respectively, the implementation in [30] evaluated SIMON-64/128 in approximately 157 seconds at 143MHz, the time delay is still very large. The hardware implementations of LTV and FV schemes were also introduced by Doröz *et al.* [29] and Roy *et al.* [32], However, they only focus on accelerating the homomorphic evaluations or homomorphic encryptions and cannot take both into account, and the performance is not high enough.

From the perspective of hardware implementation, the existing FHE hardware accelerators include the FPGA-based implementation and ASIC-based implementation. However, the ASIC-based implementation has the defects of long development period and high cost. While the FPGA-based implementation has the advantages of better programming flexibility, lower development difficulty and cost, and can achieve a good compromise between different design factors. Therefore, this paper uses FPGA to realize FHE accelerator. As a general hardware implementation platform and tool, FPGA has been widely used in various hardware design fields, such as neural network [34], image processing [35], cryptographic algorithms [36], [37], etc. We implement FHE accelerator on a FPGA platform, which has novelty in algorithm selection and hardware acceleration architecture, and it is also a new extension of FPGA platform in fully homomorphic encryption application.

In this paper, we present a complete FPGA-based hardware accelerator for homomorphic encryption and homomorphic evaluation of BGV leveled FHE scheme, which is the first efficient FHE scheme based on Learning with Error (LWE) or Ring LWE (RLWE) problem, and it is an important basis for other FHE variant algorithms. To the best of our knowledge, there are no published complete hardware implementations of Ring-LWE based BGV prior to this paper. A very recently paper by Pedrosa [37] implements the hardware of encryption and decryption of BGV algorithm based on FPGA, but they have not provided the hardware architecture of homomorphic evaluation function. The goal of our accelerator is to provide a complete implementation of a solution that is mature for practical application. We implement all required components for homomorphic encryption and homomorphic evaluation in hardware.

A. OUR CONTRIBUTIONS

The contributions of our paper can be summarized as follows:

We propose an efficient hardware implementation of BGV leveled FHE scheme, to the best of our knowledge, is the first complete FPGA-based Ring-LWE accelerator for BGV algorithm. In contrast to the prior art for other FHE scheme, our architecture supports both homomorphic encryption and homomorphic evaluation computation, and can be tailored according to concrete application needs. We leverage multi-layer parallelism to accelerate the operations from circuit-level to arithmetic block-level. However, we see our work as the first step towards a practical accelerator.

We improve the performance of polynomial multiplication over ring by designing the NTT-based negative wrapped convolution (NWC) algorithm, which adopts four-level pipelines and a single round iterative structure. The optimized structure can achieve a good trade-off between performance and area. In addition, a resource-saving and high performance modular reduction algorithm is presented, which occupies only half of resources of Barrett reduction.

We introduce the hardware architecture of KeySwith module and ModSwitch module which are necessary to implement the leveled BGV FHE scheme. For KeySwitch, we select the switchkey parameters that are more suitable for hardware implementation, and improve the efficiency of KeySwith by using multi-level pipelines. We propose the first hardware structure of ModSwitch which can decrease the noise of ciphertext of homomorphic evaluation by using a smaller modulus.

B. PAPER OUTLINE

The organization of this paper is as follows. Section II describes the related works and BGV leveled FHE scheme,

as well as the parameter set that we use. Section III provides the algorithms and optimization methods for computation intensive operation of homomorphic evaluation function. The hardware architecture overview and details are provided in Section IV. The resource utilization and performance of our implementations are shown in Section V. Section VI summarizes the paper.

II. BACKGROUND AND RELATED WORK

A. RELATED WORK

As mentioned above, software implementations are not yet efficient enough for real-time applications, which may require minutes or hours to evaluate some simple functions or algorithms. For instance, a homomorphic evaluation of AES-128 [16] is reported to take over 36 hours based on the NTL C++ library, and running on a Intel Xeon processor with 2.0GHz and 256GB RAM. Even using SIMD techniques, the amortized rate is about 40 minutes per block. Another software homomorphic evaluation of the decryption function of a lightweight block cipher SIMON-32/64 (resp. SIMON-64/128) [17] is reported to take around 3062s (resp. 12418s) and 1029s (resp. 4196s) using C++ implementation of FV and YASHE respectively on 4-core Intel Core i7 processor at 3.4GHz. To address the shortcomings of software implementations, many optimized architectures and accelerators based on Graphics Processing Units (GPUs) and FPGAs/ASICs have been proposed.

GPU is an alternative computing platform to accelerate the homomorphic evaluation in FHE. Wang *et al.* [18] proposed the first GPU-based accelerator of FHE targeted at Gentry-Halevi scheme [19], the FHE primitives were implemented on NVIDIA C2050 GPU with a dimension of 2048, and achieved speedup factors of around 7 compared to original CPU implementation, which was on Intel Xeon X5650 processor running at 2.67GHz, 14GB RAM. Then a GPU-based implementation of BGV leveled FHE accelerator was introduced by Wang *et al.* [20] further, the CRT-based matrix-vector multiplication was evaluated on NVIDIA Tesla K20, which had 35.2 times and 273.6 times speedup compared to the CRT-based method and NTL library implementations on CPU. Badawi *et al.* [21] proposed the multi-threaded CPU execution as well as GPU implementation of RNS variants of the BFV scheme on NVIDIA Tesla K80 and V100-PCIe, the performance was faster by two orders of magnitude than prior results. However, GPU-based implementations normally offer less performance per watt of power and the speed is still too low compared to hardware implementations.

Many FPGA-based or ASIC-based accelerators have been proposed to improve the performance of FHE schemes. A lines of research focuses on the large integer multiplication hardware accelerations [22]–[27], which are the main bottlenecks of FHE schemes. Cao *et al.* [23] proposed the first hardware implementations of encryption primitives for FHE over integers based on Xilinx Virtex-7 FPGA platform, the performance of which was improved a factor of up to 44 compared to corresponding software implementation. A large-integer FFT multiplier and a Barrett modular reduction were proposed that could accelerate the FHE operations by 11 times. A 768K-bit multiplier based on 64K-point FFT processor was introduced by Wang *et al.* [24], the multiplier was prototyped on Altera Stratix-V FPGA at 100MHz, and was about twice as fast as the same algorithm executed on the NVIDA C2050 GPU at 1.15GHz. Doröz *et al.* [27] presented a custom architecture for Gentry-Halevi FHE scheme, the architecture featured an optimized multi-million bit multiplier based on Schönhage-Strassen multiplication algorithm, which occupied a footprint of less than 30 million gates with the frequency of 666MHz when synthesized using 90nm TSM library, the performance was equivalent to that of Xeon software but slower than GPU implementation.

Several works [28]–[33] focus on improving the performance of concrete FHE schemes based on FPGAs. Pöppelmann *et al.* [28] proposed an architecture for YASHE [38] scheme and implemented their design on the Catapult board equipped with an Altera Stratix V FPGA and two 4GB DRAMs, an efficient double-buffered memory access scheme and a Number Theoretic Transform (NTT) based polynomial multiplier were proposed, for parameter set ($n = 16384$, $\lceil \log_2 q \rceil = 512$), they can perform a homomorphic addition in 0.94ms and a homomomorphic multiplication in 48.67ms. a hardware accelerator for LTV [39] based SWHE scheme was introduced by Doröz *et al.* [29], when synthesized for Xilinx Virtex-7 the presented architecture can compute the product of large polynomials in 6.25msec which is more than 102 times faster than software implementation. Roy *et al.* [30] also proposed the hardware architecture for YASHE scheme, which was compiled on a Xilinx Virtex-7 FPGA, the implementation evaluated SIMON-64/128 in approximately 157.s at 143MHz, and was 26.6 times faster than software implementation. However, the assumption of unlimited memory bandwidth which renders off-chip memory accesses free of cost is not a realistic. Perhaps, the closest work to ours was by Roy *et al.* [32], in which the authors presented an architecture for FV scheme and implement the design on Xilinx Zynq UltraScale+ MPSoC ZCU102, the implementation achieved over 13 times speedup at 200MHz with respect to the FV-NFLlib executing on an Intel i5 processor running at 1.8GHz.

B. BGV FHE SCHEME

In this section we briefly introduce the BGV fully homomorphic encryption scheme. The BGV scheme was proposed by Brakerski, Gentry, and Vaikuntannathan [11] in 2012. It is the first leveled FHE scheme that without Gentry's bootstrapping procedure and provides a choice of basing on the learning with error (LWE) or Ring-LWE (RLWE) problems that have 2^{λ} security against know attacks, this paper mainly focuses on the Ring-LWE based BGV.

For mathematical preliminaries, suppose the polynomial degree n is a power-of-two integer, we define an integer polynomial ring $R = \mathbb{Z}[x]/f(x)$ with reduction polynomial

$f(x) = x^n + 1$, whose elements have degrees at most $n-1$. For the ciphertext modulus q, we define the ciphertext space $R_q = R/qR$ for the residue ring of R by modulus q for each coefficient of polynomials. In actual computation, we represent the coefficients of R_q in $[0, q-1] \cap \mathbb{Z}$. For the plaintext modulus p, the plaintext space defined as $R_p = R/pR$, where each coefficient of polynomial is represented as a integer modulus p. In the operations of key generation, encryption and KeySwitch of BGV, the polynomials sampled from discrete Gaussian distribution χ_σ with a small standard deviation σ is defined on R. In practice, we take the private key as a polynomial with coefficients form a narrow set like $\{-1, 0, 1\}$. The security of scheme is determined by the degree n, the size of the ciphertext modulus q, and the Gaussian distribution χ_σ.

With these preliminaries, now we enumerate the functions used in BGV scheme as follows.

1) **BGV.Setup**(λ): For a given security parameter λ, choose the polynomial degree n, ciphertext modulus q, plaintext modulus p, and Gaussian distribution χ_σ. Normally, n is a power-of-two integer, q is a positive integer satisfying $q \equiv 1 \bmod (2n)$, $p = 2$ or a positive integer much less than q. Return the system parameters $params = (n, p, q, \chi_\sigma)$.

2) **BGV.KeyGen**($params$): Sample polynomial $s' \leftarrow \chi_\sigma$. Return the secret key $sk = \mathbf{s} = (1, s') \in R_q^2$. Sample polynomial $a' \leftarrow R_q$ uniformly at random and error polynomial $e' \leftarrow \chi_\sigma$. Compute $b \leftarrow a's' + pe' \in R_q$, where $p \ll q$. Return the public key $pk = (b, -a') \in R_q^2$, which satisfies the equation $pk \cdot \mathbf{s} = (b, -a') \cdot (1, s') = b - a's' = pe'$. The scheme needs another key called switching key in the function of SwitchKey, to compute the switching key, we first choose the parameter $t \leq q$, sample polynomial vector $\mathbf{ex\_a} \in R_q^\ell$ and $\mathbf{ex\_e} \leftarrow \chi_\sigma^\ell$ uniformly, then compute $\mathbf{ex\_b} = \mathbf{ex\_a} \cdot s' + p \cdot \mathbf{ex\_e} + Powerof((s')^2) \in R_q^\ell$. The function $Powerof_{t,\ell}(a)$ scales an element $a \in R_q$ by the different powers of t, namely, $Powerof_{t,\ell}(a) = (a \cdot t^i)_{i=0}^{\ell}$, where $\ell = \lfloor \log_t q \rfloor$ in this scheme. At last, return the switching key $epk = (\mathbf{rlk}_0, \mathbf{rlk}_1) = (-\mathbf{ex\_a}, \mathbf{ex\_b}) \in (R_q^\ell, R_q^\ell)$.

3) **BGV.Enc**($params, pk, m$): First encode the input plaintext $m \in R_p$ into a polynomial vector $\mathbf{m} = (m, 0) \in R_q^2$. Next sample error polynomial vector $\mathbf{e} = (e_1, e_2) \leftarrow \chi_\sigma$ and polynomial $r \leftarrow R_q$ uniform at random. Then compute the pair of ciphertexts $\mathbf{ct} = (c_0, c_1) \leftarrow \mathbf{m} + p\mathbf{e} + pk \cdot r \in R_q^2$, namely, $c_0 = m + pe_1 + b \cdot r \in R_q$ and $c_1 = pe_2 - a' \cdot r \in R_q$.

4) **BGV.Dec**($params, sk, \mathbf{ct}$): output the plaintext $m \leftarrow [[<\mathbf{ct}, sk>]_q]_p$. Note that for $a \in R$, we use the notation $[a]_q$ to refer to a mod q, with coefficients reduced into the range $(-q/2, q/2]$. The concrete decryption derivation process will not be described in detail.

5) **BGV.HomAdd**($\mathbf{ct}_1, \mathbf{ct}_2$): For ciphertext polynomials $\mathbf{ct_1} = (c_0, c_1)$ and $\mathbf{ct_2} = (c_0', c_1')$, the homomorphic addition can be expressed by $\mathbf{ct}_{HomAdd} = \mathbf{ct}_1 + \mathbf{ct}_2 = (c_0 + c_0', c_1 + c_1')$, corresponding to decryption key $\mathbf{s} = (1, s')$.

6) **BGV.HomMult**($\mathbf{ct}_1, \mathbf{ct}_2$): For ciphertext polynomials $\mathbf{ct_1} = (c_0, c_1)$ and $\mathbf{ct_2} = (c_0', c_1')$, the homomorphic multiplication result can be compute by the equation $\mathbf{ct}_{HomMult} = \mathbf{ct}_1 \cdot \mathbf{ct}_2 = (d_0, d_1, d_2) = (c_0c_0', c_0c_1' + c_1c_0', c_1c_1')$. The corresponding decryption key is $\mathbf{s}_{Mul} = (1, s', s'^2)$.

7) **BGV.KeySwitch**($epk, (d_0, d_1, d_2)$): Compute the KeySwitch ciphertext $\mathbf{ct}_{keyswitch} = (\tilde{c}_0, \tilde{c}_1)$, where $\tilde{c}_0 = d_0 + <WordDecomp(d_2), \mathbf{rlk}_0>$, $\tilde{c}_1 = d_1 + <WordDecomp(d_2), \mathbf{rlk}_1>$. The function $WordDecomp_{t,\ell}(a)$ is used to decomposes an element $a \in R_q$ in base t by slicing each coefficient, namely, for $\ell = \lceil \log_t q \rceil$, this function returns $a_i \in R$ with each coefficient in $[0, t)$, where $a = \sum_{i=0}^{\ell-1} a_i \cdot t^i$. Note that the function $WordDecomp_{t,\ell}(a)$ and $Powerof_{t,\ell}(a)$ satisfy the equation $<WordDecomp_{t,\ell}(a), Powerof_{t,\ell}(b)> = a \cdot b \bmod q$.

8) **BGV.ModSwitch**($\mathbf{ct}_{keyswitch}, q_l, q_{l'}$): This takes a ciphertext $\mathbf{ct}_{keyswitch}$ at with modulus q_l (say one with modulus q_l at level l, initially $q_l = q$ in this scheme) and returns a ciphertext with modulus $q_{l'}$ where $q_{l'} < q_l$ (say a ciphertext with modulus $q_{l'}$ at level l', and $l' > l$). Given $\mathbf{ct}_{keyswitch} = (\tilde{c}_0, \tilde{c}_1)$ initially, we first compute $\tilde{c}_0' = (q_{l'}/q_l) \cdot \tilde{c}_0$ and $\tilde{c}_1' = (q_{l'}/q_l) \cdot \tilde{c}_1$ over $\mathbb{Q}[x]$, then round c_i' to the nearest integer polynomial, such that $c_i' = c_i (\bmod p)$. Return the ModSwtich ciphertext $\mathbf{ct}_{\bmod swith} = (\tilde{c}_0', \tilde{c}_1')$ with less noise.

C. PARAMETER SET

From the perspective of proof-of-concept, we use a small parameter set with the polynomial degree $n = 128$ (namely, the degree of the reduction polynomial $f(x)$ is 128). In order to take advantage of the structure of polynomial ring and make our hardware implementation more practical, we choose the plaintext modulus $p = 32$ (i.e. $\log_2(p) = 5$ bits) and the ciphertext modulus $q = 257^3$(i.e. $\log_2(q) = 25$ bits), so we can not only evaluate bit-level operations, but also the integer-level operations. By using the packing method [40], we can embed multiple plaintexts into different coefficients in a single ciphertext and evaluate a function on all of them in parallel with a single execution. Normally, the ciphertext q is chosen as a big prime integer and satisfying $q \equiv 1 \bmod (2n)$. However, we choose q as the product of three primes mainly for the convenience of implementing and verifying the ModSwitch primitive of BGV, which can change the original modulus to a smaller number while preserving the correctness of decryption under the same secret key. Moreover, since the coefficients of polynomials in BGV scheme are signed integers, to keep some redundancy, we set the bit-width of polynomial coefficient and modulus q to 27-bit signed integers in our actual implementation. Following [11], [31], we take the discrete Gaussian distribution χ_σ as a polynomial with coefficients from $\{-1, 0, 1\}$ uniformly at random for simplicity. Despite this may affect the security to some extent, it is feasible from the perspective of hardware implementation of primitives. The parameters of our design are detailed in Table 1.

III. ALGORITHMS AND OPTIMIZATIONS

Through the analysis of BGV scheme, we find that the polynomial multiplication and modular reduction are the most

TABLE 1. Parameter set for accelerator.

Parameter	Value	Description
n	128	Degree of polynomial
p	32	Plaintext modulus (5 bits)
q	16974593	Ciphertext modulus (27 bits)
ω	908870	Primitive n-th root of unity in R_q
e	3259673	Square root of ω in R_q

frequently used and time-consuming operations. Without considering the encryption and decryption algorithm (normally belonging to the client), only homomorphic multiplication and KeySwitch operations (normally belonging to the server) need at least 8 polynomial multiplications and much more modular reductions each time. Therefore, we optimize these two operations respectively which will lay the foundation for the later BGV hardware implementation.

A. POLYNOMIAL MULTIPLICATION

In our parameter set, the polynomials consist of 128 coefficients and each has the size of 27 bits signed integers. For such large polynomials and coefficients, the computation time is significantly determined by the complexity of polynomial multiplication algorithm. At present, the main methods of polynomial multiplication include [41]: School-book algorithm, Karatsuba based algorithm, Toom algorithm, Fast Fouriter Transform (FFT) based algorithm and so on. FFT-based polynomial multiplication applies a divide and conquer technique to reduce the computation of the Discrete Fourier Transform (DFT) into smaller problems and has the lowest time complexity $O(n \log n)$, During a polynomial multiplication, forward FFT transform is applied on the input polynomials to bring them to Fourier domain. Then, a coefficient-wise multiplication is performed in Fourier domain. Finally, an inverse-FFT (IFFT) transform is required to bring the results back to polynomial representations. However, the FFT and IFFT transforms are computed on the real number field, and thus suffer from the approximation errors, which is not suitable for cryptographic applications. Instead, we use the NTT transform which is a generalization of FFT and performs the polynomial multiplication.

Suppose $a(x)$ is the polynomial of degree less than n in the ring R_q, let ω be a primitive n-th root of unity in R_q, Then the n-point NTT of $a(x)$ are defined as follows:

$$A_i = \sum_{j=0}^{n-1} a_j \omega^{ij} \bmod q, \quad i = 0, 1, \ldots, n-1 \tag{1}$$

And n-point inverse-NTT (INTT) can be calculated by the following formula:

$$a_i = n^{-1} \sum_{j=0}^{n-1} A_j w^{-ij} \bmod q, \quad i = 0, 1, \ldots, n-1 \tag{2}$$

Since q is the product of primes, n has an inverse n^{-1} modulo q, where $n \cdot n^{-1} \equiv 1 \bmod q$, ω has an inverse ω^{-1} modulo q satisfying $\omega \cdot \omega^{-1} \equiv 1 \bmod q$. Note that NTT transform holds if and only if $p-1$ can be divided by n for each prime factor p of q, and for the primitive n-th root of unity ω, it satisfies $\omega^n = 1 \bmod q$. Because the time complexity of basic NTT algorithm is still $O(n^2)$ and does not have any advantages, so we use the butterfly-based NTT transformation with time complexity $O(n \log n)$ to construct the polynomial multiplications. An iterative version of the butterfly-based NTT algorithm is shown in Algorithm 1. There are three nested loops performing the NTT algorithm, inside the inner-most loop, the butterfly operation which consists of a modular multiplication by the twiddle factors ω_m followed by a modular addition and modular subtraction is computed.

Algorithm 1 Iterative Butterfly-Based NTT

Input: Coefficient vector $\boldsymbol{a} = (a_0, \ldots, a_{n-1})$ of degree n for the polynomial $a(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$, where $a_i \in R_q$, $i = 1, 2, \ldots, n-1$, primitive n-th root of unity $\omega \in R_q$.
Output: Transformed vector $\mathbf{A} = NTT^n_\omega(\mathbf{a})$.

```
for m ← n/2 to 1 by m/2 do
  for j ← 0 to m do
    ω_m ← ω^{jn/2m}
    for i ← j to n by 2m do
      a[i] ← (a[i] + a[i + m]) mod q
      a[i + m] ← ω_m(a[i] − a[i + m]) mod q
    end for
  end for
end for
A ← BitReverse(a)
Return(A)
```

Using the NTT algorithm, one can perform the polynomial multiplication through application of the convolution theorem efficiently. Let R_q be an arbitrary ring with polynomials $a(x)$ and $b(x)$ of degree $n-1$ with the coefficients $a_i \in R_q$ and $b_i \in R_q$ for $i = 0, 1, \ldots, n-1$. Then the convolution of coefficient vectors results in the product polynomial $c(x)$ of degree $2n-2$, the coefficients of which are $c_i \in R_q$ for $i = 0, 1, \ldots, 2n-2$. Note that the product vector $\mathbf{c}$ has a length of $2n$, the input vectors $\mathbf{a}$ and $\mathbf{b}$ must be zero padded to a length greater or equal to $2n$ with the form of 2^k to get the correct result. Though the time complexity is linear using NTT algorithm, the length of vectors and the number of point-wise multiplications are doubled.

In this paper, we use the negative wrapped convolution method [42] to compute the product of polynomials $a(x)$ and $b(x)$ in R_q, which can avoid the zero padding. Let $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$, $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})$, and $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ be vectors of length n, then each elements of $\mathbf{c}$ can be calculated as

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} - \sum_{j=i+1}^{n-1} a_j b_{n+i-j} \tag{3}$$

This computation is equivalent to perform the polynomial multiplication modulo $(x^n + 1)$, which has the property $x^n \equiv -1 \bmod (x^n + 1)$. Let ω be the n-th primitive root of unity in R_q and e is the square root of ω, which satisfies the equation $e^2 = \omega \bmod q$. In order to guarantee the existence of e, when q is the product of primes or a prime and n is power of 2, we should have $q = 1 \bmod 2n$. To compute the negative wrapped convolution based on NTT, we do the following first:

$$\tilde{\mathbf{a}} = (a_0, ea_1, \ldots, e^{n-1}a_{n-1}) \tag{4}$$

$$\tilde{\mathbf{b}} = (b_0, eb_1, \ldots, e^{n-1}b_{n-1}) \tag{5}$$

Then the negative wrapped convolution of $\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$ can be computed by NTT and $INTT$ as

$$\tilde{\mathbf{c}} = INTT^n_\omega(NTT^n_\omega(\tilde{\mathbf{a}}) \cdot NTT^n_\omega(\tilde{\mathbf{b}})) \tag{6}$$

At last, we perform the following operations to get the final result of polynomial multiplication of $a(x)$ and $b(x)$:

$$\mathbf{c} = e^{-i}\tilde{\mathbf{c}} = (\tilde{c}_0, e^{-1}\tilde{c}_1, \ldots, e^{-(n-1)}\tilde{c}_{n-1}) \tag{7}$$

Details of polynomial multiplication of negative wrapped convolution using NTT is shown in Algorithm 2. Compared with the zero padding method, the negative wrapped convolution method can reduce the length of NTT transformation and point-wise multiplication from $2n$ to n. Moreover, the modular operation x^n+1 is omitted. Therefore, the performance of polynomial multiplication can be improved greatly, especially with the increasing of the length of polynomials, the improvement becomes more obviously.

Algorithm 2 NTT-Based Polynomial Multiplication

Input: Coefficient vectors $\mathbf{a} = (a_0, a_1, \ldots, a_{n-1})$ and $\mathbf{b} = (b_0, b_1, \ldots, b_{n-1})$ of polynomials $a(x)$ and $b(x)$ respectively, where $a_i, b_i \in R_q$, $i = 0, 1, \ldots, n-1$, primitive n-th root of unity ω and ω^{-1} in R_q, square root e and e^{-1} of ω, polynomial degree n and n^{-1}.
Output: Coefficient vector $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ of polynomial $c(x)$, where $c(x) = a(x) \cdot b(x) \bmod (x^n + 1)$, and $c_i \in R_q$, $i = 0, 1, \ldots, n-1$.
1: **Pre-computation:** $\omega^i \bmod q$, $\omega^{-i} \bmod q$, $e^i \bmod q$, $e^{-i} \bmod q$, where $i = 0, 1, \ldots, n-1$.
2: **for** $i = 0$ to $n-1$ **do**
3: $\tilde{a}_i = (e^i \cdot a_i) \bmod q$
4: $\tilde{b}_i = (e^i \cdot b_i) \bmod q$
5: **end for**
6: $\tilde{\mathbf{A}} \leftarrow NTT^n_\omega(\tilde{\mathbf{a}})$
7: $\tilde{\mathbf{B}} \leftarrow NTT^n_\omega(\tilde{\mathbf{b}})$
8: **for** $i = 0$ to $n-1$ **do**
9: $\tilde{C}_i \leftarrow \tilde{A}_i \cdot \tilde{B}_i \bmod q$
10: **end for**
11: $\tilde{\mathbf{c}} = INTT^n_\omega(\tilde{\mathbf{C}})$
12: **for** $i = 0$ to $n-1$ **do**
13: $c_i = (e^{-i} \cdot \tilde{c}_i) \bmod q$
14: **end for**
15: **Return (c)**

B. MODULAR REDUCTION

Modular reduction is another time-consuming unit in BGV scheme, especially for polynomial multiplications. Almost all polynomial coefficients in ciphertext space are bounded over R_q, where $q = 16974593$ in our scheme. Concretely, a modular reduction must be performed after each multiplication in order to work with elements in the polynomial ring at all times, the same is true for multiple addition operations. Modular reduction operations take a significant amount of time and resources, representing a critical part within the polynomial multiplications. Therefore, it is important to design an efficient and area saving modular reduction. Since the modular reduction of addition is relatively simple, it only needs a few steps of conditional subtraction operation. This paper focuses on the modular reduction of multiplication.

Some previous works exist to perform efficient general modular reduction based on the specific modulus, such as Solinas primes [28] or pseudo-Fermat primes [42], and most of them use the Barret modular reduction algorithm [30], [36]. However, these implementations are restrictive and do not allow for arbitrary selection of the modulus value, and through some experiments, it is confirmed that these methods take up a relatively large resource for the value of q that we choose.

Therefore, from the perspective of hardware design, we propose a lookup table (LUT) based modular reduction algorithm as shown in Algorithm 3. We adopt a divide and conquer strategy. Let the bit length of modulus q is k, and the bit length of input number x is m. When $m > k$ we firstly divide the bits higher than k and the bits lower than k. For the bits lower than k, we judge whether the value of x is greater than q, and determine whether to subtract q from x. Moreover, for the bits higher than k, we divide every four

Algorithm 3 Modular Reduction Algorithm

Let k be the bit length of q, and x be the integer with a maximum bit length m.
Input: x, q, k, m
Output: $y = x \bmod q$
1: **while** $m > k$ **do**
2: **if** $x[k-1:0] > q$ **then**
3: $t_0 = x[k-1:0] - q$
4: **else**
5: $t_0 = x[k-1:0]$
6: **end if**
7: **for** $i = 1, sum = 0$ to $\lceil (m-k)/4 \rceil + 1$ **do**
8: **Pre-computation:**
$t_i = x[k+4(i-1)+3, k+4(i-1)] \cdot 2^{k+4(i-1)} \bmod q$
9: $sum = sum + t_i$
10: **end for**
11: $x = sum$
12: $m = sum.length$
13: **end while**
14: $y = x$
15: **Return**(y)

bits into a group, and perform pre-computations, which can be mapped into a 4-input LUT in hardware. Then add the results, and take the sum result as a new value x and the length as a new value m, repeat above operations until $m \leq k$, and output the modular reduction result y. Compared with traditional methods, our modular reduction algorithm can support arbitrary length of modular operation. At the same time, by introducing the pre-computation method, we can quickly reduce the number of input bits, the overall algorithm complexity and resource occupation.

IV. ARCHITECTURE OVERVIEW

In this section, we first describe the overall architecture design of BGV accelerator. Then the kernel acceleration units including polynomial multiplication and modular reduction are introduced in detail. Finally, the hardware implementation of each component of BGV accelerator will be presented.

A. OVERALL ARCHITECTURE

The architecture overview of our BGV accelerator is presented in Fig.1. Despite the goal of our implementation is to accelerate the server-based homomorphic evaluation operations BGV.HomMult and BGV.KeySwitch (and polynomial multiplication and modular reduction in general), from the perspective of proof-of-concept, we also implement the encryption and decryption operations which are assumed to be performed on a client. We would like to note that except for the Gaussian sampler, almost all components required for BGV scheme are already present in our design. By default, we think that the secret key $sk = (1, s')$, public key $pk = (b, -a')$ and switch key $epk = (\mathbf{rlk}_0, \mathbf{rlk}_1)$ are directly generated by CPU, the plaintext polynomials m_1 and m_2 are read from the memory of the CPU according to the requirements of the specific application program.

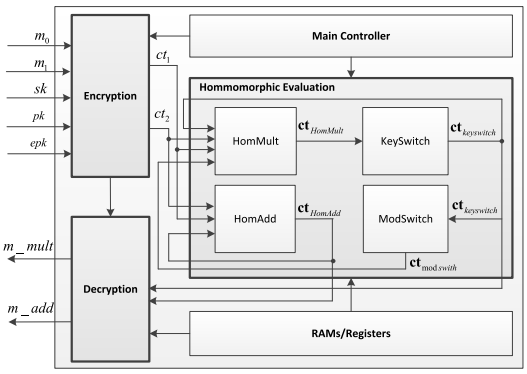


FIGURE 1. Architecture overview of BGV accelerator.

When the accelerator receives the plaintext polynomials, it first performs the encryption operation under the control of main controller. Then, the ciphertext ct_1 and ct_2 are sent to homomorphic addition and homomorphic multiplication module respectively to perform homomorphic evaluation computations. Due to the problems of dimension expansion of ciphertext and the excessive noise growth in homomorphic multiplication, the KeySwitch and ModSwitch operations are required. If more than one homomorphic addition or multiplication operation is needed, the result of homomorphic evaluation function can be fed back to the corresponding unit to perform the next round computations. Otherwise, the final result of homomorphic addition or multiplication can be obtained by decryption.

It should be noted that the modulus q in our polynomial multiplication unit is fixed, so the output of ModSwitch cannot be directly decrypted by the accelerator. The ModSwitch module in our accelerator is just a primitive verification, and it is not connected to the decryption circuit. In order to reduce the length of critical path and keep the data updated in time, the intermediate results or parameters can be temporarily stored in block RAMs or registers. Further analysis shows that, compared with homomorphic addition, the calculation path of homomorphic multiplication is extremely long, it is necessary to optimize the kernel computing units (e.g. polynomial multiplication and modular reduction) and KeySwitch module.

B. KERNEL ACCELERATION UNITS

1) NTT-BASED POLYNOMIAL MULTIPLIER

Following the iterative version of the butterfly based NTT algorithm, a pipelined architecture for the NTT transformation is designed and presented in Figure 2. As can be seen from Figure 2, the overall data path is divided into four functional processing units. First of all, in order to temporarily store the input data and enable pipelining in our architecture, a simple dual-port RAM, which can read and write concurrently, are used to store the n coefficients of polynomials. The dual-port RAM has two input buses and two output buses, corresponding to the upper data path and lower data path of butterfly unit respectively. Since the dual-port RAM has two ports for reading and two ports for writing, it can support two reads and writes in a cycle. Then, the butterfly unit with Decimation in Frequency (DIF) structure performs the addition and multiplication with modular reduction and shares the same data path for all the stages in NTT, the output data is feedback to the dual-port RAM for the next butterfly computation. Instead of generating the $\omega^i \bmod q$, $\omega^{-i} \bmod q$, $e^i \bmod q$ and $e^{-i} \bmod q$ on the fly, the ROMs are designed to store these pre-computed values respectively. In addition,

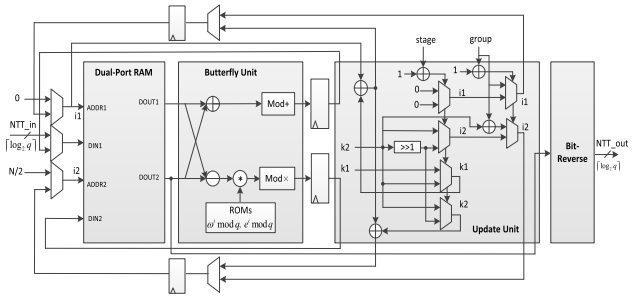


FIGURE 2. Architecture for NTT transformation.

an update unit is responsible for updating the address index i_1, address index i_2, offset k_1, offset k_2, group index and stage index of NTT transformation, and writing the coefficients back to corresponding RAM address. With the fully pipelined architecture, after a certain period of computation, the coefficients of the NTT will appear successively at the output at each cycle. Last but not least, since the input values of polynomials occur in natural order, while the index of the NTT values is in bit reversed order, a Bit-Reverse operation is performed after the computation of NTT to reverse the order of the output sequence.

Update unit is the most significant and complex component of NTT, which directly determines the processing order of butterfly operations. According to the parameter set we proposed in Section II and the NTT algorithm presented in Section III, the NTT transform consists of $\log_2 n = 7$ stages, from stage 1 to stage 7. Each stage is divided into multiple groups, and each group further contains multiple butterflies. When the stage index is increased by one level, the number of groups is doubled, and the number of butterflies in each group is halved. At the last stage, there are total $n/2 = 64$ groups, and a group includes only a single butterfly. In addition, note that the exponents of twiddle factors $\omega^i \bmod q$ only participate in the multiplication of the lower part of each group, while the upper part remains unchanged, and for each group the same type of twiddle factors occurs. The address index i_1 is used to write the upper path data of butterfly in each group to the dual-port RAM, and the address index i_2 performs the similar operations, but is the lower path data of the same butterfly. The parameter k_1 and k_2 are the inter group offset and intra group offset of the address index i_1 and i_2 respectively, and they satisfy the relationship $k_1 = 2k_2$.

With the NTT processing unit mentioned above, we find that the structure of INTT is almost the same as the structure of NTT, except that the exponents of twiddle factors $\omega^i \bmod q$ are replaced by $\omega^{-i} \bmod q$, and an additional multiplication by $n^{-1} \bmod q$ is needed at the end of the INTT. Hence, the INTT processing unit can reuse the NTT unit, which can simplify the complexity of control logic. The data flow of NTT based multiplier is depicted in Figure 3. Considering that step 3, 4 and step 6, 7 have no data dependency in Algorithm 2, we can enable the high speed design by parallel processing these two steps. Owing to the more compact iterative butterfly based NTT structure, thus the $NTT^n_\omega(\tilde{\mathbf{a}})$ and $NTT^n_\omega(\tilde{\mathbf{b}})$ can be computed directly by copy the NTT structure, and the $INTT^n_\omega(\tilde{\mathbf{C}})$ can also be computed using the NTT structure just substitute for the powers of twiddle factors. The coefficients point-wise multiplication can be performed by DSPs between the NTT and INTT.

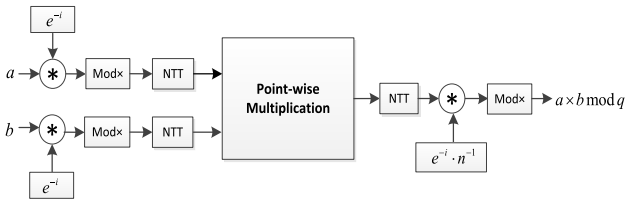


FIGURE 3. Data flow of NTT based multiplier.

Carefully researching the architecture of NTT based multiplier, one can find that in the last stage (stage 7 in our design), the parameter j always equals to 0 in algorithm 2, correspondingly, the twiddle factor $\omega_m = 1$. This indicates that the multiplication of $\omega_m(a[i] - a[i+m])$ always equals to $(a[i] - a[i+m])$. In other words, only coefficient addition and subtraction operations of $a[i]$ and $a[i+m]$ are performed in the last stage. In order to fully utilize the last stage of NTT, the point-wise multiplication can be absorbed in the last stage. That is, instead of performing the multiplication by the powers of ω_m in the last stage, the point multiplication is computed directly, which will lead to a reduction of n cycles compared with the original method. In general, the values of $n^{-1} \bmod q$ and $e^{-i} \bmod q$ are pre-computed and their multiplications are performed separately, we improve this point by pre-computing the values of $e^{-i} \cdot n^{-1} \bmod q$ directly, which can further save n cycles. Furthermore, similar to the point-wise multiplication, the multiplication by the $e^{-i} \cdot n^{-1} \bmod q$ can also be combined with the multiplication by the powers of ω_m in the last stage of INTT, and the number of cycles is further reduced by n. The cycle requirement for the NTT based multiplier is presented in Table 2. Without considering the pipeline, the cycles required for one polynomial multiplication can be reduced to $n \cdot (\log_2 n + 1)$.

TABLE 2. Cycle analysis of NTT-based multiplier.

Operation	Original Design	Our Design
Multiply by e^i	n	n
NTT(stage $0 \sim \log_2 n - 2$)	$n \cdot (\log_2 n - 1)$	$n \cdot (\log_2 n - 1)$
NTT(stage $\log_2 n - 1$)	n	n
Point-wise multiplication	n	
INTT(stage $0 \sim \log_2 n - 2$)	$n \cdot (\log_2 n - 1)$	$n \cdot (\log_2 n - 1)$
INTT(stage $\log_2 n - 1$)	n	n
Multiply by e^i	n	
Multiply by n^{-1}	n	

2) LUT-BASED MODULAR REDUCTION

Since the coefficients of polynomials in BGV scheme are signed integers, and modulo q is a 25-bit number, in order to keep some redundancy, we set the bit-width of polynomial coefficient and modulus q to 27-bit signed integers. Correspondingly, the product of two polynomial coefficients will produce a maximum 54-bit signed number. The modular reduction unit is used to reduce the 54-bit output from the polynomial multiplication by 27-bit modulus q. Considering the input data is 27-bit larger than modulus q, the efficiency of bit-by-bit modular reduction is too slow. Hence, we propose a resource-saving and high performance LUT-based modular reduction unit to perform the reduction of multiplication in $[0, q-1]$, and then we can central-lift the result to a value in $(-q/2, q/2)$. Since the input of additive modular reduction

increases only a few bits compared to modulus q, generally 1∼2 bits, so the architecture is relatively simple, it only needs a few steps of conditional subtraction operations by judging whether input data is larger than modulus q or not, i.e. if it is larger than q, then we subtract q from input data as the output, otherwise, we directly output original input data. This paper mainly focuses on the architecture of modular reduction of multiplication. The architecture for computing the reduction modulo q for multiplication is presented in Figure 4.

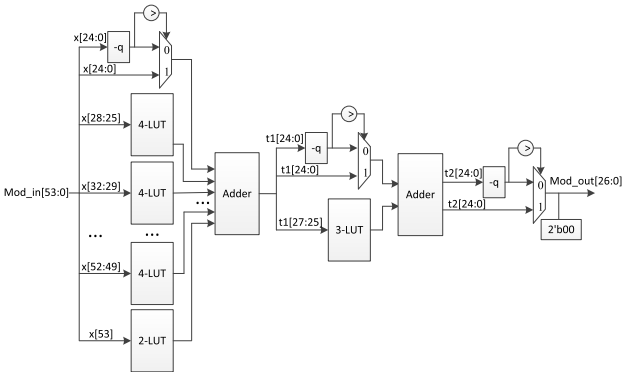


FIGURE 4. Architecture for reduction modulo q.

As can be seen from Figure 4, the architecture has three stages. At the beginning of the computation, the low 25-bit input data is compared with the modulus q by performing a conditional subtraction. Based on the sign of the subtraction, either $x[24:0]$ or $x[24:0]-q$ participates in the subsequent computation as the input of the adder. For the high 27-bit input data, we can take every 4 bits as a group, and pre-compute the modular reductions of each group and store them in the corresponding 4-input LUTs. Note that the high 27-bit cannot be divided by 4, so the remaining 2 bits can be set as the input address of a 2-input LUT separately. Then we sum all the outputs of LUTs and MUX using the adder circuit. Similarly, in the stage 2, we perform the conditional subtraction on the addition result of the previous stage, and pre-compute the LUT value of the remaining high 3-bit input data. In the last stage, only a conditional subtraction is performed on the previous addition result, and the final reduction result is output by padding 2'b00 to the most significant 2 bits. Through analysis, we find that in our implementation, the number of input bits can be reduced a lot after each stage, for example in stage 1, the number of bits has been reduced from 54-bit to 27-bit, almost a half. In addition, since the LUTs are the inherent resource of FPGA, we can make full use the hardware resource of FPGA to improve the performance of modular reduction and decrease the area cost. Note that our modular reduction architecture can be easily extended to support arbitrary modulo with larger bit-width.

C. UNITS OF HOMOMORPHIC ENCRYPTION

Theoretically, the homomorphic encryption runs on the client side, including encryption and decryption algorithms. However, for functional integrity and performance optimization,

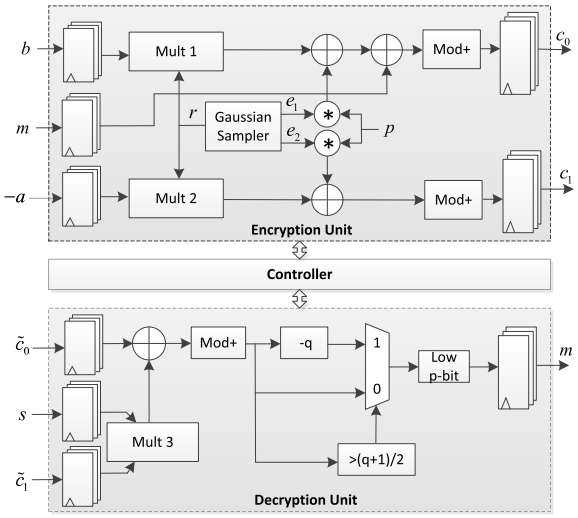


FIGURE 5. Architecture for encryption and decryption.

this paper also gives the architecture of homomorphic encryption as depicted in Figure 5. As can be seen, the whole architecture is directed by the control signal generated by a controller. The architecture includes three NTT-based multipliers that performing the modular polynomial multiplication over ring R_q, a Gaussian sampler to generate the error polynomials which in fact a binary uniform random distribution is used in hardware implementation, adders and modular reductions to compute the addition result, and some registers that enable the temporary storage of input and output data.

1) ENCRYPTION UNIT

The encryption unit is used to encrypt the input plaintext polynomial m with the public key $pk = (-a, b)$. Initially, all input polynomials are stored in the input registers for a level, and then the error polynomial r is multiplied by the public key $-a$ and b simultaneously using previous designed NTT-based multipliers Mult 1 and Mult 2. After Mult 1 and Mult 2 are multiplied, their multiplication results are added in parallel with $p \cdot e_1$ and $p \cdot e_2$ respectively. Moreover, the plaintext polynomial m can be further added to $p \cdot e_1 + b \times r$, and 27-bit additive modular operations are performed on the polynomials $m + p \cdot e_1 + b \times r$ and $p \cdot e_2 - a \times r$ in parallel which reduce the coefficients of the resulting polynomials to within R_q. Note that in our hardware implementation, instead of directly using DSPs to realize the multiplications of $p \cdot e_1$ and $p \cdot e_2$, they can be implemented by left-shifting each coefficient of polynomials p bits (for $p = 32$, is the power of 2), which can effectively reduce the occupation of DSPs and speed up the coefficient multiplications. At last, the ciphertext polynomials c_0 and c_1 are obtained after buffering a level.

2) DECRYPTION UNIT

Correspondingly, the decryption unit is used to recover the original plaintext polynomial m from the ciphertext polynomials $\tilde{c}_0$ and $\tilde{c}_1$, which may come from the output of encryption module directly or the output of KeySwitch module.

When the decryption unit is enabled by the controller, the multiplication between ciphertext polynomial $\tilde{c}_1$ and private key polynomial s is computed using the NTT-based multiplier Mult 3. Then the output of Mult 3 is added with the ciphertext $\tilde{c}_0$, and next an 27-bit additive modular reduction is performed on the addition result, which reduce the coefficient of the polynomial to the rang of $[0, q-1]$. However, according to the BGV scheme, in order to decrypted the ciphertext correctly, the coefficients of the polynomial need to be further reduced to $(-q/2, q/2)$. Hence, we add a conditional judgment using a MUX logic to reduce the coefficients of the polynomials to the correct range, that is, if the input coefficient is larger than $(q+1)/2$, then we subtract q from the coefficient as the output, otherwise, we directly output original coefficient. At last, a modulo p is performed on the output of MUX, due to modulus p equals to 32, which is the power of 2, we directly take the low p bits of the polynomial coefficient as the output of the modular reduction p. Similarly, the plaintext polynomial m can be obtained after a level of buffering. It should be noted that since all operations in our architecture are performed on signed numbers, more attention should be paid to the sign bits and specific values of the polynomial coefficients during the computation process. When necessary, we need to carry out the sign extension operations.

D. UNITS OF HOMOMORPHIC EVALUATION

In this section we describe the hardware architectures of homomorphic encryption to accelerate BGV.HomAdd, BGV.HomMult, BGV.KeySwtich and BGV.ModSwitch. Note that these functions are normally executed on the cloud server side and are the focus of our acceleration. Let ciphertext polynomials $\mathbf{ct_1} = (c_0, c_1)$ and $\mathbf{ct_2} = (c'_0, c'_1)$, BGV.HomAdd is used to add the two input ciphertext polynomials and output the sum of them $\mathbf{ct}_{HomAdd} = (c_0 + c'_0, c_1 + c'_1)$. BGV.HomMult is the most complicated function and it directly leads to dimension expansion and rapid noise growth of ciphertexts, which ultimately leads to the failure of decryption. To solve these two problems, the BGV.KeySwitch and BGV.ModSwitch are introduced. BGV.KeySwitch can reduce the dimension of ciphertexts from 3 dimensions back to 2 dimensions, while the BGV.ModSwitch can reduce the noise by diminishing the modulus, and still ensure the correctness of decryption.

1) HOMOMORPHIC ADDITION AND MULTIPLICATION

The architecture for homomorphic addition and multiplication of BGV is depicted in Figure 6. Initially, the ciphertext polynomials $\mathbf{ct_1}$ and $\mathbf{ct_2}$ are input to the MUXs, and are cached by the input registers. The homomorphic addition logic is relatively simple, the output of the registers directly perform the modular addition operations $\mathbf{ct}_{HomAdd} = (c_0 + c'_0, c_1 + c'_1)$ in parallel. If we need to perform homomorphic addition or homomorphic multiplication further, the results of modular addition will be written back to the corresponding MUX after storing in the output registers for a level.

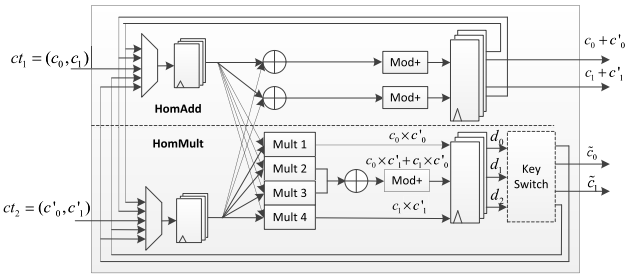


FIGURE 6. Architecture for BGV.HomAdd and BGV.HomMult.

Otherwise, the results of the modular addition will be directly output to the decryption unit to recover the addition of the two plaintext polynomials. Similarly, the output of input registers can simultaneously perform the homomorphic multiplication using the NTT-based Mult 1, Mult2, Mult3 and Mult4. For the output of the intermediate two multipliers, an addition modular operation is needed further. Finally, we can get the multiplication result $\mathbf{ct}_{HomMult} = (d_0, d_1, d_2) = (c_0c'_0, c_0c'_1 + c_1c'_0, c_1c'_1)$. However, since dimension of input polynomials and output polynomials of homomorphic multiplication are not matched, a KeySwitch module should be added further, which can transform the multiplicative ciphertext polynomials from (d_0, d_1, d_2) to $(\tilde{c}_0, \tilde{c}_1)$. The details of KeySwitch will be described in next subsection. The output ciphertext polynomials of KeySwitch can also be fed back to homomorphic addition and multiplication units to participate in the next round computation.

2) KEYSWITCH UNIT

KeySwitch technique is used to make a ciphertext decryptable with a different secret key homomorphically, it can reduce the dimension of homomorphic multiplication results at the cost of small noise growth. Concretely, when multiplying two ciphertexts, KeySwitch can transform the multiplicative ciphertext $\mathbf{ct}_{HomMult}$ with three components, which can be decrypted with $\mathbf{s}_{Mul} = (1, s', s'^2)$, back to a new ciphertext $\mathbf{ct}_{keyswitch}$ with two components that decryptable with secret key $\mathbf{s} = (1, s')$. KeySwitch includes the generation of switching key and switching key two functional parts. Generally, the former one is generated by the software in the key generation step, it takes an extended key (initially we have pk) and produces another extended key by adding in a so-called "key-switching matrix" from the s'^2 to s', to return a new extended key epk (i.e. switching key). The latter one is used to transform the ciphertext decryptable with s'^2 to a new ciphertext decryptable with s' using the previous switching key, and generate the ciphertext $\mathbf{ct}_{keyswitch}$. Excluding the generation of switching key, the KeySwitch algorithm is presented in Algorithm 4.

Given a homomorphic multiplication ciphertext polynomials $\mathbf{ct}_{HomMult} = (d_0, d_1, d_2) \in R_q^3$ and a switching key $epk = (\mathbf{rlk}_0, \mathbf{rlk}_1) = (-\mathbf{ex\_a}, \mathbf{ex\_b}) \in (R_q^\ell, R_q^\ell)$, where $\ell = \lfloor \log_t q \rfloor$ and t is decomposition base of ciphertext

Algorithm 4 KeySwitch Algorithm

Input: Multiplicative ciphertext $\mathbf{ct}_{HomMult} = (d_0, d_1, d_2) \in R_q^3$, switching key $epk = (\mathbf{rlk}_0, \mathbf{rlk}_1) = (-\mathbf{ex\_a}, \mathbf{ex\_b}) \in (R_q^\ell, R_q^\ell)$, where $\ell = \lfloor \log_t q \rfloor$ and t is decomposition base of d_2.
Output: KeySwitch ciphertext $\mathbf{ct}_{keyswitch} = (\tilde{c}_0, \tilde{c}_1)$.
//Decompose each element of d_2 in the base t.
1: **for** $(i = 0; j = 0; i \le \ell; i = i + 1)$ **do**
2: $\quad d_{2,i} = round(d_2/t)$
3: $\quad d_{2,j} = d_{2,j} - t \cdot d_{2,i}$
4: $\quad j = j + 1$
5: **end for**
//Sum the product of the components of switching key and d_2.
6: **for** $(i = 0; sum_1 = 0; i \le \ell; i = i + 1)$ **do**
7: $\quad sum_1 = sum_1 + Mult(\mathbf{rlk}_{1,i}, d_{2,i})$
8: $\quad sum_2 = sum_2 + Mult(\mathbf{rlk}_{0,i}, d_{2,i})$
9: **end for**
10: $\tilde{c}_0 = (d_0 + sum_1) \bmod q$
11: $\tilde{c}_1 = (d_1 + sum_2) \bmod q$
12: **Return**$(\mathbf{ct}_{keyswitch} = (\tilde{c}_0, \tilde{c}_1))$

d_2, we can write d_2 in base t (according to step 1∼ step 5 in Algorithms 4) as

$$d_2 = \sum_{i=0}^{\ell} d_{2,i} \cdot t^i \tag{8}$$

where $d_{2,i}$ is a polynomial with coefficients in $[0, t-1]$.Then, we can further output the KeySwitch ciphertext (according to step 6∼ step 11) as

$$\tilde{c}_0 = \left(d_0 + \sum_{i=0}^{\ell} \left(d_{2,i} \cdot \mathbf{rlk}_{1,i} \right) \right) \bmod q \tag{9}$$

$$\tilde{c}_1 = \left(d_1 + \sum_{i=0}^{\ell} \left(d_{2,i} \cdot \mathbf{rlk}_{0,i} \right) \right) \bmod q \tag{10}$$

where $epk = (\mathbf{rlk}_{0,i}, \mathbf{rlk}_{1,i})_0^\ell = \{(-ex\_a_i, ex\_b_i)\}_0^\ell$ is the switching key for the key s'^2. Note that the function *Mult* in step 7∼ step 8 represents the modular polynomial multiplication in R_q.

KeySwitch is another most computationally intensive operation in BGV scheme. In order to reduce the number of for-loops in step 1 and step 6 of Algorithm 4, we set the key switching parameter t to 2^{13} in out implementation, so $\ell = \lfloor \log_t q \rfloor = 1$ and the number of components of switching key epk is equal to 2, i.e. $\mathbf{rlk}_0 = \{-ex\_a_0, -ex\_a_1\}$ and $\mathbf{rlk}_1 = \{ex\_b_0, ex\_b_1\}$. A parallel processing architecture for KeySwitch is shown in Figure 7. Since the parameter t equals to the exponents of 2, the decomposition of ciphertext component d_2 can be simply realized by dividing d_2 into low 13 bits and high 14 bits respectively. When performing the KeySwitch computation, the low 13 bits of d_2 should be padded with 14 bits 0 in the most significant position to obtain a 27-bit component $d_{2,0}$. Similarly, the high 14 bits of d_2 are

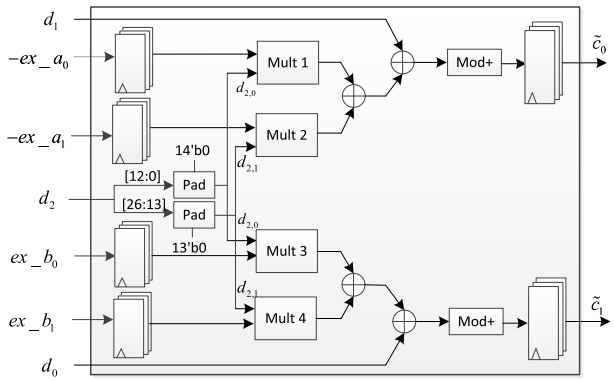


FIGURE 7. Architecture for KeySwitch module.

also padded with 13 bits 0 in the most significant position to get another component $d_{2,1}$. Next, the switching keys $\mathbf{rlk}_0 = \{-ex\_a_0, -ex\_a_1\}$ are simultaneously multiplied by decomposition components $d_{2,0}$ and $d_{2,1}$ using the polynomial multiplier Mult 1 and Mult 2 respectively, and the sum of them further participates in the following modular addition operation, and finally the KeySwitch ciphertext component $\tilde{c}_0$ is obtained. Meanwhile, the products of switching keys $\mathbf{rlk}_1 = \{ex\_b_0, ex\_b_1\}$ and components $d_{2,0}$ and $d_{2,1}$ are also performed in parallel using the polynomial multiplier Mult 3 and Mult 4 respectively, then the other ciphertext component $\tilde{c}_1$ is calculated by modular addition of the products.

Due to adopting the full pipeline and parallel processing architecture, and the polynomial multipliers have been optimized, the overall performance of KeySwitch module can be improved greatly. However, when KeySwitch reduces the dimension of multiplicative ciphertext, it also brings some additional noise, which may lead to the failure of decryption. Hence, we need to consider how to reduce the KeySwitch noise further to ensure the correctness of decryption.

3) MODSWITCH UNIT

ModSwitch (also known as modulus switching) gives us a very powerful and lightweight way to manage the noise in BGV scheme. This technique permits the evaluator to reduce the magnitude of the noise in a ciphertext by scaling down the ciphertext, and without knowing the secret key. More specifically, suppose $\mathbf{c}$ is a valid encryption of m under secret key $\mathbf{s}$ modulo Q, and $\mathbf{c}'$ is a simply scaling of $\mathbf{c}$, which is closest to $(q/Q)\mathbf{c}$ such that $\mathbf{c}' = \mathbf{c} \bmod 2$. If $\mathbf{s}$ is a short vector and q is sufficiently smaller than Q, it can be proved that $\mathbf{c}'$ is a valid encryption of m under secret key $\mathbf{s}$ modulo q. In other words, we can reduce the noise of ciphertext $\mathbf{c}$ by transforming $\mathbf{c}$ modulo Q into a smaller ciphertext $\mathbf{c}'$ modulo q while preserving the correctness under the same secret key.

As mentioned before, if the noise of ciphertext generated by key switching grows too fast, we can choose to reduce the ciphertext noise using the ModSwitch for further increasing the depth of multiplication circuit. Compared with the previous works, one of the main contributions of our work is to design a ModSwitch algorithm and hardware architecture for

plaintext modulus $p \neq 2$(p equals to 32 in our scheme). The ModSwitch algorithm we proposed is shown in Algorithm 5. Suppose $\mathbf{ct}_{keyswitch} = (\tilde{c}_0, \tilde{c}_1)$ is the KeySwith ciphertext with modulus q_l at level l (initially, $q_l = q$ and $l = 1$), and q_l is the product of primes satisfying $q_l = \prod_{j=0}^{l} p_j$ for $l = 0$ to $L-1$, where $p_j \equiv 1(\mathrm{mod}p)$, L is the system parameter. Suppose the $q_{l'}$ is the modulus of ModSwitch ciphertext at level l', where $l' > l$, and the scaling factor $\Delta = q_l/q_{l'}$ satisfying $q_{l'} < q_l$. Then, we first perform modulo Δ reduction and modulo p reduction respectively for $(\tilde{c}_0, \tilde{c}_1)$, and in order to ensure the coefficients in $\delta_{\tilde{c}_0}$ and $\delta_{\tilde{c}_1}$ is divisible by p, we further subtract $\delta'_{\tilde{c}_0}$ and $\delta'_{\tilde{c}_1}$ from each coefficient of $\delta_{\tilde{c}_0}$ and $\delta_{\tilde{c}_1}$ respectively (as shown in step 3∼ Step 17). After fixing the coefficients of $\delta_{\tilde{c}_0}$(s.t. $\delta_{\tilde{c}_0} \equiv \tilde{c}_0(\mathrm{mod}\Delta)$ and $\delta_{\tilde{c}_0} \equiv 0(\mathrm{mod}p)$) and $\delta_{\tilde{c}_1}$(s.t. $\delta_{\tilde{c}_1} \equiv \tilde{c}_1(\mathrm{mod}\Delta)$ and $\delta_{\tilde{c}_1} \equiv 0(\mathrm{mod}p)$), the ModSwitch ciphertext $(\tilde{c}'_0, \tilde{c}'_1)$ is obtained by performing $\tilde{c}'_0 = floor((\tilde{c}_0 - \delta_{\tilde{c}_0})/\Delta)$ and $\tilde{c}'_1 = floor((\tilde{c}_1 - \delta_{\tilde{c}_1})/\Delta)$ respectively.

Algorithm 5 ModSwitch Algorithm

Input: KeySwitch ciphertext $\mathbf{ct}_{keyswitch} = (\tilde{c}_0, \tilde{c}_1)$ with modulus q_l at level, q_l is the product of primes s.t. $q_l = \prod_{j=0}^{l} p_j$ for $l = 0$ to $L-1$, where $p_j \equiv 1(\mathrm{mod}p)$, the modulus $q_{l'}$ of ModSwitch ciphertext at level l' (s.t. $l' > l$), and scaling factor $\Delta = q_l/q_{l'}$(s.t. $q_{l'} < q_l$).
Output: ModSwitch ciphertext $\mathbf{ct}_{\mathrm{mod}\ swith} = (\tilde{c}'_0, \tilde{c}'_1)$.
1: $\delta_{\tilde{c}_0} = \tilde{c}_0(\mathrm{mod}\Delta)$, $\delta'_{\tilde{c}_0} = \delta_{\tilde{c}_0}(\mathrm{mod}p)$
2: $\delta_{\tilde{c}_1} = \tilde{c}_1(\mathrm{mod}\Delta)$, $\delta'_{\tilde{c}_1} = \delta_{\tilde{c}_1}(\mathrm{mod}p)$
//Fix $\delta_{\tilde{c}_0}$ s.t. $\delta_{\tilde{c}_0} \equiv \tilde{c}_0(\mathrm{mod}\Delta)$ and $\delta_{\tilde{c}_0} \equiv 0(\mathrm{mod}p)$
3: **for** $i = 0$ to $n-1$ **do**
4: **if** $\delta'_{\tilde{c}_{0,i}} = 0$ **then**
5: $\delta_{\tilde{c}_{0,i}} = \delta_{\tilde{c}_{0,i}}$
6: **else**
7: $\delta_{\tilde{c}_{0,i}} = \delta_{\tilde{c}_{0,i}} - \delta'_{\tilde{c}_{0,i}} \cdot \Delta$
8: **end if**
9: **end for**
//Fix $\delta_{\tilde{c}_1}$ s.t. $\delta_{\tilde{c}_1} \equiv \tilde{c}_1(\mathrm{mod}\Delta)$ and $\delta_{\tilde{c}_1} \equiv 0(\mathrm{mod}p)$
10: **for** $i = 0$ to $n-1$ **do**
11: **if** $\delta'_{\tilde{c}_{1,i}} = 0$ **then**
12: $\delta_{\tilde{c}_{1,i}} = \delta_{\tilde{c}_{1,i}}$
13: **else**
14: $\delta_{\tilde{c}_{1,i}} = \delta_{\tilde{c}_{1,i}} - \delta'_{\tilde{c}_{1,i}} \cdot \Delta$
15: **end if**
17: **end for**
18: $\tilde{c}'_0 = floor((\tilde{c}_0 - \delta_{\tilde{c}_0})/\Delta)$
19: $\tilde{c}'_1 = floor((\tilde{c}_1 - \delta_{\tilde{c}_1})/\Delta)$
20: **Return**$(\mathbf{ct}_{\mathrm{mod}\ swith} = (\tilde{c}'_0, \tilde{c}'_1))$

According to the Algorithm 5, we further propose the architecture of ModSwitch as shown in Figure 8. In our implementation, the initial ciphertext modulus $q_l = q = 257^3 = 16974593$, the ModSwitch ciphertext modulus $q_{l'} = 257^2 = 66049$, and the scaling factor $\Delta = 257$. Since there is no data dependency between ciphertext $\tilde{c}'_0$ and $\tilde{c}'_1$

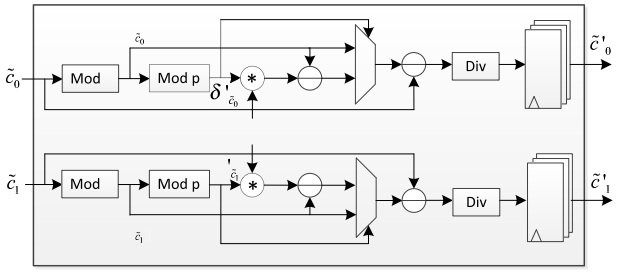


FIGURE 8. Architecture for ModSwitch module.

during the modulus switching process, we can perform the ModSwitch operations of them completely in parallel. When ciphertexts $\tilde{c}_0$ and $\tilde{c}_0$ comes, we first perform the modulo Δ reduction operation and modulo p reduction operation respectively (step 1∼step 2 in Algorithm 5). Since modulus p is the power of 2, its modular reduction can be simplified by directly taking the lower 5 bits. Next, the results of modulo Δ (namely $\delta_{\tilde{c}_0}$ and $\delta_{\tilde{c}_1}$) subtract the product of Δ and the results of modulo p (namely $\delta'_{\tilde{c}_0}$ and $\delta'_{\tilde{c}_1}$). The MUXs are used to select output the results of modulo Δ or the results of subtraction (step 3∼Step 17 in Algorithm 5). Then, the subtractions $\tilde{c}_0 - \delta_{\tilde{c}_0}$ and $\tilde{c}_1 - \delta_{\tilde{c}_1}$ are performed on the output of MUXs respectively. At last, we can get the ModSwitch cipertexts $(\tilde{c}'_0, \tilde{c}'_1)$ by performing the division Δ operations on the previous subtraction results. The division operation can be directly realized by IP Core. Noted that although we present the hardware architecture of ModSwitch unit, it is not connected in the overall accelerator at last, mainly because the ciphertext modulus of the NTT-based multiplier we designed is fixed and it can no longer be applied to the ModSwitch result.

E. GENERALIZATION AND DISCUSSSION

Although our FPGA-based high parallelism architecture mainly focuses on the BGV FHE scheme, it is worth noting that the architectures of kernel acceleration unit (including polynomial multiplication unit and modular reduction unit), KeySwitch unit and ModSwitch unit are also perfectly suitable for other leveled Ring-LWE homomorphic encryption algorithms such as FV [12], [31], [32] and YASHE [17], [28] etc. Furthermore, the homomorphic encryption unit and homomophic evaluation unit can be applied to other Ring-LWE FHE schemes well with minor modification of multiplication factors, the sign of polynomials or some key parameters. For example, when performing encryption algorithm, we only need to multiply the plaintext polynomial by $\Delta = \lfloor q/t \rfloor$, and multiply the ciphertext by $\delta = t/q$ additionally when performing decryption algorithm and homomorphic multiplication, then our homomorphic encryption accelerator can be fully applicable to FV and YASHE algorithms. On the other hand, although we use a small parameter set with the polynomial degree $n = 128$ and 27-bit ciphertext modulus, but our hardware accelerator still supports larger parameter set just by increasing computation

cycles of polynomial multipliers and minor modifying the architecture of modular reduction unit. Therefore, the Ring-LWE accelerator we proposed has high-level generalization ability for different application scenarios.

V. IMPLEMENTATION RESULTS

A. RESOURCE CONSUMPTION

The proposed hardware accelerator for Ring-LWE based BGV scheme is descripted with Verilog HDL language, synthesized and implemented in Xilinx VIVADO on a Virtex UltraScale FPGA platform, which has a chip XCVU125-FLVA2014-1HV-E. We evaluate our design on a small range of parameters: the size (n) of the ciphertext polynomial is 128, the coefficients of ciphertext polynomial are 27-bit signed integers with the ciphertext modulus (q). However, from the perspective of hardware implementation and the application scenarios of packing technology, we set the plaintext modulus (p) as a 5-bit number, which means that the sum or product of the coefficients of two plaintext polynomials cannot exceed p, otherwise, folding will occur. We elaborate in detail on the resource consumption of each component of our design from the basic modules to the whole accelerator as shown in Table 3.

TABLE 3. Resource consumption of basic modules.

Module	LUTs	Registers	BRAMs	DSPs
NTT	14204	3756	2	2
Multiplier	43731	11256	2	15
MR+ (27-bit)	108	0	0	0
MR× (54-bit)	338	0	0	0
Encryption	95854	22604	5.5	30
Decryption	47793	11213	2	15
HomMult (HomAdd)	190529	44974	8	60
KeySwitch	190539	44975	8	60
ModSwitch	1191	630	0	2
Complete Design	527493	133813	23.5	165

As can be seen, since NTT based multiplier occupies three identical NTT transformations, two of which are used for input polynomials and one for output polynomial, the resource consumption of the multiplier is about 3 times that of NTT transformation. Further analysis shows that the multiplier is the primary part of the accelerator resource overhead, which consumes a total of 11 multipliers. Specifically, the encryption module and decryption module occupy two multipliers and one multiplier, while homomorphic multiplication and KeySwitch module consume four multipliers respectively. Therefore, the area overhead of these modules is approximately a multiple of the number of multipliers consumed, e.g. the LUTs/ Registers/ BRAM/ DSPs of KeySwitch module are almost 4 times that of NTT based multiplier. Note that the 27-bit modular reduction (MR) and 54-bit modular reduction represents the modular operation for addition and multiplication, the architecture of which are implemented by combinational logic circuits. The LUTs and registers consumed by ModSwitch are slightly large, this is mainly due to the use of two divider IP Cores, each of which occupies 495 LUTs and 315 registers. The Block RAM (BRAM) in our implementation represents the on-chip memories for fast reading and writing operation, and can be used to realize the dual-port RAMs or read-only ROMs. The BRAM consists of RAMB36 units in our design can hold 128-many 27-bit values. The Digital Signal Processor (DSP) is capable to perform the 27-bit coefficient multiplications using DSP48E2. Finally, the resource consumption of complete design (excluding that of ModSwitch module) is given in the last row of the Table 3.

B. PERFORMANCE EVALUATION

In our implementation, all coefficients of polynomials are input to each component of the accelerator in serial, and the intermediate results of computation are temporarily stored by BRAMs or registers to maintain high-speed pipeline processing. The operating clock frequency directly affects the performance of accelerator, in order to reduce the time delay of critical path of our design, we have eliminated some critical paths during many design iterations by altering the data flow of computation, minimizing the number of logic circuits per pipeline stage, etc. Finally, our accelerator can run at 150MHz on Virtex UltraScale FPGA. In Table 4 the performance of basic operations are presented.

TABLE 4. Performance of basic operations.

Operation	Speed		
	(# cycles)	(# μ seconds) (Non-pipeline)	(# μ seconds) (Full-pipeline)
NTT	1153	7.69	6.84
Multiplier	2180	14.54	
Encryption	2182	14.55	
Decryption	2181	14.54	
HomAdd	128	0.85	0.85
HomMult	2181	14.54	6.84
KeySwitch	2181	14.54	
ModSwitch	315	2.10	2.10
HomAdd_Enc_Dec	4235	28.24	6.84
HomMult_Enc_Dec	8341	55.63	

Similarly, the NTT based multiplier is still the most significant unit affecting the performance of the accelerator. A single NTT transformation takes 1153 cycles to process 128 coefficients in serial using four stages of finite state machine. At 150MHz, this corresponds to 7.69 μs. Since the multiplier employs two NTT transformation (which are used as NTT and INTT), it consumes approximately twice as many cycles as that of NTT. However, if the pipeline is full, the speed of multiplier will be reduced to 6.84 μs. For the same reason, the encryption, decryption, homomorphic

multiplication and KeySwitch module all adopt a single level of multiplier respectively, thus, the clock overhead is the same as that of a multiplier. Since the homomoprhic addition is a combinational logic circuit, the addition operation can be performed with the following register storing operation, which only occupies very few clocks. Then, the performance of homomorphic addition and multiplication from the encryption module to decryption module are evaluated. For a single set of polynomial inputs, there are total 4235 cycles and 8341 cycles are spent on the HomAdd_Enc_Dec and HomMult_Enc_Dec, which correspond to 28.24 μs and 55.63 μs respectively. If the pipeline is fully, the cycles of HomAdd_Enc_Dec and HomMult_Enc_Dec will be reduced to 1025, corresponding to 6.84 μs respectively.

C. COMPARISON WITH RELATED WORKS

Firstly, we compare our 54-bit LUT-based modular reduction unit with the Barrett, pseudo-Fermat primes and straight forward modular reduction in Reference [37] as shown in Table 5. A 64-bit input value with 31-bit modulus q (which is equal to 0×439.0001) is chosen as the input parameter in Reference [37], as can be seen, though the bit width of input value and modulus in our design is slightly smaller, the LUTs consumed by our modular reduction is still about 10 times less than the Barrett algorithm, which has the lowest resource overhead in Reference [37]. For a fair comparison, we also refer to the implementation method of Barrett in Reference [43] and design an improved Barrett modular reduction unit with the same parameters as our LUT-based modular reduction. Still, the LUT cost of our design is about 2 times less than the improved Barrett method in Reference [43] under the same condition.

TABLE 5. Comparison of modular reduction.

Ref	Input	q	LUTs	Regs	BRAMs	DSPs	Cycles
Barrett [37]	64bit	31bit	3176	4729	0	8	108
Fermat [37]	64bit	31bit	15956	41363	0	345	119
Modulus [37]	64bit	31bit	5701	6226	50	16	83
Barrett [43]	54bit	27bit	872	0	0	6	-
Our Design	54bit	27bit	338	0	0	0	-

Secondly, the resource consumption and performance of polynomial multipliers are compared. Because there are differences in the choice of the parameters and implementation platforms, a totally fair comparison between the different implementations is not always possible. Hence, we compare our polynomial multiplier with related works from the perspective of throughput and normalized efficiency as shown in Table 6. In Reference [37], a Pease's polynomial multiplier and a Cooley-Tukey's polynomial multiplier for BGV

TABLE 6. Comparison of modular reduction.

Ref	Pease's [37]	Cooley's [37]	Strassen's [44]	NTT [45]	Karat [46]	Our Design
Device	Stratix V	Stratix V	Zynq UltraScale	Zynq -7000	Virtex -7	Virtex UltraScale
N	32768	32768	1024	128	512	128
Bit (q)	192	64	31	14	32	27
LUTs	187664	145381	5277	66251	323698	43731
Regs	481056	343836	5454	16805	373841	11256
BRAMs	4398	1402	18	3.5	769	2
DSPs	3270	1494	112	26	3072	15
Freq (MHz)	180	100	250	100	234	150
Speed (µs)	23518	2029	459	23.04	5.19	6.84
TP † (Mbps)	267.52	1033.5	69.15	77.78	3156.8	505.26
Speedup -TP	×1.89	<	×7.31	×6.49	<	1
NE †† (Kbps/ LUT)	1.43	7.10	13.23	1.17	9.75	11.55
Speedup -NE	×8.07	×1.62	<	×9.87	×1.18	1

† Throughput (TP) = N × (No. of bits (q))/Speed.
†† Normalized Efficiency (NE) = Throughput/No. of LUTs.

homomorphic encryption are proposed with the polynomial length $N = 32768$, the bits $q = 192$ and $q = 64$ respectively. A pipelined and loop unrolled Schoenhage-Strassen FFT polynomial multiplier ($N = 1024$, $q = 31$ bits) is presented in Reference [44]. Reference [45] describes an open-source NTT-based polynomial multiplier ($N = 128, q = 14$ bits) FPGA implementation for post-quantum cryptographic primitives. At last, an optimized Karatsuba-based multiplier is proposed in Reference [46].

In order to improve the processing speed and throughput of multiplier, we adopt the multi-level pipeline structure in the horizontal direction and the parallel processing structure of NTT in the vertical direction. When the pipeline is full, the speed of the proposed multiplier is about 6.84 μs, and the corresponding through is 505.26 Mbps. The throughput of our design can achieve a 1.89∼7.31 times speedup when compared with other works, except for the Cooley-Tukey's multiplier in [37] and Karatsuba-based multiplier in [46]. This is mainly due to the fact that Cooley-Tukey's multiplier [37] adopts a ''ping-pong'' BRAM structure with higher performance and two parallel butterflies at the cost of area, while Karatsuba-based multiplier [46] has a lower algorithm complexity. However, in terms of normalized efficiency, our multiplier still has the advantages of 1.62 times and

1.18 times compared with Cooley-Tukey's multiplier [37] and Karatsuba-based multiplier [46]. In addition, though our efficiency is slightly lower than Schoenhage-Strassen multilier in [44], we can achieve more than 7× throughput. Therefore, if the pipeline is full, the proposed NTT based multiplier will have great advantages in performance and normalized efficiency compared with state of arts. However, if the pipeline is not full, the speed of the proposed multiplier will be between 6.84 μs and 14.54 μs, the performance and efficiency of our multiplier will be reduced by half at maximum, and the advantages of our parallel and pipeline structure cannot be fully utilized. At last, it is worth noting that the resource occupation and performance of the reference [37] in Table 6, which uses Stratix V as the FPGA device, are directly indexed from the original literature, so they can be used as the performance comparison without resource occupation conversion. If it is necessary to consider the impact on the time cost due to different technologies of different devices, we can normalize the throughput by dividing the throughput indexes by the clock frequency. The results show that the speedup ratio is further increased by 1.2 times compared to Pease's implementation [37], while compared to Cooley-Tukey's implementation [37] the speedup ratio still maintains a certain advantage.

Lastly, we compare the performance of our accelerator with the similar works as shown in Table 7. Since the implementations of the BGV scheme are limited in the literature, in addition to comparing with the performance of the existing BGV software and hardware implementations in [37], we also compare our accelerator with the FV implementation in [31], which is the most similar Ring-LWE FHE scheme to BGV. As discussed in Subsection E of Section IV, the BGV scheme can be easily extended to FV scheme with minor modifications, so it is reasonable to compare our BGV accelerator with the FV implementation.

In Reference [37], they proposed a typical software implementation on general purpose computer, and further presented two hardware implementations of BGV homomorphic encryption accelearator based on Pease's multiplier and Cooley-Tukey's multiplier respectively. In order to improve the performance of accelerator, they not only use the negative wrapped convolution to speed up the NTT-based polynomial multiplier, but also use the Chinese Remainder Theorem (CRT) to optimize the polynomial multiplication on a larger ciphertext space. For the parameter set with polynomial size of 32768 and the ciphertext space 1088 bits, the speed of software implementation is about 670 ms and 324ms for the encryption and decryption algorithms, while the Pease's implementation and Cooley-Tukey's implementation require 327 ms and 166ms for encryption algorithm, 53ms and 73 ms for decryption algorithm respectively. As mentioned previously, if there are multiple sets of input polynomials and the pipeline is full, the processing speed of our design is up to 6.84 μs for encryption and decryption algorithm, and the throughput is 505.26 Mbps, which is about 9.49 times and 4.60 times larger than that of software implementation [37], and improves about 4.64 times and 2.17 times compared to the Pease's implementation [37]. When compared to the Cooley-Tukey's implementation [37], though our design can achieve 2.36 times improvement for the encryption algorithm, while the throughput of the decryption algorithm is increased by 1.03 times.

Roy *et al.* [31] introduced an FPGA-based multicore processor HEPCloud for FV somewhat homomorphic function evaluation, to efficiently implement the homomorphic addition and homomorphic multiplication of FV scheme, they simplify the modular reduction by lifting a polynomial in R_q to the ring R_Q with larger modulus Q, and scaling back to the ring R_q when the computations are completed. They report the computation time of homomorphic addition and homomorphic multiplication is about 0.05s and 26.67s respectively due to the slow memory access. The throughput of our design is improved about 5.05 times and 167.30 times for homomorphic addition and homomorphic multiplication evaluations. In terms of normalized efficiency, although the throughput per LUT of our homomorphic addition is slightly less than that of [31], but the normalized efficiency of homomorphic multiplication of our design is still increased by 31.6 times.

Furthermore, if the pipeline is not full, the latency of homomorphic encryption and homomorphic evaluation will be increased to a maximum of 14.55 μs and the throughput of our implementation will be reduced accordingly. However, our performance is still better than the previous works.

Finally, it is noted that although Table 7 lists two different FPGA devices (i.e., Stratix and Virtex) which belong to different companies, when comparing resource overhead and performance, reference [37] only provides the time cost of encryption and decryption, and does not provide the resource consumption. Therefore, different FPGA platforms and devices have no impact on the resource overhead comparison. If it is necessary to consider the impact on the time cost due to different technologies for different devices, the throughput in Table 7 can be divided by clock frequency to eliminate the impact of clock frequency. In this case, the speedup ratios are reduced by 2/3 times. It can be found that the performance of our accelerator still has several to dozens of times advantage, except for the decryption unit and homomorphic addition unit.

VI. SECURITY DISCUSSION

The security of our accelerator includes the security of the homomorphic encryption algorithm and the security of FPGA-based hardware accelerator architecture two parts.

From the perspective of proof-of-concept, we use a small parameter set with the polynomial degree $n = 128$ and 27-bit ciphertext modulus, so the security level of our design is slightly less than 128 bits. However, as we discussed in subsection E of Section IV, our hardware accelerator can be easily extended to larger polynomial degrees to support higher security levels with only minor modifications of the computation cycles and the structure of modular reduction. For example, when the polynomial degree $n = 1024$ and

TABLE 7. Comparison of accelerator.

Reference	CPU_{CRT} [37]	$FPGA_{Pease's}$ [37]	$FPGA_{Cooley's}$ [37]	FV [31]	Our Design
Algorithm	BGV (RLWE)	BGV (RLWE)	BGV (RLWE)	FV (RLWE)	BGV (RLWE)
Device	Core i7	Stratix V	Stratix V	Virtext-6	Virtex UltraScale
N	32768	32768	32768	32768	128
Bit(q)	1088	1088	1088	1228	27
LUTs	-	-	-	72613	527493 (HomEval-381068) ‡‡‡
Registers	-	-	-	63086	133813 (HomEval-89949) ‡‡‡
BRAMs	-	-	-	84	23.5 (HomEval-16) ‡‡‡
DSPs	-	-	-	250	165 (HomEval-120) ‡‡‡
Freq (MHz)	3.10GHz	100	100	100	150
HomAdd (μs)	-	-	-	5×10^4	0.85
HomMult (μs)	-	-	-	26.67×10^6	6.84
Enc (μs)	669687	327145	166291	-	6.84
Dec (μs)	324449	153178	72751	-	6.84
TP-1 ‡ (Mbps)	53.24 (Enc)	108.98 (Enc)	214.39 (Enc)	804.78 (HomAdd)	4065.88 (HomAdd) 505.26 (Enc)
TP-2 ‡ (Mbps)	109.88 (Dec)	232.75 (Dec)	490.04 (Dec)	3.02 (HomMult)	505.26 (HomMult) 505.26 (Dec)
Speedup-TP-1	×9.49 (Enc)	×4.64 (Enc)	×2.36 (Enc)	×5.05 (HomAdd)	1
Speedup-TP-2	×4.60 (Dec)	×2.17 (Dec)	×1.03 (Dec)	×167.30 (HomMut)	1
NE-1 ‡‡ (Kbps/LUT)	-	-	-	11.35 (HomAdd)	10.93
NE-2 ‡‡ (Kbps/LUT)	-	-	-	0.043 (HomMult)	1.358
Speedup-NE-1	-	-	-	<	1
Speedup-NE-2	-	-	-	31.6	1

‡ Throughput (TP-1, TP-2) = N × (No. of bits (q))/Speed of HomAdd, HomMult, Enc, or Dec.
‡‡ Normalized Efficiency (NE) = Throughput/No. of LUTs.
‡‡‡ Only the resource consumption of homomorphic evaluation units are listed in brackets.

ciphertext modulus $\log_2 q = 27$, the security level of homomorphic encryption algorithm will equal to 128 bits [47]. Meanwhile the security of our BGV scheme is based on the Ring-LWE assumption, which is reducible to worst-case problems on ideal lattices, can ensure our accelerator and the FHE algorithm resistant the attacks of future quantum computer.

On the other hand, the security of our FPGA-based hardware accelerator is mainly guaranteed by the FPGA platform and homomorphic encryption algorithm. The classical method to reverse engineer a chip is the black box attack [48], the attacker inputs all possible combinations, while saving the corresponding outputs. Due to the complexity of our design and the size of our state-of-the-art FPGA platform, it is infeasible to extract the inner logic of our accelerator without a lot of powerful computers. Furthermore, the nature of our FHE algorithm also prevents the attack as well. Readback attack [49] is another conventional attack

of FPGA implementation, the idea of the attack is read the configuration of the FPGA through the programming interface or JTAG to obtain private keys or FHE algorithms. The readback attack can be prevented by setting a security bit in FPGA which is used to disable different features, and it is better to embed our FPGA-based accelerator into a secure environment, where the configuration information can be deleted once detecting interference. In order to get the private keys or the FHE algorithms, one has to reverse-engineer the bitstream [50]. FPGA manufactures claim that the security of the bitstream relies on the disclosure of the layout of the configuration information. Hence, the encryption of the configuration file is the most effective and practical countermeasure, it not only prevents the reverse-engineering attack, but also the cloning of SRAM FPGAs. Although we have listed some possible conventional attacks and countermeasures for our FPGA-based hardware implementation, but with the development of the FHE algorithm and its acceleration technology, other attack methods and protection strategies for FHE will emerge in endlessly, and the security discussion of the FPGA-based hardware architecture is a separate and complex problem, so this paper will not further discuss in detail due to the length limitation.

VII. CONCLUSION

This paper focuses on the FPGA hardware implementation for Ring-LWE based leveled fully homomorphic encryption. We present a hardware implementation of BGV scheme that implements all components required for homomorphic encryption and homomorphic evaluation. Our architecture provides a trade-off between the hardware cost and performance. To accelerate the computational intensive operations of homomorphic encryption functions, we put forward an iterative NTT-based modular polynomial multiplier with high performance and a self-designed LUT-based modular reduction unit with less resource consumption. On this basis, we accelerate the each functional component of hommorphic encryption and homomorphic evaluation from the functions of client and server. In particular, we implement the homomorphic evaluation module including modulus switching of BGV scheme for the first time. Finally, we evaluate the resource consumption and performance of our implementation on Virtex UltraScale FPGA platform. We find that our modular reduction can save at least 2 times area, and our polynomial multiplier has at least 20% higher normalized efficiency when compared to existing implementations. Besides, we demonstrate that the performance of our overall architecture is also optimal, at the cost of slightly larger resource occupation. As for future work, we will extend our design implementation to the multicore application scenarios and support wider range of parameters.

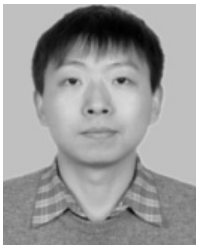
REFERENCES

[1] C. Gentry and D. Boneh, *A Fully Homomorphic Encryption Scheme*, no. 9. Stanford, CA, USA: Stanford Univ. Stanford, 2009.
[2] M. Kim, Y. Song, and J. H. Cheon, "Secure searching of biomarkers through hybrid homomorphic encryption scheme," *BMC Med. Genomics*, vol. 10, no. S2, p. 42, Jul. 2017, doi: 10.1186/s12920-017-0280-3.
[3] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino, "Single-database private information retrieval from fully homomorphic encryption," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 5, pp. 1125–1134, May 2013.
[4] J. W. Bos, W. Castryck, I. Iliashenko, and F. Vercauteren, "Privacy-friendly forecasting for the smart grid using homomorphic encryption and the group method of data handling," in *Proc. Int. Conf. Cryptol. Afr.* Dakar, Senegal: Springer, 2017, pp. 184–201.
[5] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 201–210.
[6] R. Xu, J. B. D. Joshi, and C. Li, "CryptoNN: Training neural networks over encrypted data," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1199–1209.
[7] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secure Comput.*, vol. 4, no. 11, pp. 169–180, 1978.
[8] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Symp. Theory Comput. STOC*, 2009, pp. 169–178.
[9] 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.* Riviera, France: Springer, 2010, pp. 24–43.
[10] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM J. Comput.*, vol. 43, no. 2, pp. 831–871, 2014.
[11] 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.
[12] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, Mar. 2012.
[13] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. Annu. Cryptol. Conf.* Santa Barbara, CA, USA: Springer, 2013, pp. 75–92.
[14] L. Ducas and D. Micciancio, "FHEW: Bootstrapping homomorphic encryption in less than a second," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Sofia, Bulgaria: Springer, 2015, pp. 617–640.
[15] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, Jan. 2020.
[16] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Proc. Annu. Cryptol. Conf.* Santa Barbara, CA, USA: Springer, 2012, pp. 850–867.
[17] T. Lepoint and M. Naehrig, "A comparison of the homomorphic encryption schemes FV and YASHE," in *Proc. Int. Conf. Cryptol. Afr.* Marrakesh, Morocco: Springer, 2014, pp. 318–335.
[18] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Accelerating fully homomorphic encryption using GPU," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, Sep. 2012, pp. 1–5.
[19] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Tallinn, Estonia: Springer, 2011, pp. 129–148.
[20] W. Wang, Z. Chen, and X. Huang, "Accelerating leveled fully homomorphic encryption using GPU," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Jun. 2014, pp. 2800–2803.
[21] A. Qaisar Ahmad Al Badawi, Y. Polyakov, K. M. M. Aung, B. Veeravalli, and K. Rohloff, "Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme," *IEEE Trans. Emerg. Topics Comput.*, early access, Mar. 4, 2019, doi: 10.1109/TETC.2019.2902799.
[22] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Accelerating fully homomorphic encryption over the integers with super-size hardware multiplier and modular reduction," *IACR Cryptol. ePrint Arch.*, vol. 2013, p. 616, Sep. 2013.
[23] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan, "High-speed fully homomorphic encryption over the integers," in *Proc. Int. Conf. Financial Cryptogr. Data Secur.* Christ Church, Barbados: Springer, 2014, pp. 169–180.
[24] W. Wang and X. Huang, "FPGA implementation of a large-number multiplier for fully homomorphic encryption," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2013, pp. 2589–2592.
[25] W. Wang, X. Huang, N. Emmart, and C. Weems, "VLSI design of a large-number multiplier for fully homomorphic encryption," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 9, pp. 1879–1887, Sep. 2014.

[26] Y. Doroz, E. Ozturk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *Proc. Euromicro Conf. Digit. Syst. Design*, Sep. 2013, pp. 955–962.
[27] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *IEEE Trans. Comput.*, vol. 64, no. 6, pp. 1509–1521, Jun. 2015.
[28] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Saint-Malo, France: Springer, 2015, pp. 143–163.
[29] Y. Doröz, E. Öztürk, E. Savaş, and B. Sunar, "Accelerating LTV based homomorphic encryption in reconfigurable hardware," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, Springer, 2015, pp. 185–204.
[30] S. S. Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Saint-Malo, France: Springer, 2015, pp. 164–184.
[31] S. Sinha 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.
[32] 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.
[33] M. Sadegh Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," 2019, *arXiv:1909.09731*. [Online]. Available: http://arxiv.org/abs/1909.09731
[34] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, vol. 275, pp. 1072–1086, Jan. 2018.
[35] R. Karthick, A. M. Prabaharan, P. Selvaprasanth, N. Sathiyanathan, and A. Nagaraj, "High resolution image scaling using fuzzy based FPGA implementation," *Asian J. Appl. Sci. Technol. (AJAST)*, vol. 3, no. 1, pp. 215–221, 2019.
[36] E. Öztürk, Y. Doröz, B. Sunar, and E. Savas, "Accelerating somewhat homomorphic evaluation using FPGAs," *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 294, Mar. 2015.
[37] A. R. Pedrosa, "Implementing fully homomorphic encryption schemes in FPGA-based systems," E.T.S. de Ingenieros Informáticos (UPM), Madrid, Spain, Tech. Rep., Jan. 2016. [Online]. Available: http://oa.upm.es/39925/
[38] 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* Oxford, U.K.: Springer, 2013, pp. 45–64.
[39] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proc. 44th Symp. Theory Comput. STOC*, 2012, pp. 1219–1234.
[40] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba, "New packing method in somewhat homomorphic encryption and its applications," *Secur. Commun. Netw.*, vol. 8, no. 13, pp. 2194–2213, Sep. 2015.
[41] C. Du and G. Bai, "Towards efficient polynomial multiplication for lattice-based cryptography," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 1178–1181.
[42] 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.
[43] R. Zhang, *FPGA Design and Implementation of the Prime Field Multipliers*. Xi'an, China: Xidian Universiy, 2013.
[44] K. Millar, "Design of a flexible schoenhage-strassen FFT polynomial multiplier with high-level synthesis," M.S. thesis, New York, NY, USA: Rochester Institute of Technology, 2019.
[45] R. Agrawal, L. Bu, A. Ehret, and M. Kinsy, "Open-source FPGA implementation of post-quantum cryptographic hardware primitives," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 211–217.
[46] C. Jayet-Griffon, M.-A. Cornelie, P. Maistri, P. Elbaz-Vincent, and R. Leveugle, "Polynomial multipliers for fully homomorphic encryption on FPGA," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2015, pp. 1–6.
[47] M. R. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. E. Lauter, and S. Lokam, "Homomorphic encryption standard," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 939, May 2019.
[48] T. Zhang, J. Wang, S. Guo, and Z. Chen, "A comprehensive FPGA reverse engineering tool-chain: From bitstream to RTL code," *IEEE Access*, vol. 7, pp. 38379–38389, 2019.
[49] M. E. S. Elrabaa, M. Al-Asli, and M. Abu-Amara, "Secure computing enclaves using FPGAs," *IEEE Trans. Depend. Sec. Comput.*, early access, Aug. 6, 2019, doi: 10.1109/TDSC.2019.2933214.
[50] J. Zhang and G. Qu, "Recent attacks and defenses on FPGA-based systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 3, pp. 1–24, Sep. 2019.

YANG SU received the B.S. degree of engineering in microelectronics and solid state electronics from the Information Engineering University of PLA, in 2012. He is currently pursuing the Ph.D. degree with the PLA Rocket Force University of Engineering. He is also a Lecturer with the Engineering University of People's Armed Police. His research interests include fully homomorphic encryption hardware accelerator design, reconfigurable crypto chip, and integrated circuit design.

BAILONG YANG received the B.S. and M.S. degrees in computer applications technology and the Ph.D. degree in aeronautics and astronautics manufacturing engineering from the PLA Rocket Force University of Engineering, Xi'an, China, in 1990, 1993, and 2001, respectively. He is currently a Professor with the PLA Rocket Force University of Engineering. His research interests include homomorphic encryption based on lattice, network security and post-quantum cryptography, complex networks, and computer simulation.



CHEN YANG (Member, IEEE) received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 2004, and the M.S. and Ph.D. degrees from the Institute of Microelectronics, Tsinghua University, in 2007and 2016, respectively. He was with VIA Technologies, Inc., Beijing, from 2007 to 2009. He is currently a Lecturer with the School of Microelectronics, Xi'an Jiaotong University. His current research interests include hardware security, homomorphic encryption acceleration, on-chip memory management technology, reconfigurable computing, and VLSI SoC design.

LUOGENG TIAN received the B.S. degree of military science of information communication from the Air Force Engineering University of PLA, in 2013. He is currently pursuing the Ph.D. degree with the PLA Rocket Force University of Engineering. He is also a Lecturer with the National University of Defense Technology. His research interests include information security and security of deep learning.