2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)

Hardware Architecture of a Number Theoretic Transform for a Bootstrappable RNS-based Homomorphic Encryption Scheme

Sunwoong Kim*, Keewoo Lee†, Wonhee Cho†, Yujin Nam‡§, Jung Hee Cheon†§, and Rob A. Rutenbar¶
*Division of Engineering and Mathematics, University of Washington, Bothell, WA 98011, USA
†Department of Mathematical Sciences, Seoul National University, Seoul 08826, South Korea
‡Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, South Korea
§CryptoLab, Seoul 08826, South Korea
¶Department of Computer Science and Department of Electrical and Computer Engineering,
University of Pittsburgh, Pittsburgh, PA 15260, USA
Email: sunwoong@uw.edu, {activecondor, wony0404, skyyujin96, jhcheon}@snu.ac.kr, rutenbar@pitt.edu

***Abstract*—Homomorphic encryption (HE) is one of the most promising solutions to secure cloud computing. The number theoretic transform (NTT) that is widely used for convolution operations in HE requires a large amount of computation and has high parallelism, and therefore it has been a good candidate for hardware acceleration. Nevertheless, prior NTT hardware solutions for HE-based applications are impractical in most applications because they do not seriously consider the critical bootstrapping procedure that allows unlimited homomorphic operations on encrypted data.**

In this paper, we suggest practical bootstrappable parameters, specifically for an established residue number system (RNS)-based HE scheme, and apply them to our NTT hardware design. In addition, to limit the size of internal memory for roots of unity increased by the bootstrappable parameters, only a few roots of unity are stored and others are generated on the fly. In our NTT hardware architecture, multiple NTT butterfly units (BUs) are efficiently deployed for high throughput and high resource utilization. In particular, several groups of BUs for respective moduli work in a parallel and pipelined manner, which is effective in an RNS-based HE scheme with a number of moduli.

Our implementation on a Xilinx UltraScale FPGA with the bootstrappable parameters achieves a 118× faster processing speed than a software implementation, and it further provides various trade-off choices such as the number of DSP slices against BRAMs based on available FPGA resources.

I. Introduction

Recently, machine learning has received much attention as an outstanding solution for a variety of applications such as speech recognition, image classification, and precision medicine. Although the demand for analysis of sensitive data such as financial data or medical data is increasing, traditional machine learning services require large datasets to be used during both training and inference to get meaningful results. Therefore, privacy becomes a major concern when providing cloud-based data analysis services. Homomorphic encryption (HE), an encryption scheme which allows computation between encrypted data, is an ideal solution for this privacy problem. By adopting HE, service providers do not need to decrypt the private data to perform computation on them [1].

Contrary to so-called somewhat HE (SHE) schemes where only a limited amount of computations are allowed, fully HE (FHE) schemes accompany *bootstrapping*, a method of initializing noise in encrypted data, which allows an unbounded amount of computation without decryption [2]. However, the bootstrapping procedure itself consists of a massive amount of homomorphic computation and accordingly requires large parameters such as a large polynomial degree N, which slows down the overall processing speed.

Residue number system (RNS)-based HE schemes are widely used these days [3]–[7]. They avoid operations on very large numbers by splitting a coefficient into several smaller ones, which enables HE operations to have high potential parallelism. In particular, RNS-HEAAN [3] is supported by many HE libraries including Microsoft's SEAL [8], IBM's HElib [9], and EPFL's Lattigo [10]. However, the RNS-based HE schemes still suffer from prohibitively slow processing speed on conventional software platforms, which renders them impractical for many critical HE-based applications.

Many studies have been conducted to accelerate HE schemes using FPGAs [7], [11]–[19]. Specifically, a number of works focus on the number theoretic transform (NTT) to speedup the polynomial ring multiplication which is the critical bottleneck of homomorphic operations. However, they do not seriously consider the critical bootstrapping procedure. For example, recent works that do not take into account the bootstrapping procedure have a shallow circuit depth under ten [7], [11]. Furthermore, to the best of the authors' knowledge, most of the previous works do not fully utilize parallelism in the RNS domain, which is feasible on recent large-scale FPGAs such as Xilinx UltraScale+ FPGAs on the Amazon cloud.

This paper presents a novel hardware architecture of NTT on an FPGA, especially focusing on the bootstrapping procedure and RNS domain. Its contributions are as follows:

- We suggest bootstrappable parameter sets for an RNS-based HE algorithm. The obtained special primes and

2576-2621/20/$31.00 ©2020 IEEE
DOI 10.1109/FCCM48280.2020.00017

IEEE computer society

their scaled inverse values are applied to modular operators in an NTT butterfly unit (BU).
- We deploy multiple BUs while utilizing input/output (I/O) and memory bandwidth efficiently. In addition, we propose a highly pipelined NTT architecture that uses various levels of parallelism between coefficients of a polynomial, NTT stages, and moduli.
- We reduce the number of roots of unity stored in block RAMs (BRAMs) from $O(N)$ to $O(\log N)$ and generate others on the fly. Specifically, roots of unity for all NTT stages are generated in parallel every cycle.

The remainder of this paper is organized as follows. The background is introduced in Section II. Section III suggests bootstrappable parameters for an RNS-based HE scheme. Sections IV and V present our root of unity generation method and novel NTT hardware architecture. Section VI evaluates our proposed design and Section VII concludes this paper.

II. BACKGROUND

A. Number Theoretic Transform

Instead of performing on a complex number field ($\mathbb{C}$), discrete fourier transform (DFT) can be generalized to other rings. We are interested in the case where the ring is over a finite field, or more specifically where the ring is integers modulo a prime p. We call this DFT on $\mathbb{Z}_p$ an NTT. Many algorithms for fast DFT on size N vectors with the time complexity $O(N \log N)$ (e.g., Cooley-Tukey [20]) can also be applied to NTT.

We call ω an Nth root of unity (or a twiddle factor) modulo prime p, if it satisfies $\omega^N \equiv 1 \pmod{p}$. A primitive N^{th} root of unity is an Nth root of unity that generates every N^{th} root of unity multiplicatively. By definition, the primitive N^{th} root of unity is required to perform DFT on N-sized vectors. It is known that there exists a primitive N^{th} root of unity modulo prime p if and only if $p \equiv 1 \pmod{N}$ holds.

In lattice-based cryptography including HE, we usually work over a ring $\mathbb{Z}_p[x]/\langle x^N + 1\rangle$ where N is a power of two and p is a prime. Multiplication in such a ring corresponds to *negative wrapped convolutions*, whereas an NTT-multiply-inverse NTT (INTT) paradigm performs multiplication over a ring $\mathbb{Z}_p[x]/\langle x^N - 1\rangle$, which corresponds to normal convolutions. With a slight modification to NTT and INTT algorithms, we also can perform multiplication over $\mathbb{Z}_p[x]/\langle x^N + 1\rangle$ efficiently [21]. To use this modification, the modulus p should satisfy $p \equiv 1 \pmod{2N}$, whereas the normal NTT/INTT requires $p \equiv 1 \pmod{N}$. In this paper, we follow this modified framework for efficiency and with slight abuse of notation we call these modified algorithms NTT and INTT.

The efficient INTT for negative convolution is described in Algorithm 1. For the sake of simplicity, the last scaling step is omitted. This algorithm takes the list of negative powers of a fixed primitive $(2N)^{\text{th}}$ root of unity ψ in bit-reversal order as an input, which is denoted as Ψ_{rev}^{-1}. More precisely, $\Psi_{rev}^{-1}[i]$ contains ψ^{-j}, where j is the bit-reverse of i.

Typically, an NTT/INTT is performed using BUs which are building blocks. The function ButterflyUnit($a[j]$, $a[j+t]$, W, p) in Algorithm 1 computes $(a[j]-a[j+t]) \cdot W \pmod{p}$ and $a[j]+a[j+t] \pmod{p}$ and stores the results in $a[j+t]$ and $a[j]$, respectively. When the number of input samples is N, the number of stages of NTT is log$N$ (line 2) and each stage consists of $\frac{N}{2}$ radix-2 BUs (lines 5 and 8), and thus the total number of BUs is $\frac{N}{2} \times \log N$.

Algorithm 1 INTT based on Gentleman-Sande butterfly [21]

Input: $\mathbf{a} = (a[0], a[1], \cdots, a[N-2], a[N-1])$, p, $\Psi_{rev}^{-1} = (\Psi_{rev}^{-1}[0], \Psi_{rev}^{-1}[1], \cdots, \Psi_{rev}^{-1}[N-1])$
Output: $\mathbf{a} \leftarrow$ INTT($\mathbf{a}$)

```
1:  t = 1
2:  for (m = N; m > 1; m = m/2) do
3:      j1 = 0
4:      h = m/2
5:      for (i = 0; i < h; i = i + 1) do
6:          j2 = j1 + t − 1
7:          W = Ψ^{-1}_{rev}[h + i]
8:          for (j = j1; j ≤ j2; j = j + 1) do
9:              ButterflyUnit(a[j], a[j + t], W, p)
10:         end for
11:         j1 = j1 + 2 × t
12:     end for
13:     t = 2 × t
14: end for
15: return a
```

B. RNS-HEAAN and Types of Moduli

HEAAN, also known as the CKKS scheme, is one of the most promising HE schemes [22]. This scheme is an invaluable solution to HE-based applications because it supports approximate arithmetic with fixed point numbers. More precisely, its message space is a ring of complex number vectors $\mathbb{C}^{N/2}$ with certain precision. The main idea of HEAAN is to consider the noise which is originally added to the plaintext for security as a part of error occurring during approximate computations. That is, unlike concurrent HE schemes, HEAAN does not separate the message with the noise. HEAAN works over a polynomial ring $\mathcal{R}_Q = \mathbb{Z}_Q[x]/\langle x^N + 1\rangle$, where Q is a *large modulus* ($Q = \Pi_{i=0}^{l} q_i$). To encrypt a message in the ring $\mathbb{C}^{N/2}$, we first encode the message into a plaintext in a set of moderate size elements (e.g., each coefficient being smaller than q_0). The plaintext m is then encrypted as a ciphertext ct = (a, b = as + m + e) which is in $\mathcal{R}_Q^2$, where s, $e \in \mathcal{R}_Q$ refer to a small secret key and a small noise, respectively.

Its full RNS-based variant called RNS-HEAAN splits a large coefficient of a polynomial into several smaller coefficients to avoid operations on very large numbers, which assures high potential parallelism [3]. However, it is not fully parallelized on existing CPU platforms but provides a good opportunity for custom hardware acceleration.

A homomorphic multiplier (HomMult) is one of the most-used homomorphic operators, but at the same time it is one of the biggest obstacles to the practical use of end-to-end HE-based applications because it is too time-consuming. This

TABLE I
PARAMETER SETTING OF PREVIOUS WORKS AND OUR SUGGESTED ONES

	λ	$dnum$	N	$l+1$	k	log$Q$	log$P$	logPQ	logq_0	logq_i	logp_i
RNS-HEAAN [3]	73	1	2^{15}	11	12	611	660	1271	62	55	55
RNS-HEAAN [4]	108	4	2^{16}	24	6	1,090	273	1363	62	45	-
RNS-HEAAN [4]	105	7	2^{16}	28	-	1,270	182	1452	62	45	-
HEAX [11] Set-A	128.1	-	2^{12}	2	-	-	-	109	-	-	-
HEAX [11] Set-B	128.5	-	2^{13}	4	-	-	-	218	-	-	-
HEAX [11] Set-C	128.1	-	2^{14}	8	-	-	-	438	-	-	-
Our Set-A	129.8	2	2^{17}	36	16	1,882	992	2874	62	52	62
Our Set-B	127.3	3	2^{17}	42	12	2,194	744	2938	62	52	62

inefficiency of HomMult comes from its complicatedness. For example, HomMult is composed of several large degree polynomial ring multiplications and also non-arithmetic operations such as changing moduli and rounding. More precisely, for controlling the noise to grow at a reasonable rate for each HomMult, HE literatures utilize so-called *Modup* technique in HomMult. In a nutshell, the Modup procedure first converts an element in $\mathcal{R}_Q$ into the $\mathcal{R}_{PQ}$ domain by changing the modulus, where P is a product of primes ($P = \Pi_{i=1}^{k} p_i$). After some operations in $\mathcal{R}_{PQ}$, Modup sends the result back to $\mathcal{R}_Q^2$ by scaling down $1/P$ and rounding. For a detailed description of HomMult, we refer the readers to [3], [22].

To achieve reasonable timing, it is inevitable to utilize NTT for large degree polynomial ring multiplications. However, the ring elements cannot stay in the NTT domain because the non-arithmetic operations do not commute with NTT/INTT. Therefore, for every non-arithmetic step, we need to perform INTT and NTT before and after the operations. The point here is that frequent NTT/INTT is the critical bottleneck of HomMult and HE schemes.

The *partial moduli* introduced above (q_i and p_i) are categorized into three types as follows:

- Base modulus (q_0): Whenever each HomMult is performed, the number of q_is decreases by 1, which means that the circuit depth decreases by 1, and this modulus is the last remaining one.
- Rescale modulus (q_i, where $1 \le i \le l$): The number of rescale moduli refers to the depth of a circuit. Typically, it is advantageous to make this number large to avoid bootstrapping as often as possible.
- Modup modulus (p_i, where $1 \le i \le k$): These are used to scale down the size of noises incurred during HomMult.

For area efficient modular multiplier (ModMult) design, Kim *et al.* represented these moduli and their scaled inverse values by a signed binary form with the minimum Hamming weight [14]. Although their approach is adopted in this paper, it is amended for our suggested bootstrappable parameters.

III. PARAMETERS FOR BOOTSTRAPPABLE RNS-HEAAN

HE schemes encrypt a message using noise. However, the magnitude of the noise increases as homomorphic operations are performed on the encrypted data. Specifically, it grows rapidly whenever HomMult is performed. If the magnitude exceeds a specific level, the correct message cannot be obtained after decryption. Here, the number of HomMults before reaching this threshold is called the *circuit depth*. The bootstrapping procedure that resets the noise and the circuit depth allows unlimited homomorphic operations on the encrypted data. However, its extremely slow speed makes HE-based solutions impractical. There are two approaches to speed up HE schemes with the bootstrapping: (i) accelerating the bootstrapping procedure itself; and (ii) increasing the interval between bootstrapping, i.e. the circuit depth. This paper focuses on the latter one. Generally, the bootstrapping procedure consumes the circuit depth of 15-20 itself. Therefore, for a practical design that requires a sufficient depth such as 20-25, the initial circuit depth should be set to around 40.

Table I compares parameters used in previous works and our suggested ones. In early studies, the security parameter λ of 80 was widely used [3]. However, for a variety of recent fields that deal with private data, λ needs to be increased to 128. As shown in the second through fourth rows, parameters of the original RNS-HEAAN scheme [3] and its variant [4] do not satisfy the 128 security. Meanwhile, as shown in the fifth through seventh rows, parameters of HEAX that satisfy the security of 128 do not consider bootstrapping, and therefore HomMults are allowed only less than 8 times [11]. The last two rows show our suggested parameters. The main difference between our two parameters is the number of evaluation keys, $dnum$. In the original RNS-HEAAN scheme [3], the size of log$P$ is set to be similar to the size of log$Q$. To increase the initial circuit depth by around 40, log$Q$ should be increased but there is an upper limit on the size of logPQ to ensure the security. To resolve this issue, Han *et al.* decompose ciphertexts by increasing $dnum$ [4]. As a result, log$Q$ is set to log$P \times dnum$. However, as $dnum$ increases, the size of memory to store evaluation keys increases, and therefore they cannot be stored in internal memory. Furthermore, NTT needs to be performed $dnum$ times more, which significantly increases latency. Therefore, as shown in the third column of Table I, we carefully chose 2 and 3 as $dnum$ which are small enough and achieve the circuit depth around 40.

In our parameter sets, the size of base modulus, logq_0, is set to 62 to preserve precision of a decrypted message, which is the same as in [3] and [4]. On the other hand, we set the size of rescale moduli, logq_i, to 52, which satisfies the following two criteria: First, it is large enough to do accurate approximate computation in RNS-HEAAN. Second, it is large enough to find sufficiently many lightweight prime numbers which are introduced in [14]. Using these primes, we can

TABLE II
q_i FOR A BASE MODULUS ($i = 0$) AND RESCALE MODULI ($1 \leq i \leq l$)

i	q_i	i	q_i
0	$2^{61} - 2^{26} + 1$	21	$2^{51} - 2^{24} + 2^{21} + 2^{18} + 1$
1	$2^{51} - 2^{29} - 2^{19} + 1$	22	$2^{51} - 2^{23} + 2^{20} + 1$
2	$2^{51} - 2^{28} - 2^{23} + 2^{19} + 1$	23	$2^{51} - 2^{23} + 2^{21} - 2^{18} + 1$
3	$2^{51} - 2^{28} - 2^{22} + 1$	24	$2^{51} + 2^{21} - 2^{18} + 1$
4	$2^{51} - 2^{28} + 2^{26} + 2^{20} + 1$	25	$2^{51} + 2^{22} - 2^{20} - 2^{18} + 1$
5	$2^{51} - 2^{27} - 2^{24} + 2^{22} + 1$	26	$2^{51} + 2^{22} + 2^{20} + 1$
6	$2^{51} - 2^{27} - 2^{19} + 1$	27	$2^{51} + 2^{23} + 2^{20} + 2^{18} + 1$
7	$2^{51} - 2^{27} + 2^{24} + 1$	28	$2^{51} + 2^{25} - 2^{23} - 2^{21} + 1$
8	$2^{51} - 2^{26} - 2^{22} - 2^{19} + 1$	29	$2^{51} + 2^{25} - 2^{22} + 2^{20} + 1$
9	$2^{51} - 2^{26} + 2^{23} + 2^{20} + 1$	30	$2^{51} + 2^{25} + 1$
10	$2^{51} - 2^{26} + 2^{24} - 2^{18} + 1$	31	$2^{51} + 2^{25} + 2^{19} + 1$
11	$2^{51} - 2^{25} - 2^{23} + 2^{19} + 1$	32	$2^{51} + 2^{25} + 2^{21} + 1$
12	$2^{51} - 2^{25} - 2^{22} - 2^{20} + 1$	33	$2^{51} + 2^{25} + 2^{22} + 2^{19} + 1$
13	$2^{51} - 2^{25} - 2^{22} + 2^{18} + 1$	34	$2^{51} + 2^{26} - 2^{20} - 2^{18} + 1$
14	$2^{51} - 2^{25} - 2^{21} - 2^{18} + 1$	35	$2^{51} + 2^{26} + 2^{21} + 2^{18} + 1$
15	$2^{51} - 2^{25} + 2^{20} - 2^{18} + 1$	36	$2^{51} + 2^{27} - 2^{24} - 2^{21} + 1$
16	$2^{51} - 2^{25} + 2^{23} - 2^{19} + 1$	37	$2^{51} + 2^{28} + 2^{18} + 1$
17	$2^{51} - 2^{25} + 2^{23} + 2^{20} + 1$	38	$2^{51} + 2^{28} + 2^{20} + 1$
18	$2^{51} - 2^{24} + 2^{19} + 1$	39	$2^{51} + 2^{29} + 1$
19	$2^{51} - 2^{30} - 2^{20} + 1$	40	$2^{51} - 2^{30} + 2^{19} + 1$
20	$2^{51} - 2^{29} - 2^{22} + 2^{18} + 1$	41	$2^{51} + 2^{29} + 2^{21} + 2^{18} + 1$

TABLE III
p_i FOR MODUP MODULI ($1 \leq i \leq k$)

i	p_i	i	p_i
1	$2^{61} - 2^{24} + 1$	9	$2^{61} - 2^{22} + 2^{19} + 1$
2	$2^{61} - 2^{21} + 1$	10	$2^{61} + 2^{22} + 2^{20} + 1$
3	$2^{61} + 2^{23} - 2^{18} + 1$	11	$2^{61} + 2^{23} + 2^{21} + 1$
4	$2^{61} + 2^{24} - 2^{19} + 1$	12	$2^{61} + 2^{25} + 2^{23} + 1$
5	$2^{61} + 2^{27} + 2^{21} + 1$	13	$2^{61} + 2^{28} - 2^{25} + 1$
6	$2^{61} + 2^{29} - 2^{22} + 1$	14	$2^{61} + 2^{29} + 2^{19} + 1$
7	$2^{61} + 2^{30} + 2^{18} + 1$	15	$2^{61} + 2^{30} + 2^{20} + 1$
8	$2^{61} + 2^{30} + 2^{26} + 1$	16	$2^{61} + 2^{30} + 2^{28} + 1$

speed up ModMult by replacing integer multiplication with bit-shift and addition. There is a relatively small limit when determining the size of modup moduli, $\log p_i$. However, the product of modup moduli must be larger than a certain value. In other words, as each modup modulus becomes smaller, the number of modup moduli increases. Since we already have 62-bit modular operators for the base modulus, the size of modup moduli is set to 62. All our suggested prime numbers for base/rescale moduli and modup moduli are shown in Table II and Table III, respectively. To conserve space, their scaled inverse values are omitted. Our moduli and scaled inverse values have Hamming weights of 5 or less, which allows an area-efficient hardware design.

Our suggested parameter sets, which require a larger N value, 2^{17}, and more moduli than the previous works, increase execution times in NTT and INTT. To speed up the NTT and INTT, a novel hardware architecture that fully exploits various levels of parallelism is proposed in Section V.

IV. ON-THE-FLY ROOT OF UNITY GENERATION

As we noted in Section II-A, the root of unity is required to perform NTT. In previous NTT hardware designs, all roots of unity are naively stored in internal memory [7], [11]–[13], [16], [17], [23], [24]. However, the required memory is linear in N and $(l+1) \cdot k$, and it leads storing all roots of unity in internal memory to be infeasible when N and/or $(l+1) \cdot k$ is too large. For example in INTT with our parameter sets, the naive method requires total 400Mb of internal memory (400Mb $\approx$ (62b $\times$ 17 + 52b $\times$ 41) $\times\ 2^{17}$).

Algorithm 2 INTT with on-the-fly roots of unity generation

Input: $\mathbf{a} = (a[0], a[1], \cdots, a[N-2], a[N-1])$, p,
$\Psi_{pow}^{-1} = (\Psi_{pow}^{-1}[0], \Psi_{pow}^{-1}[1], \cdots, \Psi_{pow}^{-1}[\log N - 1])$
Output: $\mathbf{a} \leftarrow$ INTT($\mathbf{a}$)

```
1:  t = 1
2:  l = 0
3:  for (m = N; m > 1; m = m/2) do
4:      h = m/2
5:      W = Ψ_pow^{-1}[l]
6:      for (k = 0; k < h; k = k + 1) do
7:          i = BitReverse(k, log h)
8:          j1 = i × 2 × t
9:          j2 = j1 + t
10:         for (j = j1; j < j2; j = j + 1) do
11:             ButterflyUnit(a[j], a[j + t], W, p)
12:         end for
13:         W = W × Ψ_pow^{-1}[l + 1]
14:     end for
15:     t = 2 × t
16:     l = l + 1
17: end for
18: return a
```

In this aspect, we designed an NTT/INTT algorithm without storing all roots of unity: it rather stores only a few roots of unity and computes other roots of unity on the fly from the stored ones. This algorithm captures the trade-off between computation and storage. However, our modification does not increase the computation asymptotically, i.e. the computational cost is still $O(N \log N)$. In contrast, it decreases the storage from $O(N)$ bits to $O(\log N)$ bits.

Our method for INTT is described in Algorithm 2. Again, the scaling step is omitted. The algorithm takes the list of $(-2^i)^{\text{th}}$ powers of a fixed primitive $(2N)^{\text{th}}$ root of unity ψ as an input, which is denoted as Ψ_{pow}^{-1}. More precisely, $\Psi_{pow}^{-1}[i]$ contains ψ^{-2^i}. The notation BitReverse$(k, \log h)$ is for the bit reverse of k as a $\log h$-bit integer. The main differences from Algorithm 1 are as follows: (i) it takes Ψ_{pow}^{-1} instead of Ψ_{rev}^{-1}, decreasing the input size; (ii) instead of taking the roots of unity in bit-reversed order, the bit-reversing procedure is done in line 7; and (iii) instead of precomputing all roots of unity, roots of unity are generated and updated in lines 2, 5, 13, and 16.

Note that there are several prior works that generate roots of unity on-the-fly [18], [19], [25]. However, our hardware design differs from them in that multiple roots of unity for all INTT stages with respective moduli are generated in parallel. In addition, roots of unity are generated every cycle by using a fully pipelined ModMult design. Lastly, a special ordering method is used. The details are explained in Section V-D.

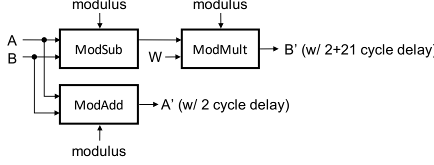

Fig. 1. Architecture of a radix-2 BU for INTT.

V. NTT Hardware Architecture

In this section, our proposed hardware architecture of NTT is presented. NTT and INTT have almost the same architecture, except that their directions are reversed and a last scaling step is added to INTT. Therefore, we reuse the same circuits for both NTT and INTT, and the architecture of INTT is only presented in this paper.

A. Butterfly Unit

First, we describe a BU that is a fundamental unit of INTT. Fig. 1 shows a radix-2 BU architecture for INTT. In this figure, A and B represent input samples, while A' and B' represent output samples. In addition, W refers to a root of unity. The BU architecture is composed of a modular subtractor (ModSub), a modular adder (ModAdd), and a ModMult. The ModSub and ModAdd architectures are almost the same as in [12] and require a two cycle delay. For the ModMult architecture, we adopt the fully pipelined Barrett ModMult architecture with lightweight moduli in [14]. Since the maximum Hamming weight of our moduli and scaled inverse values is one greater than that in [14], our ModMult design requires a one cycle longer delay that is 21 cycles. Since the BU is a fully pipelined design, two input samples are continuously put into the BU and two output samples are generated per cycle after a delay. Since ModMult is only applied to generation of B', B' has a 21 cycle longer delay than A'. When multiple BUs are connected in series, the output samples re-enter to the next BU as input samples.

B. Group of Butterfly Units

To improve the speed of INTT on an FPGA, multiple BUs should be exploited at the same time. However, it is difficult to deploy all $\frac{N}{2} \times \log N$ BUs on an FPGA when N is very large because each BU includes expensive modular operators [15]. Therefore, we need to use a few BUs selectively. There are mainly two methods in deployment of BUs: (i) BUs are deployed in parallel for the same stage; and (ii) a single (or a few) BU is deployed for each stage and the multiple BUs are connected in serial. The first method is intuitive and its intermediate data ordering is simple. However, it requires a high I/O or memory bandwidth in a short time as the number of BUs increases. Therefore, we adopt the second method.

However, the serial deployment method incurs a long delay when integrated with Algorithm 2 that changes the order of input samples. Fig. 2 shows an example of the processing order when N is 32. In this figure, each row corresponds to each operation of BU. The first and second columns of each stage represent indices of input samples. The exponents of roots of unity shown in the third column of each stage increase by a fixed amount called the *update constant*, and the update constant is doubled as the stage index is incremented by 1. As shown in the case between Stage 1 and Stage 2, the input sample with the index 2 of Stage 2 is put after the input sample with the same index of Stage 1 is processed, which is represented by an arrow. The next three stages have the same dependencies, and therefore the delay becomes accumulated.

Stage 1			Stage 2			Stage 3			Stage 4			Stage 5		
Input		Exp.ω	Input		Exp.ω	Input		Exp.ω	Input		Exp.ω	Input		Exp.ω
0	1	1	0	2	2	0	4	4	0	8	8	0	16	16
16	17	3	1	3		1	5		1	9		1	17	
8	9	5	16	18	6	2	6		2	10		2	18	
24	25	7	17	19		3	7		3	11		3	19	
4	5	9	8	10	10	16	20	12	4	12		4	20	
20	21	11	9	11		17	21		5	13		5	21	
12	13	13	24	26	14	18	22		6	14		6	22	
28	29	15	25	27		19	23		7	15		7	23	
2	3	17	4	6	18	8	12	20	16	24	24	8	24	
18	19	19	5	7		9	13		17	25		9	25	
10	11	21	20	22	22	10	14		18	26		10	26	
26	27	23	21	23		11	15		19	27		11	27	
6	7	25	12	14	26	24	28	28	20	28		12	28	
22	23	27	13	15		25	29		21	29		13	29	
14	15	29	28	30	30	26	30		22	30		14	30	
30	31	31	29	31		27	31		23	31		15	31	

Fig. 2. Processing order of INTT by Algorithm 2 when N is 32. The arrows represent dependencies between gray colored samples.

	C1				C2				C3				C4			
	Cycle	Input		Exp.ω	Cycle	Input		Exp.ω	Cycle	Input		Exp.ω	Cycle	Input		Exp.ω
Stage 1	0	0	1	1	0	4	5	9	0	2	3	17	0	6	7	25
	1	16	17	3	1	20	21	11	1	18	19	19	1	22	23	27
	2	8	9	5	2	12	13	13	2	10	11	21	2	14	15	29
	3	24	25	7	3	28	29	15	3	26	27	23	3	30	31	31

	C5				C6				C7				C8			
Stage 2	2	0	2	2	2	4	6	18	…							
	3	16	18	6	3	20	22	22	23	1	3	2	23	5	7	18
	4	8	10	10	4	12	14	26	24	17	19	6	24	21	23	22
	5	24	26	14	5	28	30	30	25	9	11	10	25	13	15	26
									26	25	27	14	26	29	31	30

	C9				C10				C11				C12			
Stage 3	4	0	4	4	…											
	5	16	20	12	25	1	5	4	25	2	6	4	…			
	6	8	12	20	26	17	21	12	26	18	22	12	46	3	7	4
	7	24	28	28	27	9	13	20	27	10	14	20	47	19	23	12
					28	25	29	28	28	26	30	28	48	11	15	20
													49	27	31	28

GBU1

Refreshing order (by RB1)

	C1				C2				C3				C4			
Stage 4	p.t.+0	0	8	8	p.t.+0	4	12	8	p.t.+0	16	24	24	p.t.+0	20	28	24
	p.t.+1	1	9	8	p.t.+1	5	13	8	p.t.+1	17	25	24	p.t.+1	21	29	24
	p.t.+2	2	10	8	p.t.+2	6	14	8	p.t.+2	18	26	24	p.t.+2	22	30	24
	p.t.+3	3	11	8	p.t.+3	7	15	8	p.t.+3	19	27	24	p.t.+3	23	31	24

…

GBU2

Fig. 3. Modified order of Fig. 2 when four BUs per stage ($c = 4$) are used in parallel. In this figure, *p.t.* refers to a pipetime.

To solve this problem, additional BUs are deployed for each stage. We determine the number of BUs per stage, denoted by c, based on the total number of available digital signal processing (DSP) slices on a target FPGA because the number of DSP slices is usually more limited than those of look-up tables (LUTs) and flip flops (FFs). We then divide an input sample sequence into c parts, and the partial sequences are put into the BUs. Fig. 3 shows the modified order of Fig. 2, including cycles, when c is four. In this figure, Ci represents i-th BU core. The input samples with indices 0, 2, 4, and 6 of Stage 1 are only processed by ModAdd in C1, C3, C2, and C4, respectively, and thus C5 and C6 of Stage 2 start after the 2 cycle delay. On the other hand, ModSub and ModMult

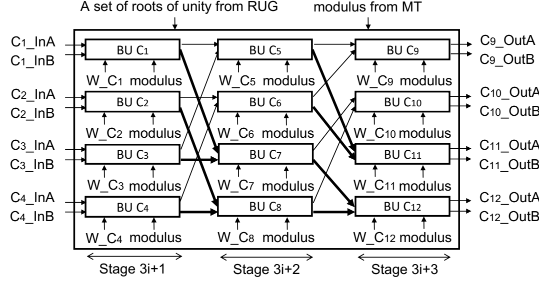

Fig. 4. Hardware architecture of GBU when four BUs are deployed per stage.

are applied to the input samples with indices 1, 3, 5, and 7 of Stage 1, so C7 and C8 of Stage 2 start after the 23 cycle delay. BU cores in Stage 3 work in the same manner. Since we aim at a large N value, 2^{17}, the accumulated delay in Stages 1-3 (the critical delay is 23×3 cycles) is negligible, and therefore the throughput is almost 8 samples/cycle. Meanwhile, BU cores of Stage 4 receive input samples with the index difference of 8, but the input samples are ready after at least the $N/(2\times 2\times 4)$ cycle delay (when $N = 2^{17}$, the delay is 2^{13} cycles). Therefore, a reordering buffer (RB) to refresh the order is needed. BU cores between two RBs comprise a group of BUs (GBU). The number of stages in a single GBU and the number of GBUs in the whole INTT design are calculated by $1+\log c$ and $\lceil \log N/(1+\log c)\rceil$, respectively.

Fig. 4 shows the fully pipelined hardware architecture of a single GBU when c is four. In this figure, intermediate and output samples generated by ModMults in BUs are represented by bold lines. The GBU receives eight input samples and 12 roots of unity generated from a root of unity generator (RUG) every cycle. After a delay, eight output samples are generated and transferred to the next RB every cycle.

To further improve the throughput, a different level of parallelism is used. To be specific, GBUs using different moduli work in parallel and pipelined manner. In HomMult of RNS-HEAAN, base and rescale moduli are only used for INTT [3], [4]. For example, when our parameter Set-B is used, 42 moduli are used for INTT. If the c value is set to four, there are 6 (= $\lceil \log 2^{17}/(1+\log 4)\rceil$) GBUs and the pipetime for each GBU is about 16K cycles (= $2^{17}/8$). In this case, 16K×(5+42) cycles are needed to complete the INTT on a polynomial.

C. Reordering Buffer

The i-th RB stores output samples generated by the i-th GBU, and transfers these samples to the (i+1)-th GBU after reordering. Fig. 5 shows the architecture of the first RB (RB1) implemented in BRAMs, including the write/read order, when N and c are 2^{17} and 4, respectively. For reordering, eight samples, which are generated by the first GBU every cycle, are stored in respective buffers in RB1. Four BU cores in Stage 4 reads eight samples with the index difference of 8 from RB1. For example, samples with indices 0, 8, ..., 48, and 56 are read at the first cycle. If these samples, which are generated by the same BU core, are stored in the same BRAM buffer, the bandwidth of the BRAM needs to be large, which reduces the

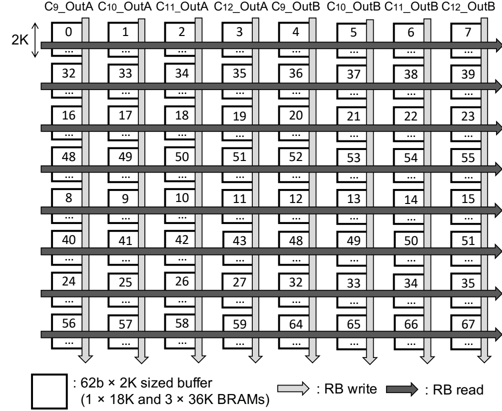

Fig. 5. The architecture of the first RB implemented in BRAMs, including the write/read orders, when N and c are 2^{17} and 4, respectively.

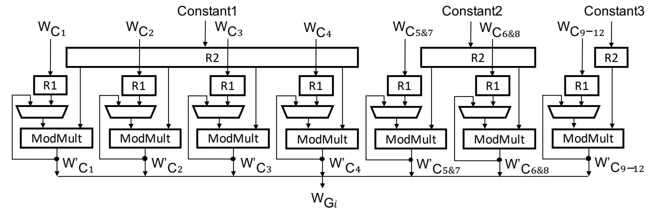

Fig. 6. Hardware architecture of RUG for a single GBU when N and c are 2^{17} and 4, respectively.

utilization of BRAMs. Therefore, an output sample sequence from each BU core is written in eight separate BRAM buffers. Although not shown in Fig. 5 for the sake of simplicity, a double buffering technique is used for simultaneous reading and writing, and therefore 128 (= $2\times 8\times 8$) 62-bit × 2K-sized BRAM buffers are included in each RB. When transferred to the four BU cores in Stage 4, eight samples are read horizontally as shown in Fig. 5. The next RBs have the same architecture except that they read 8^{i-1} samples vertically from the same buffer and then horizontally move to the next buffer.

D. Roots of Unity Generator

RUG generates all roots of unity from the *base roots of unity* of which number is $O(\log N)$ and provides them for GBUs on the fly. Fig. 6 shows the block diagram of a single RUG architecture when N and c are 2^{17} and 4, respectively. Each GBU needs 12 roots of unity, but C5 and C7, C6 and C8, and C9 through C12 use the same roots of unity, respectively, as shown in Fig. 3. Therefore, each RUG only generates seven roots of unity, denoted by W_{C1}, W_{C2}, W_{C3}, W_{C4}, $W_{C5\&7}$, $W_{C6\&8}$, and $W_{C9\text{-}12}$, every cycle. These seven roots of unity comprise a group of roots of unity, W_{Gi}, and are transferred to the i-th GBU. Simultaneously, they are put into ModMults in RUG to generate the next roots of unity.

Fig. 7 shows the ROM arrangement to store base roots of unity for both NTT and INTT when our parameter Set-B is used. In this figure, different colors refer to base roots of unity for different moduli. As mentioned above, each RUG

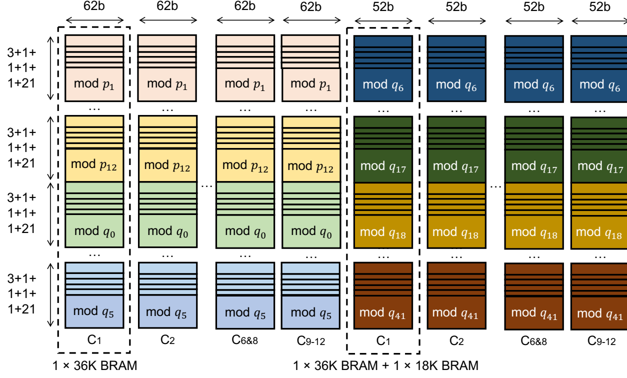

Fig. 7. ROM arrangement to store base roots of unity for both NTT and INTT when our parameter Set-B in Table I is used.

needs seven base roots of unity for each modulus. However, the 21 cycle delay incurred by ModMult changes the hardware architecture of RUG: (i) during the delay, roots of unity stored in ROMs are used as input operands of ModMults, which increases the number of base roots of unity to be stored; (ii) after the delay, roots of unity generated by ModMults are used as input operands. Since roots of unity for GBU1 vary every cycle, 21 base roots of unity are needed. On the other hand, those for GBU2 vary every eight cycles, and therefore three base roots of unity ($= \lceil 21/8 \rceil$) are stored. Lastly, those for GBU3, GBU4, GBU5, and GBU6 vary at least every 64 cycles, so only a single base root of unity is required. The 21 base roots of unity for GBU1 are directly transferred to ModMults. On the other hand, to minimize the BRAM bandwidth, the base roots of unity for other GBUs are prefetched into registers denoted by R1 in Fig. 6 and exploited during the next pipetime with the next modulus. Likewise, the update constants are moved from ROMs to registers denoted by R2 and reused.

Since the GBU receives seven base roots of unity at the same time, base roots of unity are stored in seven separate ROMs. Basically, base roots of unity for base and modup moduli are stored in 62-bit ROMs while those for the rescale moduli are stored in 52-bit ROMs. However, several base roots of unity for rescale moduli are moved to 62-bit ROMs to increase the utilization of BRAMs. For example, under our parameter Set-B, base roots of unity for p_1 through p_{12} and q_0 through q_5 are stored in 62-bit ROMs while those for q_6 through q_{41} are stored in 52-bit ROMs as shown in Fig. 7.

E. Modulus Table

Our parameter sets have more than 50 moduli and their scaled inverse values. These values are stored in the modulus table (MT), and a pair is chosen for the first GBU and RUG using a selecting signal. This pair is delayed by registers and provided to next GBUs and RUGs in a pipelined manner.

F. Overall Architecture

Fig. 8 shows the overall architecture of our proposed INTT design for our bootstrappable parameter sets. We set c to four, but this value is configurable. Specifically, higher c assures

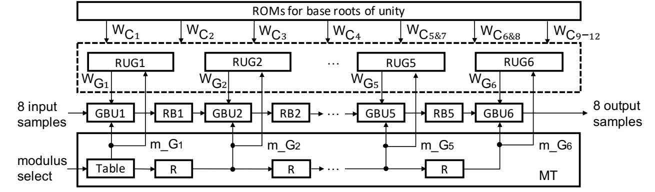

Fig. 8. Overall INTT architecture for our bootstrappable parameter sets.

a higher throughput, a shorter delay, and a fewer BRAMs for RBs but requires more DSP slices. Our INTT architecture includes six GBUs, five RBs, six RUGs, and a single MT. Since only the circuits for the last two stages are used in GBU6, the remaining circuits are used for a scaling step.

VI. Evaluation

A. Hardware Implementation Results

We implemented the INTT design using Verilog HDL, and the Xilinx Vivado Design Suite (2019.1) was used for synthesis, place-and-route, and bitstream generation stages. The generated bitstream was programmed on the Xilinx UltraScale FPGA (xcvu190-2flgc2104e) including 1,800 DSP slices, 132.9Mbit BRAMs, 1M LUTs, and 2M FFs.

The verification was conducted as follows: A test input polynomial was generated by an external host PC. It was transferred to the FPGA via an I/O interface module and an input buffer. Specifically, the I/O interface module received the input data and stored them in the input buffer. Eight samples gathered in this buffer were then sent to our INTT module every cycle. After INTT, the output data were sent back to the host PC via the output buffer and I/O interface module.

Table IV compares our proposed INTT design with three previous designs. The second row of this table shows target Xilinx FPGA devices, and the third through fifth rows present N, $l+1$, and the maximum bit-width of moduli, respectively. The sixth through eleventh rows show the maximum frequency, the numbers of LUTs, FFs, and DSP slices, the size of BRAM, and the throughput normalized to [16], respectively. Note that the throughput is affected not by N but by the number of BU cores working in parallel. To compare throughputs of the previous and proposed designs when a similar number of hardware resources are used, the throughputs are divided by the numbers of DSP slices and LUTs, which are shown in the last two rows. Originally, the previous works are designed for larger functions such as polynomial multiplication but chosen for this evaluation because they reuse the same circuits for INTT and other functions. Note that additional hardware resources deployed for non-INTT operations in Roy *et al.* [7] are excluded from this table. The throughputs of the previous works are calculated using the number of cycles or execution time of INTT provided in the papers.

Chen *et al.* [16] only deploy two BUs on an FPGA, and therefore the design uses the lowest resources and shows the second lowest throughput among the four designs. Even though it shows the highest throughput/(DSP or LUT) value, a single specific modulus (e.g., $2^{57}+25\cdot2^{13}+1$) is only available

TABLE IV
COMPARISON OF INTT HARDWARE DESIGNS

Design	Chen [16]	Roy [7]	Ozturk [12]	**Proposed**
Device	xc6slx100	xczu9eg	xc7vx690t	**xcvu190**
No. of samples	2^{11}	2^{12}	2^{15}	$\mathbf{2^{17}}$
No. of moduli	1	6	41	**~42**
Max. bit-width	58	30	32	**62**
$f_{\max}$ (MHz)	210	200	250	**200**
kLUT	6	55	219	**365**
kFF	19	22	91	**335**
DSP	64	182	768	**1,332**
BRAM (KB)	113	1,746	869	**10,163**
Norm. throughput	1.00	0.33	4.65	**19.97**
Norm. throu./DSP	1.00	0.11	0.39	**0.96**
Norm. throu./LUT	1.00	0.04	0.13	**0.34**

TABLE V
FPGA RESOURCE BREAKDOWN

Module	kLUT	kFF	BRAM (KB)	DSP
GBUs	183 (50%)	203 (60%)	0	912 (68%)
RBs	61 (17%)	45 (13%)	10,080 (99%)	0
RUGs	80 (22%)	86 (26%)	83 (1%)	420 (32%)
MT	41 (11%)	2 (1%)	0	0

for modular reduction. As a result, it cannot be used in RNS-based HE schemes. Roy *et al.* [7] show the lowest throughput in this table. However, the authors claim the throughput can be improved by deploying more core processors (e.g., ten core processors on the UltraScale+ FPGA). In addition, this design supports modular reduction with several moduli. Nevertheless, the low throughput/(DSP or LUT) and a small number of moduli make the design impractical for real applications. Ozturk *et al.* [12] exploit more resources on a larger FPGA for high-throughput than the prior two designs. In addition, this design supports arbitrary moduli. Therefore, it is the most suitable design to compare with ours. Our design shows the highest throughput that is 4× higher than that of [12]. Furthermore, the throughput/(DSP or LUT) is 2-3× larger than that of [12]. These results come from the fact that our design fully utilizes various levels of parallelism in the RNS domain.

Table V shows the FPGA resource breakdown of our design. Except for BRAMs, six GBUs occupy most of the resources. To be specific, they use 50% of LUTs and 68% of DSP slices. For BRAMs, five RBs use 10MB, which is the majority in the whole design. This size can be reduced by increasing the number of BUs that use DSP slices, which provides trade-off choices depending on available resources.

Table VI shows the improvement in internal memory size by our on-the-fly root of unity generation method. The first column presents our two parameters, and the second and third columns show the memory sizes for roots of unity of the conventional and our proposed methods, respectively. Note that 78.75KB BRAMs are allocated in our actual implementation (see Fig. 7). As shown in the last column, the memory sizes are reduced by more than 99% because the number of roots of unity to be stored decreases from $O(N)$ to $O(\log N)$.

B. Execution Time

In this subsection, our FPGA implementation is compared with the software implementation of INTT in RNS-HEAAN.

TABLE VI
IMPROVEMENT IN INTERNAL MEMORY SIZE FOR ROOTS OF UNITY

Parameter	w/o our method	**w/ our method**	Improvement
Our Set-A	44.91MB	**64.76KB**	99.86%
Our Set-B	45.91MB	**70.29KB**	99.85%

TABLE VII
EXECUTION TIME OF INTT IMPLEMENTATIONS

Parameter	Software implementation [4]	FPGA implementation Chen [16]	FPGA implementation **Proposed**
Our Set-A	387ms	93ms	**3.28ms**
Our Set-B	446ms	109ms	**3.76ms**

Note that the reference software code provided by the authors of [4] was the best code available for comparison, but it still has room for optimization. The reference software code was executed on the Intel i9-9820X CPU, running at the frequency of 3.3GHz and possessing the 16.5MB cache.

A test input was generated by the software code. Specifically, randomly selected complex numbers were encoded into a single plaintext vector. By encrypting this vector, a ciphertext polynomial was generated. This polynomial was forward-transformed and then transferred to the software implementation and our FPGA implementation. The execution time of the software implementation consumed only in the INTT function was measured using the Chrono C++ library. On the other hand, the execution time of our FPGA implementation was calculated by dividing the total number of cycles to complete INTT on the test input by the maximum frequency.

Table VII shows the execution times of the software implementation and the previous and proposed FPGA implementations. Although the previous hardware designs in Table IV cannot support our bootstrappable parameters, the execution times of [16], which are estimated using the number of cycles presented in the paper, are included in Table VII. The second and third rows show the results when our parameter Set-A and Set-B are used, respectively. On average, the execution time of our FPGA implementation is 118 and 28 times faster than the execution times of the software implementation and the previous FPGA implementation, respectively.

VII. CONCLUSION

In this paper, new parameter sets for the practical bootstrappable RNS-HEAAN scheme are suggested. These parameter sets are highly correlated with the final architecture and performance. For example, we deployed many BUs running in parallel and proposed a highly pipelined NTT architecture exploiting various levels of parallelism in the RNS domain to alleviate the increase in execution time caused by the large parameters. In addition, we proposed the RUG algorithm and its hardware architecture to limit the increase in internal memory size by the bootstrappable parameter sets. Furthermore, the suggested moduli with low Hamming weights of 5 or less allow the implementation of the ModMult in BU using shifters and adders only. For future work, we plan to extend our NTT design to a larger FPGA and apply it to HomMult of the bootstrappable RNS-HEAAN scheme.

References

[1] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsingm, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *Proc. ICML*, 2016.

[2] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *Proc. STOC*, 2009.

[3] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A Full RNS Variant of Approximate Homomorphic Encryption," in *Proc. SAC*, 2018.

[4] K. Han and D. Ki, "Better Bootstrapping for Approximate Homomorphic Encryption," in *Proc. CT-RSA*, 2020.

[5] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca, "A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes," in *Proc. SAC*, 2016.

[6] Y. Son and J. H. Cheon, "Revisiting the Hybrid Attack on Sparse Secret LWE and Application to HE Parameters," in *Proc. WAHC*, 2019.

[7] S. S. Roy, F. Turan, K. Jarvinen, F. Vercauteren, and I. Verbauwhede, "FPGA-based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data," in *Proc. HPCA*, 2019.

[8] H. Chen, K. Laine, and R. Player, "Simple Encrypted Arithmetic Library - SEAL v2.1," in *Proc. FC*, 2017.

[9] S. Halevi and V. Shoup, "Algorithms in HElib," in *Proc. CRYPTO*, 2014.

[10] Lattigo 1.3.0, [Online]. Available: http://github.com/ldsec/lattigo, 2019, EPFL-LDS.

[11] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: High-Performance Architecture for Computation on Homomorphically Encrypted Data in the Cloud," in *Proc. ASPLOS*, 2020.

[12] E. Ozturk, Y. Doroz, E. Savas, and B. Sunar, "A Custom Accelerator for Homomorphic Encryption Applications," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 3-16, Jan. 2017.

[13] D. B. Cousins, K. Rohloff, and D. Sumorok, "Accelerating Secure Computing with a Dedicated FPGA-based Homomorphic Encryption Co-Processor," *IEEE Trans. Emerging Topics Comput.*, vol. 5, no. 2, pp. 193-206, Apr.-Jun., 2017.

[14] S. Kim, K. Lee, W. Cho, J. H. Cheon, and R. A. Rutenbar, "FPGA-based Accelerators of Fully Pipelined Modular Multipliers for Homomorphic Encryption," in *Proc. ReConFig*, 2019.

[15] X. Cao, C. Moore, M. O'Neill, E. O'Sullivan, and N. Hanley, "Optimised Multiplication Architectures for Accelerating Fully Homomorphic Encryption," *IEEE Trans. Comput.*, vol 65. no. 9, pp.2794-2806, Sep. 2016.

[16] 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: Regular Papers*, vol. 62, no. 1, pp. 157-166, Jan. 2015.

[17] T. Poppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating Homomorphic Evaluation on Reconfigurable Hardware," in *Proc. CHES*, 2015.

[18] S. S. Roy, F. Vercauteren, J. Vliegen, and I. Verbauwhede, "Hardware Assisted Fully Homomorphic Function Evaluation and Encrypted Search," *IEEE Trans. Comput.*, vol 66. no. 9, pp.1562-1572, Sep. 2017.

[19] S. S. Roy, K. Jarvinen, 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.

[20] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. comput.*, vol. 19, no. 90, pp. 297–301, Apr. 1965.

[21] P. Longa and M. Naehrig, "Speeding Up the Number Theoretic Transform for Faster Ideal Lattice-based Cryptography," in *Proc. CANS*, 2016.

[22] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Proc. ASIACRYPT*, Nov. 2017.

[23] J.-H. Ye and M.-D. Shieh, "Low-Complexity VLSI Design of Large Integer Multipliers for Fully Homomorphic Encryption," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 26, no. 9, pp. 1727-1736, Sep. 2018.

[24] 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. Syst.*, vol. 22, no. 9, pp. 1879-1887, Sep. 2014.

[25] Y. Doroz, E. Ozturk, and B. Sunar, "Accelerating Fully Homomorphic Encryption in Hardware," *IEEE Trans. Comput.*, vol 64. no. 6, pp.1509-1521, Jun. 2015.