RISE: RISC-V SoC for En/Decryption Acceleration on the Edge for Homomorphic Encryption

Zahra Azad, Guowei Yang, *Graduate Student Member, IEEE*, Rashmi Agrawal, *Student Member, IEEE*, Daniel Petrisko, *Member, IEEE*, Michael Taylor, *Senior Member, IEEE*, and Ajay Joshi, *Senior Member, IEEE*

***Abstract*— Today, edge devices commonly connect to the cloud to use its storage and computing capabilities. This leads to security and privacy concerns about user data. Homomorphic encryption (HE) is a promising solution to address the data privacy problem as it allows arbitrarily complex computations on encrypted data without ever needing to decrypt it. While there has been a lot of work on accelerating HE computations in the cloud, small attention has been paid to the message-to-ciphertext and ciphertext-to-message conversion operations on the edge. In this work, we profile the edge-side conversion operations, and our analysis shows that during conversion error sampling, encryption and decryption operations are the bottlenecks. To overcome these bottlenecks, we present RISE, an area and energy-efficient RISC-V system-on-chip (SoC). RISE leverages an efficient and lightweight pseudorandom number generator (PRNG) core and combines it with fast sampling techniques to accelerate the error sampling operations. To accelerate the encryption and decryption operations, RISE uses scalable data-level parallelism to implement the number theoretic transform (NTT) operation, the main bottleneck within the encryption and decryption operations. In addition, RISE saves area by implementing a unified en/decryption datapath, and efficiently exploits techniques like memory reuse and data reordering to utilize a minimal amount of ON-chip memory. We evaluate RISE using a complete RTL design containing a RISC-V processor interfaced with our accelerator. Our analysis reveals that for message-to-ciphertext and ciphertext-to-message conversions, using RISE leads up to 5986.99× and 1164.1× more energy-efficient solution, respectively, than when using just the RISC-V processor.**

***Index Terms*— Cheon–Kim–Kim–Song (CKKS) scheme, edge-side operations, hardware acceleration, homomorphic encryption (HE), privacy-preserving computing, RISC-V.**

I. Introduction

CLOUD computing has enabled reliable and affordable access to shared computing resources at scale. Hence, energy- and area-constrained edge devices outsource their computing needs to a third-party cloud system. However, outsourcing data to a third-party cloud raises data security and privacy concerns. While an edge device can encrypt the data and send it to the cloud, within the cloud the data needs to be decrypted before processing. The cloud needs to decrypt the data, which again leaves the data vulnerable to all kinds of data breaches.

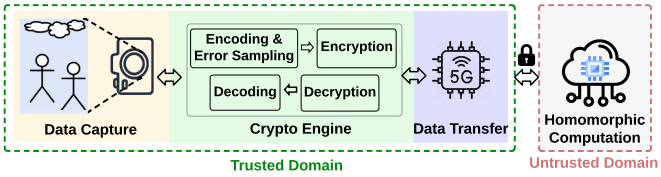

Fig. 1. Dataflow of an end-to-end encrypted computation based on HE.

Homomorphic encryption (HE) [1], [2] has emerged as a class of encryption schemes that address this problem by enabling computation on encrypted data. Fig. 1 shows an illustrative use case of how HE can be used to outsource secure computation. A user captures an image/video using an edge device. The captured image/video is encoded and encrypted on the edge device and then transferred to a third-party cloud system. The untrusted cloud system can process the encrypted data and send the encrypted result back that can be decrypted and decoded only by the user.

Although HE-based privacy-preserving computing seems plausible, it is several orders of magnitude slower than operating on unencrypted data [3]. To bridge this performance gap, several existing works take advantage of software and hardware optimizations to accelerate cloud-side HE operations running on CPU [4], [5], GPU [6], [7], and custom hardware accelerators [8], [9], [10], [11], [12]. Unfortunately, small attention has been paid to edge-side operations even when the edge-side operations are nontrivial. For encrypting the data, the edge device needs to perform encoding, error sampling, and encryption. These three operations together form the "message-to-ciphertext" conversion operation. Similarly for decrypting the data received from the cloud, the edge device needs to perform decryption and decoding. These two operations together form the "ciphertext-to-message" conversion operation. These edge-side operations incur huge memory consumption (on the order of several MBs) and computation overhead.

Manuscript received 13 February 2023; revised 15 May 2023; accepted 12 June 2023. Date of publication 17 July 2023; date of current version 27 September 2023. This work was supported in part by the Air Force Research Laboratory (AFRL) and in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement FA8650-18-2-7856. *(Corresponding author: Zahra Azad.)*

Zahra Azad, Guowei Yang, Rashmi Agrawal, and Ajay Joshi are with the Department of Electrical and Computer Engineering (ECE), Boston University, Boston, MA 02215 USA (e-mail: zazad@bu.edu; guoweiy@bu.edu; rashmi23@bu.edu; joshi@bu.edu).

Daniel Petrisko and Michael Taylor are with the Paul G. Allen School of Computer Science and Engineering (CSE), University of Washington, Seattle, WA 98195 USA (e-mail: petrisko@cs.washington.edu; profmbt@cs.washington.edu).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TVLSI.2023.3288754.

Digital Object Identifier 10.1109/TVLSI.2023.3288754

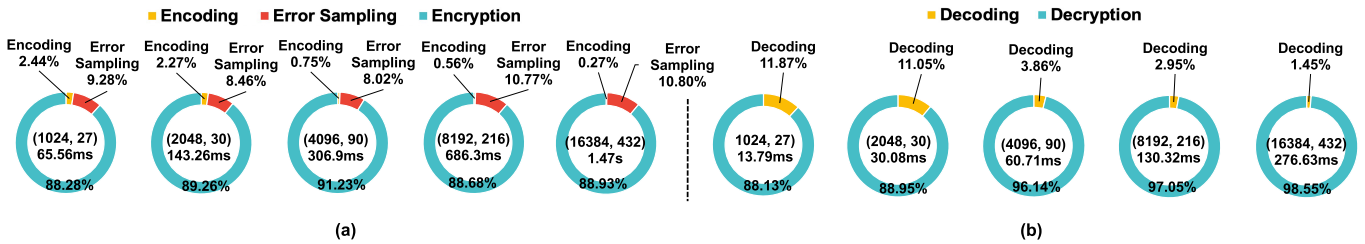

Fig. 2. Latency breakdown of (a) message-to-ciphertext and (b) ciphertext-to-message conversion operations running on BlackParrot using SEAL-Embedded library. Corresponding scheme parameters (N, $\log Q$) and latencies are specified inside the doughnut charts.

To accelerate these edge-side operations, recently Microsoft released SEAL-Embedded [3] as the first HE library targeting embedded devices. SEAL-Embedded proposes a number of optimizations for error sampling, en/decoding, and en/decryption on resource-constrained edge devices. To enable computing on a variety of data captured by the sensors on the edge device, SEAL-Embedded targets Cheon–Kim–Kim–Song (CKKS) [13] HE scheme as it enables operations on real numbers. Unfortunately, this implementation of the library is still not practical. For example, the industry-required frame rate for surveillance cameras and mobile platforms typically ranges from 15 to 60 frames/s [14]. With the SEAL-Embedded library running at 1 GHz on a RISC-V processor, like BlackParrot [15], for a polynomial of degree $N = 4096$ and three 30-bit primes,[1] we are unable to encrypt even a single low-resolution quarter–quarter video graphics array (QQVGA) frames per second (FPS) (further details in Section VI). While one could use a more powerful processor to achieve the required frame rates, it comes at the cost of high power consumption, which is not acceptable in edge devices.

As software solutions are inefficient, several prior works focused on accelerating the key performance bottlenecks within the edge-side operations in hardware. Fig. 2(a) and (b) shows the latency breakdown of the message-to-ciphertext and ciphertext-to-message conversions for different scheme parameters (polynomial degree N and coefficient bit-width $\log Q$) running on BlackParrot using SEAL-Embedded library. For all the parameter sets that we evaluated, the encryption and decryption operations incur the highest latency because they perform multiple polynomial multiplications. The latency of the encryption and decryption operations is dominated by number theoretic transform (NTT) operations. Error sampling is also a bottleneck operation accounting for up to 10% of the total message-to-ciphertext conversion latency.

To address these performance bottlenecks, there exist works that focus on accelerating suboperations like NTT [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26] in en/decryption and pseudorandom number generation (PRNG) [16], [17], [18] in error sampling. For accelerating the complete encryption and decryption operations, Su et al. [27] and Yoon et al. [28] proposed a field-programmable gate array (FPGA)-based and an ASIC-based accelerator, respectively, targeting Brakerski–Gentry–Vaikuntanathan (BGV) HE scheme [29]. Both of these solutions use small scheme parameters ($N <= 2^{10}$ and $\log Q <= 24$). However, these parameters are not practical for most real-world applications because HE schemes contain a noise term (error sample) within the ciphertext coefficients, which is essential for security. This noise within the ciphertext increases with each succeeding homomorphic operation until it reaches a critical level at which it is impossible to recover the computation output [2]. To increase the noise budget for practical HE applications with a large number of HE computations, we need large scheme parameters ($N > 2^{12}$ and $\log Q > 109$).

In this work, we present RISE, a system-on-chip (SoC) containing a RISC-V BlackParrot core and an area- and energy-efficient hardware accelerator that supports large scheme parameters (N and $\log Q$), which enables practical CKKS-based HE applications.[2] To address the performance bottlenecks, in RISE, we accelerate the error sampling and en/decryption operations while reducing the area overhead and energy consumption. To speed up error sampling, we take an efficient and lightweight implementation of a PRNG core [30] and integrate it with fast binomial and uniform samplers. We propose a shared datapath (referred to as a unified datapath later in the article) for the encryption and decryption operations because they both involve similar operations (polynomial addition and multiplication). To reduce ON-chip memory (designed using SRAM) area, we manage the data in RISE such that it does not require memory larger than what is required to store two polynomials. In contrast to prior works [16], [17], [18], [19], [20], [31] that require dual port (1R1W) SRAM banks to access polynomial coefficients for NTT computation, we propose a novel data reordering scheme for NTT so that RISE only needs single port (1RW) SRAM banks, which further reduces the area. Moreover, RISE exploits the data-level parallelism in NTT by leveraging a scalable parallel implementation of butterfly operations.

The main contributions of our work are as follows.

1) We profile edge-side operations for the CKKS scheme by executing the SEAL-Embedded library on BlackParrot core (referred to as baseline in the rest of the article), for a range of scheme parameters (N and $\log Q$) to identify the performance bottlenecks.
2) Based on the profiling results, we architect RISE, an area- and energy-efficient SoC (containing BlackParrot core

[1]These are the largest parameters supported by the SEAL-Embedded library.

[2]In addition to CKKS, RISE can support en/decryption for BFV and BGV. Unlike CKKS, for BFV and BGV schemes, the encryption circuit scales the input messages. This scaling can be done in software on the RISC-V core before sending the message to RISE for encryption. The decryption circuit is the same in all the schemes.

and an accelerator) to accelerate the error sampling and en/decryption operations. We use several optimizations, such as data level parallelism, a shared data path for the en/decryption operations, memory reuse, and data reorder techniques to architect an efficient accelerator design.

3) We evaluate RISE by executing message-to-ciphertext and ciphertext-to-message conversion operations using performance, area, and energy efficiency metrics. Across a range of parameters, RISE reduces the message-to-ciphertext and ciphertext-to-message conversion latency by 28.79×–109.96× and 7.95×–48.49×, respectively, as compared to the baseline. RISE achieves 471.24×–5986.99× lower EDP when performing message-to-ciphertext conversion and 36×–1164.1× lower EDP when performing ciphertext-to-message conversion as compared to baseline. Similarly, RISE has 24.06×–46.83× lower ADP when performing message-to-ciphertext conversion and 6.65×–20.65× lower ADP when performing ciphertext-to-message conversion as compared to baseline.

II. Preliminaries

A. Homomorphic Encryption

The HE computing model allows operating on encrypted data to maintain data privacy. Over the years, a variety of HE schemes have been developed such as BGV [29], Brakerski/Fan–Vercauteren (BFV) [32], and CKKS [13]. The CKKS scheme allows operations on real numbers, which are required for various applications including machine learning, scientific, and graph applications. Hence, we choose to focus on the CKKS scheme in our article, and we use the SEAL-Embedded library to implement it. Below we describe the process for message-to-ciphertext and ciphertext-to-message conversions.

The CKKS scheme works with a native plaintext data type that is a vector of length $N/2$, where each element is chosen from the field of complex numbers $\mathbb{C}$. The encoding operation takes as input this $N/2$-dimensional vector and returns polynomial $\mathsf{m}(X)$ with integer coefficients. The polynomial $\mathsf{m}(X)$ can be encrypted under the public key pk, generating a ciphertext ct by computing

$$\mathsf{c}_0 = \mu \cdot \mathsf{pk}_0 + \mathsf{m} + \mathsf{e}_0 \tag{1}$$

$$\mathsf{c}_1 = \mu \cdot \mathsf{pk}_1 + \mathsf{e}_1. \tag{2}$$

Here, the μ polynomial is sampled from a uniform distribution, and the error polynomials e_0 and e_1 are sampled using a discrete Gaussian noise sampler. The coefficients in the ciphertext polynomials $(\mathsf{c}_0, \mathsf{c}_1)$ are elements of $\mathbb{Z}_Q$, where $\mathbb{Z}$ is a set of integers and Q defines the order of finite field. Here, modulus Q is typically on the order of thousands of bits to account for the noise growth. The CKKS scheme supports the use of a residue number system (RNS) (also known as the Chinese remainder theorem (CRT) representation) to compute such large operands efficiently. Using the RNS approach, each coefficient is represented modulo $Q = \prod_{i=1}^{\ell} q_i$, where each q_i is a prime number. We can represent $x \in \mathbb{Z}_Q$ as a length-ℓ vector of scalars $[x]_{\mathcal{B}} = (x_1, x_2, \ldots, x_\ell)$, where $x_i \equiv x \pmod{q_i}$. We refer to each x_i as a limb of x. The ciphertext is decrypted to obtain the original message back using the following equation:

$$\mathsf{m} = \mathsf{c}_0 + \mathsf{c}_1 \cdot \mathsf{s} \pmod{q_\ell}. \tag{3}$$

Here s is the secret key. Using RNS, both encryption and decryption can be performed with respect to a smaller modulus q_i instead of a large modulus Q.

B. Number Theoretic Transform

Polynomial multiplication is a critical step in encryption and decryption operations. A naïve approach to perform a polynomial multiplication has a complexity of $O(N^2)$ multiplications for a polynomial of degree N. Therefore, to reduce this computational complexity, an NTT operation is applied to the polynomials so as to perform a point-wise multiplication. Using NTT we can reduce the polynomial multiplication complexity to $O(N \log N)$. NTT can be viewed as the finite field version of the fast Fourier transform (FFT). During NTT, coefficients of the input polynomial are multiplied with the power of an Nth primitive root of unity and combined with each other in a butterfly fashion. Before each polynomial multiplication takes place in an encryption operation [see (1) and (2)], the polynomials are converted into an NTT domain. Similarly, we need to perform an inverse NTT (iNTT) operation in the decryption operation. Both NTT and iNTT operations add high computational complexity to the encryption and decryption operations, respectively.

C. BlackParrot: RISC-V Multicore Processor

BlackParrot is an agile open-source RISC-V multicore processor for accelerator SoCs [15]. The BlackParrot multicore implements the RISC-V RV64G architecture and is designed as a scalable, heterogeneously tiled multicore microarchitecture. BlackParrot microarchitecture has four different tile types: 1) a core tile, which contains a BlackParrot processor with one or more coherent caches, a directory shard, and an L2 slice; 2) an L2 extension tile, which is used to scale out the ON-chip L2 in the BlackParrot system; 3) a coherent accelerator tile, which has a local cache engine (LCE) with a backing coherent cache; and 4) a streaming accelerator tile, which does not have a cache memory behind their LCE link and does not control any physical memory. Streaming tiles can be used for basic I/O devices, network interface links, or GPUs.

BlackParrot provides a robust and scalable end-to-end framework for accelerator integration, which simplifies the interfacing of both coherent and streaming accelerators and the offloading of parts of the user application from the processor to the accelerator. This framework provides hardware implementation of streaming and coherent accelerator tiles in SystemVerilog (simulation and FPGA prototype). This helps accelerator designers and system architects to evaluate their accelerator-related ideas using hardware implementation rather than simulation, and find the integration strategy that has low offload and synchronization overheads for their application to improve the end-to-end application time.

D. Video Encryption Example

In this article, we use the example of video encryption to discuss the choice of N and $\log Q$, sizes of the ciphertext, memory, and compute requirements for message-to-ciphertext and ciphertext-to-message conversions, and how that influenced the microarchitecture of RISE. A video is made up of multiple frames, where a frame size is defined by $f_w \times f_h \times b_{\text{pp}}$. Here, b_{pp} defines the bits per pixel and assumes a value of 8 for a grayscale pixel. For a given N, $\log q$, and limbs value, we can encode $N/2 \times \log q$ bits in a single ciphertext, which implies that a single frame will be encoded and encrypted within multiple ciphertexts (cts) and will have a total size of $N \times \log q \times$ limbs $\times$ #cts bits.

For a QQVGA, the frame resolution is 120×160 pixels. If this frame is in grayscale, the frame size will be $120 \times 160 \times 8 = 153\,600$ bits $= 18.75$ KB. With $N = 4096$ and $\log q = 30$ bits, we can encode $N/2 \times \log q = 2048 \times 30 = 61\,440$ bits in a single ciphertext, which implies that a single frame will be encoded and encrypted within three ciphertexts and will have a total size of 270 KB. While $N = 4096$ and $\log Q = 90$ bits parameter set provide 128-bit security, to enable practical applications using HE computing approach, we need to have larger parameters such as $N = 16\,384$ and $\log Q = 432$ bits ($\log q = 54$ bits). For this N and $\log Q$ combination, a single frame will be encoded and encrypted within a single ciphertext and will have a total size of 1.68 MB. Similarly for QVGA, the frame resolution is 320×240 pixels. If this frame is in grayscale, the frame size will be $320 \times 240 \times 8 = 614\,400$ bits $= 75$ KB. With $N = 4096$ and $\log q = 30$ bits, we can convert $N/2 \times \log q = 2048 \times 30 = 61\,440$ bits of a frame in a single ciphertext, which implies that a single frame will be encoded and encrypted within ten ciphertexts and will have a total size of 900 KB. With $N = 16\,384$ and $\log q = 54$ bits, a single frame will be encoded and encrypted within two ciphertexts and will have a total size of 3.37 MB.

Given the limited ON-chip memory in edge devices, we cannot use batch processing for message-to-ciphertext and ciphertext-to-message conversions of the frames. We need to architect RISE such that it can match the throughput of the message-to-ciphertext conversions with the typical frame rates of 15–60 frames/s. In contrast, the ciphertext-to-message conversion is constrained by the bandwidth (100–900 Mbps) of the network connecting the edge device and the cloud.

III. Related Work

Over the years, there have been several works that have focused on accelerating HE computing on the cloud side. These works include algorithmic optimizations for CPU [4], [5] and GPU [6], [7], and custom hardware accelerators [8], [9], [10], [11], [12] running in the cloud. All these works assume that the cloud receives encrypted data from the edge device and that the cloud sends the encrypted result back to the edge device for decryption. There is an implicit assumption in these works that the edge devices have the capability to encrypt and decrypt the data with high performance and do not need any hardware acceleration. However, the encryption and decryption of data for HE computing is compute intensive and has a very high memory usage. For the edge devices that are constrained by power, performance, and area, we need to develop an efficient solution for edge-side operations.

A. Software-Based Solutions

Microsoft SEAL [33] is a HE library that allows addition and multiplication operations on encrypted integers or real numbers. Recently, SEAL has been extended to SEAL-Embedded [3] for resource-constrained edge devices. SEAL-Embedded exploits RNS partitioning, data type compression, memory pooling, and reuse to reduce memory consumption. However, this software-based implementation of encryption operation is still slow and not efficient for real-time applications. As mentioned earlier, for a video application with a low resolution of QQVGA, SEAL-Embedded fails to encrypt even 1 frame/s running at 1 GHz on a RISC-V core like BlackParrot [15] for a practical set of scheme parameters (polynomial degree of $N = 4096$ and three 30-bit primes).

B. Hardware-Based Solutions

There are a few works focusing on accelerating edge side operations for HE [27], [28]. Su et al. [27] present an FPGA-based accelerator for the BGV HE scheme as against the CKKS scheme that we support. Their BGV accelerator only supports small scheme parameters ($N = 128$ and $\log Q = 27$), which are impractical for HE computation. However, we claim that their accelerator can be extended to larger polynomial degrees to support higher security levels, but support for larger parameters is left as future work. Moreover, the accelerator is mainly optimized to achieve high performance and throughput, while ignoring area/energy efficiency. Yoon et al. [28] present an ASIC-based en/decryption accelerator for HE operations. The accelerator is again evaluated only for small parameters ($N = 16$). Even to support these small polynomials, it needs large buffers to store the in/outputs and the precomputed twiddle factors, increasing the memory area.

In our work, we architect an accelerator that can perform message-to-ciphertext and ciphertext-to-message conversions for practical scheme parameters. Our accelerator uses data-level parallelism, shares the datapath between encryption and decryption operations, adopts memory reuse and memory reordering strategies, and eliminates the need for additional ON-chip memory to store twiddle factors by computing them on-the-fly.

IV. RISE System View

In this section, we present the overall design of RISE, an end-to-end SoC (see Fig. 3) that consists of a single BlackParrot RISC-V core, and an accelerator that performs error sampling, encryption, and decryption. The accelerator is interfaced with the BlackParrot core in a streaming fashion because a large amount of data needs to be frequently transferred between the two. To move all the input data from the main memory of BlackParrot core to the accelerator, we configure a hardware DMA logic. The user provides public

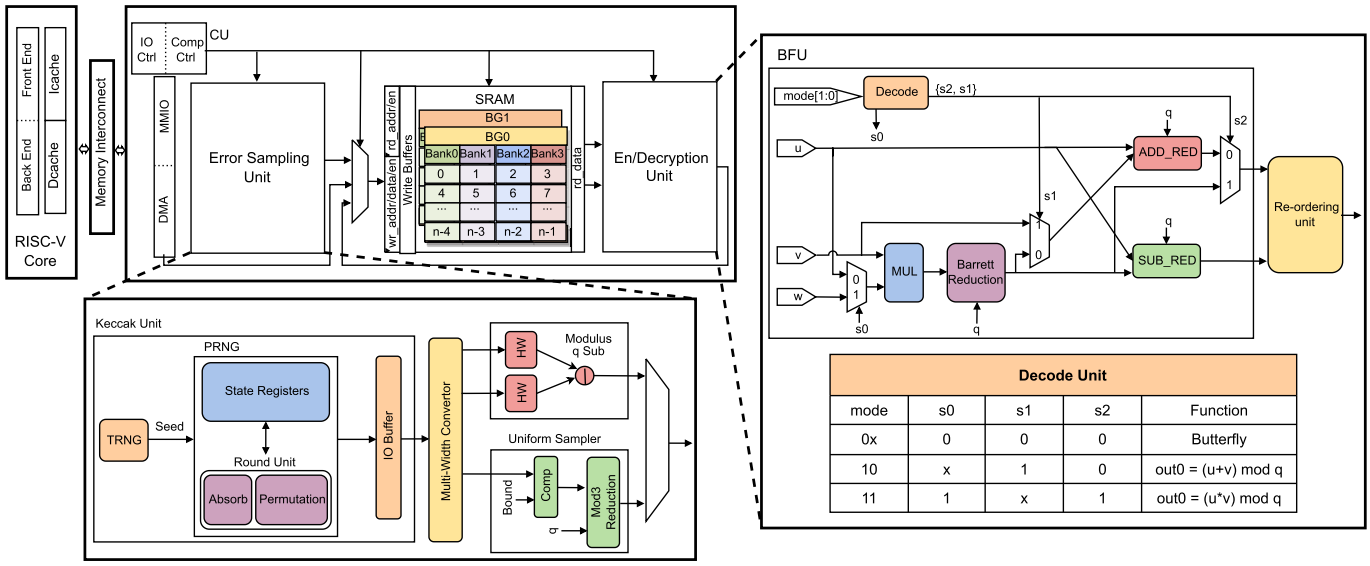

Fig. 3. System-level view of RISE, a RISC-V SoC for accelerating message-to-ciphertext and ciphertext-to-message conversion operations on the edge for supporting homomorphic operations in the cloud.

keys (pk_0, pk_1) and input message m to the BlackParrot core. The BlackParrot core is responsible for performing en/decoding operations and the random seed generation using the SEAL-Embedded library. The PRNG unit in the accelerator receives the random seed from the BlackParrot core and uses it to generate a bit stream of pseudorandom numbers. These pseudorandom numbers are passed to a fast error sampler to generate the required error polynomials, i.e., e_0, e_1, and μ. These error polynomials along with the encoded message and public keys are then used to perform encryption. The encryption operation performs the operations described in (1) and (2). Similarly, the decryption operation performs the operations listed in (3). For the decryption operation, we need the ciphertext (c_0 and c_1) that is sent by the cloud and the secret key that is generated by the BlackParrot core as inputs. Once the en/decryption operation is completed, the BlackParrot core receives an interrupt from the accelerator. Then, the DMA logic transfers the output of the accelerator back to the main memory of the BlackParrot core.

V. Accelerator Microarchitecture

In this section, we provide a detailed description of the microarchitecture of our accelerator (see Fig. 3).

A. Error Sampling Unit

Error samples are critical to maintaining the required security level while performing HE operations. However, generating these high-quality error samples is one of the bottlenecks in edge-side operations. As shown in Fig. 3, error sampling basically consists of two steps: generation of pseudorandom numbers using a true random seed, and generation of uniform and binomially distributed error samples using the generated pseudorandom numbers. Below we present the microarchitecture of a lightweight PRNG, a binomial sampler, and a uniform sampler.

1) Pseudorandom Number Generator: We have a customized PRNG unit as part of the accelerator to speed up PRNG process [30]. One of the prior works [21] evaluated various PRNGs and concluded that the SHA-3 hash family in the SHAKE mode [34], is $2\times$ and $3\times$ more energy efficient than ChaCha20 [35] and AES [36], respectively. This is due to the fact that SHA-3 in SHAKE mode generates the highest number of pseudorandom numbers per round. Therefore, in our PRNG unit design, we use a SHAKE function, which is more commonly referred to as Keccak. For our use case of en/decryption operation that requires a large number of error samples (as N is $>2^{12}$), Keccak makes a perfect PRNG because its output length is not predetermined. Hence, we can generate as many error samples as needed for the en/decryption operation with just one invocation of the Keccak unit.

A Keccak unit typically consists of a round unit with two subunits: absorb and permutation. A true random seed (generated by TRNG) and the desired length of the pseudorandom number, and the rate at which the pseudorandom numbers are generated [provided by the BlackParrot core via control and status registers (CSRs)] are input to the absorb subunit. In our design, the true random seed consists of 1600 bits. A Keccak round operates on the data organized as an array of 5×5 computation lanes, each of length 64. Hence, the absorption phase changes the random seed from a 1-D 1600-bit representation into a 2-D 25×64-bit representation, and we store this 2-D representation in a state register (see Fig. 3). The value in the state register is permuted by performing a series of shift, XOR, AND, and NOT operations in the permutation unit [30]. We store the output of the permutation unit in the state register. We set the length of the pseudorandom number to 1088 bits, which is the maximum length supported by Keccak.

2) Error Sampler: The output of PRNG is passed to a uniform sampler and a binomial sampler to generate error polynomials. For RLWE cryptosystems, the original worst

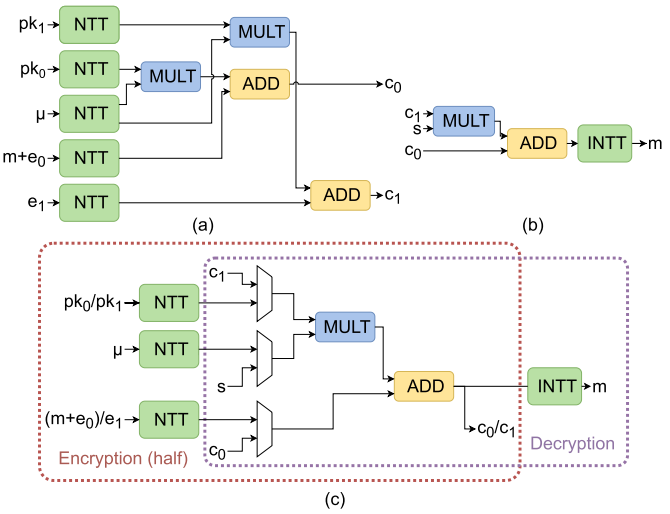

Fig. 4. (a) Encryption dataflow. (b) Decryption dataflow. (c) Unified en/decryption dataflow. In the unified dataflow, encryption, and decryption operations share the datapath and the control logic.

case to average-case security reductions hold for both continuous (rounded) Gaussian distributions and discrete Gaussian distributions. However, the implementation of efficient and constant-time Gaussian sampling is a challenging problem [37]. Therefore, the use of a binomial sampler instead of a Gaussian sampler is a common practice in postquantum encryption schemes [3], [19], [38]. Sampling from a centered binomial distribution B_k can be efficiently performed in constant time by calculating the difference in Hamming weights of two random bit streams of length k. Furthermore, there is currently no known attack that exploits the shape of the error distribution, as long as its standard deviation is sufficiently large. To adhere to the FHE security standard [39], which defines secure parameters for distributions with a standard deviation of 3.2, we use a centered binomial distribution with a standard deviation of $(21/2)^{1/2} \approxeq 3.24$. This choice does not reduce security levels and only increases encryption noise by a negligible amount [3].

In addition, we implement a uniform sampling unit that uniformly samples the coefficients of the polynomial from $\mathcal{R}_3$ (i.e., N coefficients sampled uniformly from $\{-1, 0, 1\}$). We implement this functionality using a rejection sampling algorithm [40]. The implementation is a constant time implementation of modulo3 reduction (see Fig. 3).

B. Encryption and Decryption Unit

Fig. 4(a) shows the encryption datapath, which follows (1) and (2). Each encryption operation calls the accelerator twice, once to compute c_0 with $(\mathsf{pk}_0, \mu, \mathsf{m}, \mathsf{e}_0)$ input set and then to compute c_1 with $(\mathsf{pk}_1, \mu, \mathsf{e}_1)$ input set. The datapath consists of polynomial addition and multiplication operations. The polynomial addition involves simple element-wise modular addition of the polynomial coefficients and has a complexity of $O(N)$. In contrast, polynomial multiplication has a complexity of $O(N^2)$, and like prior efforts, we accelerate it using NTT (more details about NTT are in Section II-B). Acceleration using NTT reduces the complexity of polynomial multiplication to $O(N)$. Similarly, Fig. 4(b) shows the datapath for the decryption operation that follows (3). Decryption datapath again performs polynomial addition and multiplication operation. It receives input polynomials that are already in the NTT domain. However, the decrypted polynomial is required to be in coefficient form for performing the decoding operation (we perform this operation on the BlackParrot core using the SEAL-Embedded library). Therefore, the decryption datapath has an iNTT operation.

1) Unified En/Decryption Datapath: In order to reduce the area overhead of the accelerator, we share the datapath and control logic of the accelerator between encryption and decryption operations [see Fig. 4(c)]. This is possible because the sequence of operations performed in the encryption and decryption operations are the same. Moreover, the encryption operation uses the exact same sequence of operations to compute both c_0 and c_1. Thus, we use the same datapath twice to perform the complete encryption operation.

2) NTT Acceleration: The main performance bottleneck in the en/decryption unit is the NTT operation. Consequently, we propose several optimization techniques to efficiently perform NTT while incurring a low memory and area overhead. We discuss these optimizations in detail in the rest of this section.

a) Butterfly unit (BFU): A butterfly operation is the basic building block of NTT/iNTT operation. An NTT/iNTT operation consists of $\log_2 N$ stages (for a polynomial of degree N), and each stage requires $N/2$ butterfly operations. Each BFU takes two coefficients (say a and b) out of the N polynomial coefficients as input and computes $(a, b) = (a + \omega{\cdot}b \pmod q, a - \omega{\cdot}b \pmod q)$ (refer Algorithm 1 lines 13 and 14). Here, ω is the twiddle factor. A degree N polynomial requires $N/2$ twiddle factors, where each twiddle factor needs $\log q$ bits. Our accelerator computes twiddle factors on-the-fly within BFU to reduce the memory overhead for storing them as precomputed values.

BFU is fully pipelined with the throughput of one butterfly operation per cycle. It is designed to perform NTT, iNTT, polynomial addition, and multiplication operations that are required by both encryption and decryption operations (see Fig. 4). BFU has an integer adder and subtractor unit that performs modular reduction using a conditional operator. BFU contains a modular multiplier where modular reduction operation is performed using a Barrett reduction [41] unit. Barrett reduction computes modular reduction operation without performing any division and only involves two multiplications and one subtraction, shift, and conditional subtraction operation [41]. In addition, it does not exploit any property of the modulus q, which makes it ideal for supporting configurable moduli. The modular multiplier lies on the critical path in the accelerator. Hence, we pipeline the multiplier to reduce the critical path and improve the operating frequency of the accelerator. As power and area are the primary design goals for edge devices, all the above computations are performed by sequentially leveraging the pipelined BFU.

b) Memory reuse technique: All the necessary polynomials (m, e_0, e_1, μ, pk_0, pk_1, c_0, and c_1) should be kept in the accelerator's ON-chip memory for efficient en/decryption

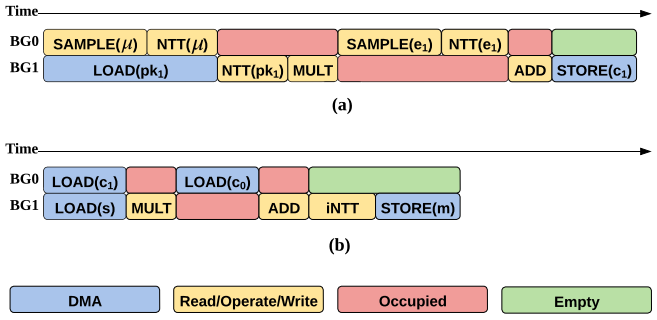

Fig. 5. Memory reuse during (a) encryption and (b) decryption operations. Each BG can store only one polynomial. "Read/Operate/Write" means the bank group is being accessed during the operations. "Occupied" means the bank group stores intermediate results.

computation. A single polynomial typically needs memory of ∼108 KB with $N = 2^{14}$ and $\log q = 54$. Therefore, we require a total of 864 KB to hold all the in/output polynomials. In our memory reuse strategy, we manage the encryption and decryption operations such that at any given time, we need to store a maximum of only two polynomials, which takes 216 KB of space.

Doubling the amount of ON-chip memory (432 KB) doubles the number of independent polynomials that can be stored ON-chip. However, if we want to increase throughput, we need to proportionally double the number of computation units that operate on those additional polynomials. Although this approach increases throughput, it also increases the area. If we halve the amount of ON-chip memory (108 KB), we can only store one polynomial or half of the coefficients of two polynomials in ON-chip memory. NTT computation, which requires a single polynomial ON-chip, can be performed without any stalling. In contrast, operations that require two polynomials (such as polynomial addition and multiplication) can be initiated using the first half of both polynomials but will eventually experience stalling when we need to retrieve the second half of both polynomials from memory. This can lead to degradation in performance.

For memory reuse, we divide the entire ON-chip SRAM memory into multiple banks that are organized into two bank groups, i.e., BG0 and BG1. Each bank group corresponds to a single polynomial and each polynomial is stored across multiple banks within a bank group. During the encryption and decryption operation, we use these bank groups to store the input, output, and intermediate polynomials. Hence, we share each of the two bank groups among several polynomials as shown in Fig. 5(a) and (b). As an illustration (see Fig. 5), we carry out an in-place NTT in an encryption operation that gets the data for polynomial μ from BG0, processes it, and then writes the results back to BG0. While we are still performing NTT on the polynomial μ, we load the next input polynomial pk_1 into BG1 in parallel. The modular addition and multiplication operations involve memory reuse as well. Both of these operations read the input from BG0 and BG1 while writing the output to bank group BG1. Therefore, after the modular addition or multiplication operations are complete, we can reuse BG0 for the subsequent operation. Therefore, by utilizing a memory reuse strategy, we can efficiently perform en/decryption operations while incurring a minimal memory footprint.

c) Memory reorder technique: The next memory level optimization that we perform is memory reorder, which helps reduce the number of memory ports required, resulting in low memory area overhead. Every butterfly operation takes as input two coefficients of the polynomials, operates on them, and stores the resultant values back to the same memory banks. As a result, a naïve implementation of NTT will require two read and two write ports (2R2W) for each memory bank that is of size N. Typically, a 2R2W memory bank is roughly twice as large as one read and one write port (1R1W) memory bank of the same size. Consequently, we can save half of the memory area simply by switching from a single 2R2W bank of size N to two 1R1W banks of size $N/2$.

However, managing memory access patterns for NTT, with 1R1W banks, becomes challenging as the memory accesses can lead to bank conflicts. Throughout all the NTT stages, the distance $(j-i)$ between the two inputs of a butterfly operation changes. This leads to bank conflicts in several stages of NTT as each stage in NTT iterates through all values from 1 to $N/2$. Thus, replacing 2R2W bank with 1R1W banks is not trivial. Some of the prior works [16], [17], [18], [19], [21], [22], [38] address this issue by modifying the NTT algorithm itself. For example, to use 1R1W memory banks for an NTT, Roy et al. [22] proposed a memory-efficient NTT algorithm, which we refer to as the NTT_swap2 algorithm. Their technique rearranges the output of the two subsequent butterfly operations to prevent bank conflicts (two 1R1W banks). As a result, it guarantees that the input pair required by the butterfly operation in the following stage is in distinct memory banks.

Although using a 1R1W memory bank reduces memory space by half, there is still scope for improvement. To further reduce memory area overhead, we suggest replacing two 1R1W banks of size $N/2$ with four one-read/write port (1RW) banks of size $N/4$. This causes newer bank conflicts, which cannot be addressed by using the NTT_swap2 method. For example, if a bank receives both read and write requests at the same time, we will need an additional write buffer to store the write requests. Now, the write requests must wait in the write buffer until there are no incoming reads before opportunistically writing back the results. Although using a write buffer is a good way to solve bank conflicts, the size of the write buffer quickly grows. Our observation is that if there are $N/4$ continuous read and write accesses to the same bank in a given stage, the write buffer must be the same size as the banks $(N/4)$ in order to hold all write requests that overlap with read requests to the same bank. If we were to use the same size buffers as the memory banks, we incur the same memory overhead as the 1R1W memory bank, making this solution impractical.

We propose a method called NTT_swap4 (refer Algorithm 1) to avoid using these large write buffers. NTT_swap4 reorders the output of four successive butterfly

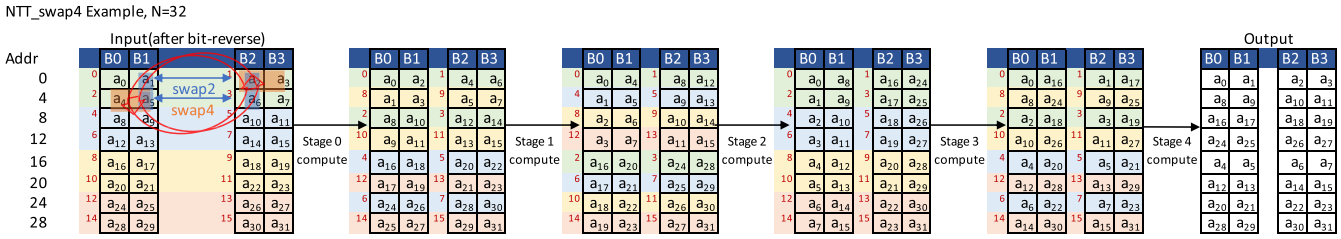

Fig. 6. NTT_swap4 with $N = 32$. The red-colored numbers before each pair of cells denote the order of butterfly operations. The four consecutive butterfly operations (two rows) being reordered are denoted with the same color.

operations, while NTT_swap2 reorders the result of only two butterfly operations (see Fig. 6). This is to ensure that not only the two inputs of each butterfly operation are stored in different banks (like NTT_swap2) but also the inputs of consecutive butterfly operations are stored in different banks (NTT_swap4). As the same bank is not repeatedly used in this scenario, the write buffer can immediately write back the outcomes in the subsequent cycle. Thus, the write buffer can be as small as one element wide ($\log q$) for a memory bank. We demonstrate NTT_swap4 technique example (for $N = 32$) in Fig. 6. The order of the butterfly operations is indicated by the numbers (in red) before each pair of cells. For example, in stage 0, the first four butterfly operations access the following pairs: (a_0, a_1), (a_2, a_3), (a_4, a_5), and (a_6, a_7). However, stage 1 expects elements in the order of (a_0, a_2), (a_4, a_6), (a_1, a_3), and (a_5, a_7). To prevent successive butterfly operations in stage 1 from accessing the same banks for reads and writes, we reorganize stage 0's outputs into the order anticipated by stage 1 (refer Algorithm 1 line 19). To carry out this reordering, we use a reordering unit (RU).

d) Reordering unit: The RU reorders the output generated by the BFU and writes it back into the memory banks. A small register array that can store up to eight pairs of butterfly outputs and a reordering logic make up the RU. Reordering logic begins by sequentially writing the two results of a butterfly operation and their addresses to the register array in each cycle. Once there are eight elements in the register array or four pairs of BFU outputs, the reordering logic will send out the elements stored in the registers to the corresponding memory bank. Both NTT and iNTT operations can be reordered effectively using RU. The RU will be active only while doing NTT/iNTT computations based on the mode signal (see Fig. 3).

e) Control unit: The control unit (CU) consists of two components—the computation controller and the I/O controller. Based on the current operation (error sampling, NTT/iNTT, modular addition, and multiplication), the computation controller, which is an FSM, chooses the BFU and RU mode signals. In addition, it generates the enable signal and read/write addresses for memory bank accesses. During NTT/iNTT operation, the computation controller is also in charge of setting up the NTT unit to compute the twiddle factors on-the-fly. Depending on the type of CPU request received by the accelerator (encryption or decryption), the I/O controller chooses the necessary set of BFU operations. Besides, based on the current en/decryption stage, it also configures the DMA unit for the input/output data transfer to/from memory banks.

Algorithm 1 NTT_swap4

Input: Polynomial $a(x) \in Z_q[x]$ in bit-reversed order
Output: $NTT(a(x))$ in normal order

1 $m = 2$;
2 **for** $(stage = 0; stage < (\log N - 1); stage+ = 1)$ **do**
3 $\quad \omega = 1$; $\omega_m = \omega_n^{2^{\log N - 1 - stage}}$; $upd\_cnt = 1$;
4 $\quad$ **for** $(j = 0; j < m * 2; j+ = 4)$ **do**
5 $\quad\quad$ **for** $(k = 0; k < N; k+ = m * 4)$ **do**
6 $\quad\quad\quad$ i0=[]; i1=[];
7 $\quad\quad\quad$ **for** $(l = 0; l < 4; l+ = 1)$ **do**
8 $\quad\quad\quad\quad$ **switch** l **do**
9 $\quad\quad\quad\quad\quad$ **case** 0 **do** $idx = j + k$;
10 $\quad\quad\quad\quad\quad$ **case** 1 **do** $idx = j + k + 2$;
11 $\quad\quad\quad\quad\quad$ **case** 2 **do** $idx = j + k + m * 2$;
12 $\quad\quad\quad\quad\quad$ **case** 3 **do** $idx = j + k + m * 2 + 2$;
13 $\quad\quad\quad\quad$ $a[idx] = a[idx] + a[idx + 1] * \omega \pmod q$;
14 $\quad\quad\quad\quad$ $a[idx+1] = a[idx] - a[idx+1] * \omega \pmod q$;
15 $\quad\quad\quad\quad$ i0.append(idx); i1.append($idx + 1$);
16 $\quad\quad\quad\quad$ **if** $upd\_cnt == N/(2^{stage+1})$ **then**
17 $\quad\quad\quad\quad\quad$ $\omega = \omega * \omega_m \pmod q$; $upd\_cnt = 1$;
18 $\quad\quad\quad\quad$ **else** $upd\_cnt+ = 1$;
19 $\quad\quad\quad$ $(a[i0[0]], a[i1[0]], a[i0[1]], a[i1[1]],$ $a[i0[2]], a[i1[2]], a[i0[3]], a[i1[3]]) =$ $(a[i0[0]], a[i0[1]], a[i0[2]], a[i0[3]],$ $a[i1[0]], a[i1[1]], a[i1[2]], a[i1[3]])$
20 $\quad m = (m == N/4)\ ?\ 2 : (m * 2)$;
21 **for** $(i = 0; i < N; i+ = 1)$ **do**
$\quad$ /* Bit manipulation */
22 $\quad phy\_addr = \{i[\log N - 3 : 2], i[\log N - 1 : \log N - 2], i[1 : 0]\}$;
23 $\quad a\_out[i] = a[phy\_addr]$;
24 return a_out;

C. Further Optimizations to NTT

In this section, we present a technique to parallelize NTT to further improve its performance. This is due to the fact that the area-efficient NTT design that we discussed above cannot meet the performance requirements of high-end edge devices and several high-speed applications. Therefore, by parallelizing the NTT computation, we can improve the performance at the cost of area and power overhead. We evaluate this performance versus area/power trade-off to identify the optimal architectures for different design objectives in Section VI.

To improve the performance of NTT computation, we can perform multiple butterfly operations in parallel. Therefore,

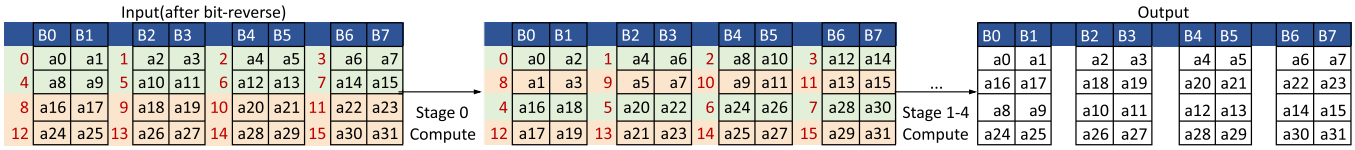

Fig. 7. Modified NTT_swap4 with $N = 32$ and two parallel BFUs. The red-colored numbers before each pair of cells denote the order of butterfly operations. The eight consecutive butterfly operations (two rows) being reordered are denoted with the same color.

we propose a scalable parallel implementation of NTT with multiple BFUs. To support a parallel NTT architecture using multiple BFUs, we need to address two main requirements: 1) multiport memory banks to read/write multiple BFUs' inputs and outputs simultaneously and 2) on-the-fly computation of multiple twiddle factors to enable multiple butterfly operations in parallel.

1) Memory Bank Organization for Parallel NTT: Moving to a multiport memory bank design is not an efficient solution as increasing the number of ports will quadratically increase the memory area overhead [42]. To reduce the memory area overhead, we still use the 1RW memory banks but we linearly increase the number of memory banks as we increase the number of BFUs. However, we keep the total memory size the same by proportionally decreasing the size of each memory bank. With an increase in the number of memory banks, the data access pattern within each stage of the NTT becomes complicated resulting in data dependencies that need to be carefully managed. Consequently, our proposed memory reorder technique (see Section II-B) will not work as it is and requires modifications.

In our NTT_swap4 technique, we can eliminate memory access conflicts by reordering the output of four successive butterfly operations. However, in our parallel NTT implementation, we need to increase the number of memory banks to allow for reading and writing multiple BFUs' inputs and outputs simultaneously. This leads to a more complex data access pattern within each NTT stage, resulting in data dependencies that require careful management. Therefore, we have extended our memory reorder scheme to eliminate memory bank conflicts by reordering the output of $4 \times$ #BFUs butterfly operations, rather than only four successive butterfly operations. Fig. 7 illustrates the modified NTT_swap4 technique example for $N = 32$ with two parallel BFUs. The order of the butterfly operations is denoted by the red numbers before each pair of cells. For instance, in stage 0, the first eight butterfly operations access the following pairs: (a_0, a_1), (a_2, a_3), (a_4, a_5), (a_6, a_7), (a_8, a_9), (a_{10}, a_{11}), (a_{12}, a_{13}), and (a_{14}, a_{15}). However, stage 1 expects elements in the order of (a_0, a_2), (a_4, a_6), (a_8, a_{10}), (a_{12}, a_{14}), (a_1, a_3), (a_5, a_7), (a_9, a_{11}), and (a_{13}, a_{15}). To prevent successive butterfly operations in stage 1 from accessing the same banks for reads and writes, we reorganize stage 0's outputs into the order anticipated by stage 1.

2) Twiddle Factor Computation for Parallel NTT: As discussed earlier, to minimize the memory area overhead RISE computes the required twiddle factors on-the-fly instead of storing the precomputed values. However, now, as we increase the number of BFUs for the parallel NTT approach, we need to compute many twiddle factors in parallel, thus introducing stalls in the NTT computation pipeline that offsets the performance gains. The stalls are introduced because RISE's area-efficient design shares the BFU to compute the twiddle factor and to perform the butterfly operation. To eliminate pipeline stalls, we introduce a separate modular multiplier to compute twiddle factors in parallel with the butterfly operations. We note that we also need to increase the number of modular multipliers that are used to compute twiddle factors, as we increase the number of BFUs.

VI. Evaluation

A. Methodology

For our analysis, we run all edge-side operations on the following systems in bare-metal mode.

1) *Baseline:* BP processor executes all the operations from the SEAL-Embedded library.
2) *RACE [42]:* In the RACE SoC, the hardware accelerator executes the en/decryption operation, while the remaining operations (error sampling and en/decoding) are performed on the BP processor. We modified the SEAL-Embedded library to invoke calls to en/decryption operations on the accelerator.
3) *RISE:* In the RISE SoC, the hardware accelerator performs error sampling and executes the en/decryption operation while the remaining operations (en/decoding) are performed on the BP processor. RISE supports a range of parallel BFUs (1–32) within a single NTT operation. In our evaluation, RISE-1BFU and RISE-MaxBFU correspond to configurations with one BFU and 32 BFUs, respectively.

All three systems, i.e., baseline, RACE, and RISE, make use of a single core BP configuration with an eight-stage pipeline, 32 KB each of L1 Icache and Dcache, and 128 KB of L2 Cache running at 1 GHz. The network-on-chip (NoC) in the BP is divided into three classes: coherence, DRAM, and I/O networks, each with different channel widths of 128, 64, and 64 bit, respectively.[3]

We implement all three systems in SystemVerilog and simulate them using VCS. The hardware implementation is cycle-accurate and captures the nuances of data movement between all parts of the systems. For power, performance, and area evaluation, we use GlobalFoundries' 12-nm technology.

[3]A complete list of SoC configuration parameters used in this article is available online at https://github.com/black-parrot/black-parrot/blob/master/bp_common/src/include/bp_common_aviary_pkgdef.svh

TABLE I

NTT OPERATION PERFORMANCE (CYCLE COUNT) COMPARISON WITH THE STATE-OF-THE-ART DESIGNS IN RELATED WORKS. A HEAD-TO-HEAD COMPARISON IN TERMS OF FREQUENCY, POWER, AND AREA NUMBERS CANNOT BE DONE BECAUSE OF DIFFERENCES IN PLATFORMS (ASIC VERSUS FPGA) AND TECHNOLOGY NODES

Design	N	$\log q$	Latency (Clock Cycles)	SRAM		Platform
				Size (KB)	R/W Ports	
[16]	256	16	18554	2.25	Dual	FPGA
[22]		14	1692	4.5	Dual	FPGA
[21]		24	1289	45	Single	ASIC
[17]		16	556	13.5	Dual	FPGA
[18]		14	327	22.5	Dual	FPGA
RISE		30	103	0.93	Single	ASIC
[17]		16	38	10	Dual	FPGA
[22]	512	16	3443	6.75	Dual	FPGA
[23]		16	1074	18	Dual	FPGA
RISE		30	215	1.87	Single	ASIC
[24]	1024	28	2568	108	Dual	FPGA
[23]		28	2114	27	Dual	FPGA
[25]		32	650	355.5	Dual	FPGA
RISE		30	447	3.75	Single	ASIC
[23]	4096	60	8284	110.25	Dual	FPGA
[25]		32	3075	355.5	Dual	FPGA
RISE		30	1918	15	Single	ASIC
[26]	65536	60	536832	616.5	Dual	FPGA
[43]		30	47795	427.5	Dual	FPGA
RISE		60	34814	480	Single	ASIC

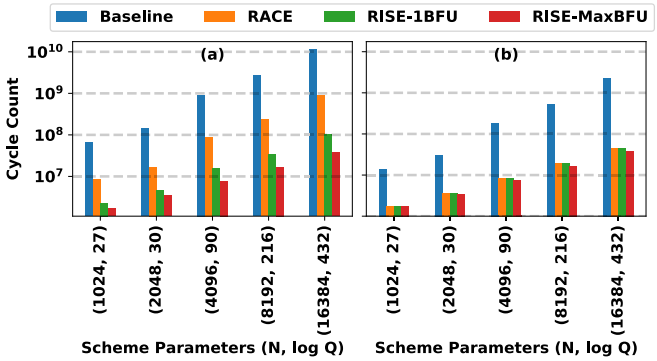

Fig. 8. Latency (in clock cycle count) of (a) message-to-ciphertext and (b) ciphertext-to-message conversion operations for baseline, RACE, RISE-1BFU, and RISE-MaxBFU with 1 GHz frequency.

We synthesize the logic components in baseline, RACE, and RISE using Synopsys design compiler, and use memory compiler for designing the SRAM arrays.

B. Performance Results

We evaluate RISE performance with different numbers of BFUs (1–32) for both message-to-ciphertext and ciphertext-to-message conversion operations for a range of scheme parameters.

As shown in Fig. 8(a), across different scheme parameter (N, $\log Q$) values, RACE configuration achieves 7.8×–12.67× and 7.9×–48.49× better performance for message-to-ciphertext and ciphertext-to-message conversion operations, respectively, compared to the baseline. The performance improvement in RACE is because we offload the encryption and decryption operations to the hardware accelerator, which speeds up encryption by 89.82×–156.11× and decryption by 204.66×–244.44×. In RISE-1BFU configuration, on top of encryption and decryption operations, we also offload the error sampling operation to the hardware accelerator, which results in 1726.39×-1734.08× speed up in the error sampling. However, RISE-1BFU configuration achieves just 3.68×-8.67× better performance for message-to-ciphertext conversion operation compared to the RACE as the performance improvement is limited by Amdahl's law.

RISE-1BFU configuration achieves similar performance as RACE for ciphertext-to-message conversion operation [refer Fig. 8(b)], as this conversion does not include the error sampling step. As we increase the number of BFUs within the NTT/iNTT operation to perform multiple butterfly operations in parallel, we observe a speed-up in encryption and decryption operations. As shown in Fig. 8(a) and (b), for RISE-MaxBFU configuration, with maximum number of BFUs (32), compared to RISE-1BFU the message-to-ciphertext and ciphertext-to-message conversion performance improves by 37.7%–272.58% and 3.4%–23.36%, respectively. Overall, compared to the baseline system, our RACE-MaxBFU improves the message-to-ciphertext conversion performance by 38.27×–299.75×, and the ciphertext-to-message conversion performance by 8.2×–59.81×.

1) Comparison With Related Works: Table I presents the performance comparison between RISE and other relevant state-of-the-art works. The performance comparison includes the latency of NTT operation for a single limb (the dominant operation in message-to-ciphertext and ciphertext-to-message conversions), the SRAM size, the number of memory ports, and the evaluation platform (ASIC or FPGA). As other existing works use different parameters (N and $\log q$), we compute the performance numbers for RISE for all of these parameter sets. It is evident from the table that RISE performs faster

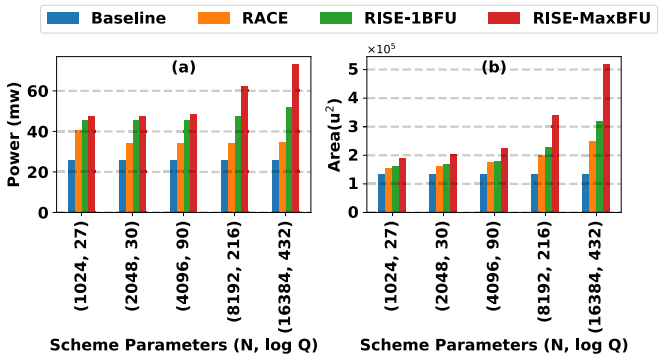

Fig. 9. (a) Power consumption and (b) area utilization for baseline, RACE, RISE-1BFU, and RISE-MaxBFU.

NTT computation when compared to other designs for all values of N except for [17]. This is because of our highly parallel and pipelined NTT computation design that leads to low NTT computation latency. Moreover, for every value of N, RISE utilizes only single-port memory with the smallest SRAM size. Li and Liu [17] can perform a single NTT in 38 cycles as they store precomputed twiddle factors in SRAM, which leads to 10× higher memory requirement than RISE. In addition, they need dual-port memories to feed the input to their vectorized NTT unit. RISE manages to perform NTT computations while using only a single-port memory by leveraging the NTT_swap4 method.

Compared to the related work, RISE has the smallest memory footprint because of our on-the-fly twiddle factor generation unit, in-place NTT computation, memory reuse, and memory reorder techniques. Roy et al. [22] proposes a cryptoprocessor for RLWE-based encryption schemes that utilize the NTT_swap2 algorithm. Their cryptoprocessor requires storing all the input polynomials (i.e., public and private keys, error samples, and input message) in ON-chip memory, resulting in significant memory overhead. Specifically, their design requires six memory blocks, each of size $\log q \times N$, and all six memory blocks are dual-port as they utilize the NTT swap2 algorithm. In contrast, RISE employs a memory reuse strategy to manage encryption and decryption operations, which requires a maximum of only two polynomials to be stored in ON-chip memory at any given time. In addition, RISE outperforms its cryptoprocessor in both encryption and decryption operations due to its highly parallel NTT computation. For instance, for N and $\log q$ values of (256 and 14) and (512 and 16), their cryptoprocessor takes 6299/2840 cycles and 13 171/5799 cycles for en/decryption operations, respectively. However, RISE only needs 4655/457 and 9464/929 cycles to perform en/decryption operations for the same scheme parameters. We do not provide a head-to-head comparison of the performance of RISE with the works by [27] and [28] as those prior works accelerate en/decryption operations to support HE operations for the BGV scheme while we enable the support for CKKS scheme.

C. Power/Energy Results

Fig. 9(a) shows the power consumption in the message-to-ciphertext and ciphertext-to-message conversion operations for different scheme parameter (N, $\log Q$) values when using baseline, RACE, RISE-1BFU, and RISE-MaxBFU systems. Message-to-ciphertext and ciphertext-to-message conversion operations have similar power consumption, within 0.01%, and we report the power consumption for the message-to-ciphertext.[4] The total power consumption for a message-to-ciphertext and ciphertext-to-message conversions in the baseline system is 27.19 mW, out of which the SRAM power consumption is 41.49% = 11.4 mW and the digital logic consumes the remaining power. Overall, the power consumption of RACE is about 25%–28% (for a range of scheme parameters) higher than the baseline system for both message-to-ciphertext and ciphertext-to-message conversion operations. The increase in the power consumption is due to 41.92%–43.55% and 3.36%–7.81% power increase in the digital logic and SRAM, respectively. The power consumption in RISE-1BFU configuration increases by 11.62%–49.35% compared to RACE due to the additional digital logic required for the error sampling unit. As we increase the number of BFUs from one to 32, the power consumption increases by 4.49%–40.12% due to the more complex memory banking logic (14.61%–244.46%) and multiple parallel BFUs (1.01%–6.56%).

D. Area Results

The area of RACE is from 15% (smallest N) to 84% (largest N) larger than the area of the baseline system. [see Fig. 9(b)]. This increase in the area is due to the area required by the accelerator where SRAMs primarily contribute to the increase in area. The area overhead in RISE-1BFU is (3.65%–27.21%) compared to RACE, as error sampling contributes very small to the overall area of RISE-1BFU With an increase in the number of parallel BFUs, the complexity of control logic and memory banking increases. Hence, as shown in Fig. 9(b), by increasing the number of BFUs from one to 32, the area overhead of RISE-MaxBFU increases by 17.59%–62.66% as compared to RISE-1BFU for different scheme parameters.

E. Area and Energy Efficiency

RISE aims to improve the performance of message-to-ciphertext and ciphertext-to-message conversions at the cost of an increase in area and power. Thus, area-delay product (ADP) and energy-delay product (EDP) metrics need to be considered for evaluating our RISE design. Fig. 10(a) and (b) compares the ADP value for baseline, RACE, RISE-1BFU, and RISE-MaxBFU systems. As we can see, RACE decreases the message-to-ciphertext and ciphertext-to-message conversion ADP by 6.76×–6.86× and 6.89×–26.27× compared to the baseline, respectively. The improvement is the result of 7.8×–12.67× and 7.9×–48.49× improvement in performance while incurring only a 15%–84% increase in the area.

[4]The ciphertext-to-message conversion does not perform the error sampling operation and so should have lower power consumption than the message-to-ciphertext conversion. However, we did not power gate or clock gate the error sampling unit during the ciphertext-to-message conversion and so it consumes some power even during the ciphertext-to-message conversion. The error sampling operation takes less than 10% of the total time required to perform message-to-ciphertext conversion, and so is not the dominant component. Hence, the power consumed during message-to-ciphertext and ciphertext-to-message conversion are comparable.

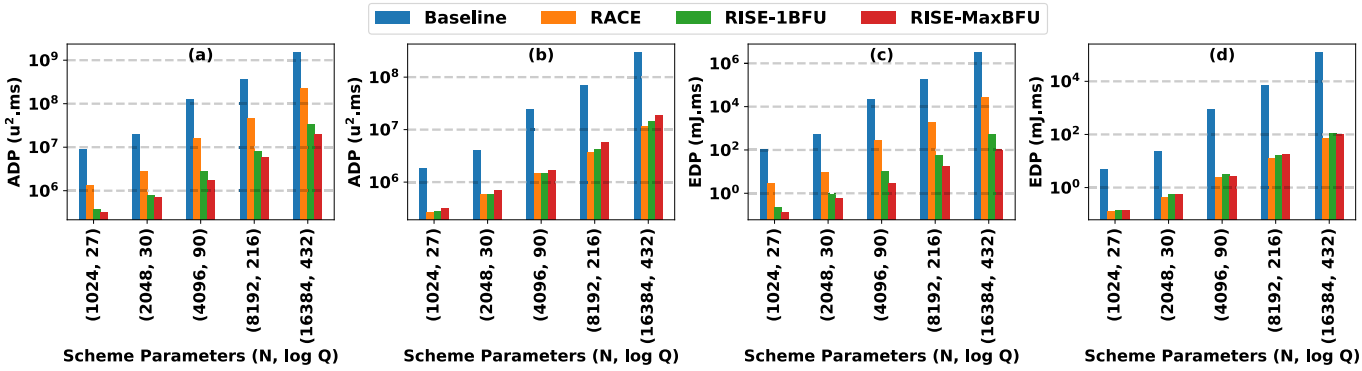

Fig. 10. (a) and (b) ADP and (c) and (d) EDP of message-to-ciphertext/ciphertext-to-message conversion operations for baseline, RACE, RISE-1BFU, and RISE-MaxBFU.

Fig. 10(a) also shows that in RISE-1BFU, the message-to-ciphertext conversion ADP outperforms RACE (3.55×–6.82× lower). This is due to 3.68×–8.67× performance improvement while incurring only a 3.65%–27.21% increase in area. Increasing the number of BFUs from one to 32 improves the message-to-ciphertext conversion ADP by 1.13×–1.67× for different scheme parameters (due to 1.37×–2.72× performance improvement and 17.59%–62.66% area overhead). For the ciphertext-to-message conversion the ADP [refer Fig. 10(b)] of the RISE-1BFU system underperforms RACE by 3.65% for the smallest N and 27.21% for the largest N as there is an increase in area overhead due to the additional error sampling unit, which is not used by ciphertext-to-message conversion. Increasing the number of BFUs to 32 worsens the ciphertext-to-message conversion ADP of RISE-1BFU by up to 12.3% for small N values. This is because, in RISE-1BFU, the decryption operation only accounts for 3.43%–10.40% of the total latency, which when improved by adding parallel BFUs within iNTT, does not improve the performance by the same proportion as the area overhead (17.59%–62.66%). For large N values, the ciphertext-to-message conversion ADP increases by up to 31.85% as now decryption operation contributes significantly to the total latency, which can be accelerated by instantiating parallel BFUs.

Fig. 10(c) and (d) compares the energy efficiency of baseline, RACE, RISE-1BFU, and RISE-MaxBFU systems. As evident from the figures, EDP follows a similar trend as ADP. There is 38.6×–118.76× and 40.19×–1738.68× improvement in the EDP for message-to-ciphertext and ciphertext-to-message conversions, respectively, when using RACE as compared to the baseline. The EDP of RISE-1BFU for message-to-ciphertext conversion is 12.19×–50.4× better compared to RACE and by increasing the number of parallel BFUs, EDP further improves by 1.69×–5.3×. Unfortunately, EDP of RISE-1BFU for ciphertext-to-message conversion worsens by 11.62%–49.35% compared to RACE for the same reason as ADP. The EDP of RISE-1BFU for ciphertext-to-message conversion can be improved by clockgating the error sampling unit. Moreover, by using 32 BFUs the ciphertext-to-message conversion EDP improves by 1.71%–8.6% compared to RISE-1BFU. This improvement is due to the fact that increasing the number of BFUs improves decryption operation performance, which leads to up to 23.36% performance improvement for ciphertext-to-message conversion. We also get up to 21.69% energy consumption reduction as the Black-Parrot core consumes less idle energy.

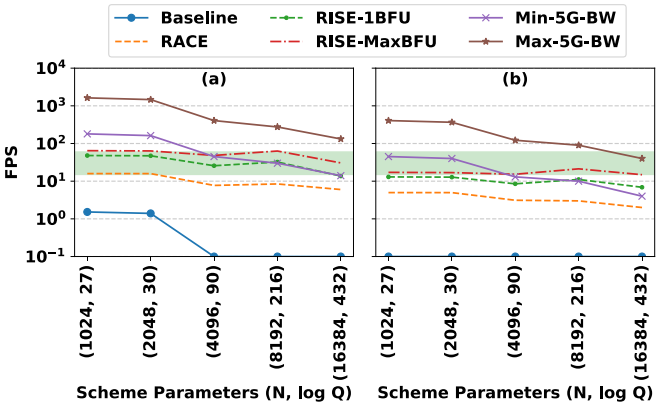

Fig. 11. Maximum supported (a) QQVGA and (b) QVGA frame rate per second for mid-band 5G, baseline, RACE, RISE-1BFU, and RISE-MaxBFU for different N and log Q values. The green region indicates the typical frame per second required for surveillance cameras and mobile platforms.

F. Video Application Evaluation

We evaluate our RISE design using QQVGA and QVGA video frame encryption examples. For calculating the number of ciphertexts required to encode and encrypt each of these frames refer to Section II-D. Fig. 11(a) and (b) shows the maximum FPS that the baseline, RACE, RISE-1BFU, and RISE-MaxBFU systems can sustain for different scheme parameter (N, log Q) values when performing message-to-ciphertext conversion operation for QQVGA and QVGA, respectively. The frames are sent to the cloud using a mid-band 5G network, which offers a balance between speed, capacity, and coverage [44]. As shown in Fig. 11, in the regions with maximum bandwidth, mid-band 5G network can transfer up to 132 (QQVGA) and 40 (QVGA) frames/s for the largest N value and in the regions with minimum bandwidth, it can only transfer 14 (QQVGA) and 4 (QVGA) frames/s for the smallest N value. So the throughput of our designs for

message-to-ciphertext and ciphertext-to-message conversions should match with these frame rates.

The baseline system is capable of encrypting up to 2 QQVGA frames/s for N values smaller than 2048 [refer Fig. 11(b)]. However, as we increase N to 4096 or larger values, it cannot encrypt even a single frame/s. On the other hand, for QQVGA, RACE encrypts ∼16 frames/s for small values of N and 6 frames/s for the largest N value (16 384). So at large values of N we cannot saturate the 5G network at both the maximum bandwidth and minimum bandwidth.

For QVGA resolution, the baseline system cannot encrypt even 1 frames/s even for the smallest N value (1024). However, RACE can encrypt 5 and 2 frames/s for the smallest and largest N values, respectively. While RACE can support higher FPS than the baseline, it cannot saturate the 5G network at both the maximum bandwidth and minimum bandwidth for QVGA.

The RISE-1BFU system is capable of encrypting up to 48 QQVGA frames/s for N values smaller than 2048 [refer Fig. 11(b)]. For the largest N value, RISE-1BFU system is capable of encrypting up to 13 QQVGA frames/s. As we increase the number of BFUs from 1 to 32, the FPS numbers change to 64 and 30 QQVGA frames/s for the smallest and largest N values, respectively. Thus, we can saturate the 5G network at minimum bandwidth but not at the maximum bandwidth.

For QVGA resolution [refer Fig. 11(a)], RISE-1BFU system is capable of encrypting up to 13 frames/s for the smallest N value (1024) and 7 frames/s for the largest N value. The RISE-MaxBFU configuration can encrypt up to 17 and 14 QVGA frames/s for the smallest and largest N values, respectively. Thus, we can saturate the 5G network at minimum bandwidth but not at the maximum bandwidth.

Typically surveillance cameras and mobile platforms have an average frame rate of 15–30 frames/s [14] [shown by the green highlighted area in Fig. 11(a) and (b)]. For QQVGA resolution, RISE-MaxBFU meets this FPS requirement for all N and $\log Q$ combinations. For QVGA, RISE-MaxBFU can barely meet the FPS requirement for smaller values of N and $\log Q$.

VII. Conclusion

In this work, we present RISE, a RISC-V-based SoC for message-to-ciphertext and ciphertext-to-message conversion accelerations on the edge to support HE operations in the cloud. RISE implements several optimizations that enable high performance, and area- and energy-efficient message-to-ciphertext and ciphertext-to-message conversion operations. These optimizations include data-level parallelism, unified encryption and decryption datapath, memory reuse and memory reordering strategies, and on-the-fly twiddle factor computation. Our analysis shows that compared to the baseline and RACE, RISE achieves higher performance with lower energy consumption. As a result, overall RISE is more area and energy efficient than the baseline and RACE system. Across different N and $\log Q$ parameters, RISE has 471.24×–5986.99× lower EDP when running a message-to-ciphertext conversion and 36×–1164.1× lower EDP when running ciphertext-to-message conversion as compared to baseline. Similarly, across different N and $\log Q$ parameters, RISE has 24.06×–46.83× lower ADP when running a message-to-ciphertext conversion and 6.65×–20.65× lower ADP when running a ciphertext-to-message conversion as compared to baseline.

Acknowledgment

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] 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.

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

[3] D. Natarajan and W. Dai, "SEAL-embedded: A homomorphic encryption library for the Internet of Things," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2021, no. 3, pp. 756–779, Jul. 2021.

[4] W. Jung et al., "Accelerating fully homomorphic encryption through architecture-centric analysis and optimization," *IEEE Access*, vol. 9, pp. 98772–98789, 2021.

[5] C. Bootland, W. Castryck, I. Iliashenko, and F. Vercauteren, "Efficiently processing complex-valued data in homomorphic encryption," *J. Math. Cryptol.*, vol. 14, no. 1, pp. 55–65, Jun. 2020.

[6] A. Al Badawi, B. Veeravalli, J. Lin, N. Xiao, M. Kazuaki, and A. Khin Mi Mi, "Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 2, pp. 379–391, Feb. 2021.

[7] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: FrodoKEM, NewHope, and kyber," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 575–586, Mar. 2021.

[8] 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.

[9] D. B. Cousins, K. Rohloff, and D. Sumorok, "Designing an FPGA-accelerated homomorphic encryption co-processor," *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 2, pp. 193–206, Apr. 2017.

[10] D. Reis, J. Takeshita, T. Jung, M. Niemier, and X. S. Hu, "Computing-in-memory for performance and energy-efficient homomorphic encryption," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 11, pp. 2300–2313, Nov. 2020.

[11] F. Turan, S. S. Roy, and I. Verbauwhede, "HEAWS: An accelerator for homomorphic encryption on the Amazon AWS FPGA," *IEEE Trans. Comput.*, vol. 69, no. 8, pp. 1185–1196, Aug. 2020.

[12] M. S. Riazi, K. Laine, B. Pelton, and W. Dai, "HEAX: An architecture for computing on encrypted data," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2020, pp. 1295–1309.

[13] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT*, Hong Kong: Springer, Dec. 2017, pp. 409–437.

[14] M. A. Usman, M. R. Usman, and S. Y. Shin, "An intrusion oriented heuristic for efficient resource management in end-to-end wireless video surveillance systems," in *Proc. 15th IEEE Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2018, pp. 1–6.

[15] D. Petrisko et al., "BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, Jul. 2020.

[16] P. Nannipieri, S. Di Matteo, L. Zulberti, F. Albicocchi, S. Saponara, and L. Fanucci, "A RISC-V post quantum cryptography instruction set extension for number theoretic transform to speed-up CRYSTALS algorithms," *IEEE Access*, vol. 9, pp. 150798–150808, 2021.

[17] C. Li, W. Zhu, and L. Liu, "A high speed NTT accelerator for lattice-based cryptography," in *Proc. Int. Conf. Commun., Inf. Syst. Comput. Eng. (CISCE)*, May 2021, pp. 85–89.

[18] X. Chen, B. Yang, S. Yin, S. Wei, and L. Liu, "CFNTT: Scalable radix-2/4 NTT multiplication architecture with an efficient conflict-free memory mapping scheme," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2022, pp. 94–126, Nov. 2021.

[19] P. Duong-Ngoc, T. N. Tan, and H. Lee, "Configurable butterfly unit architecture for NTT/INTT in homomorphic encryption," in *Proc. 18th Int. SoC Design Conf. (ISOCC)*, Oct. 2021, pp. 345–346.

[20] T. Fritzmann, U. Sharif, D. Müller-Gritschneder, C. Reinbrecht, U. Schlichtmann, and J. Sepúlveda, "Towards reliable and secure post-quantum co-processors based on RISC-V," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2019, pp. 1148–1153.

[21] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," 2019, *arXiv:1910.07557*.

[22] S. S. Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Verbauwhede, "Compact ring-LWE cryptoprocessor," in *Cryptographic Hardware and Embedded Systems–CHES*, Busan, South Korea: Springer, Sep. 2014, pp. 371–391.

[23] Z. Ye, R. C. C. Cheung, and K. Huang, "PipeNTT: A pipelined number theoretic transform architecture," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 10, pp. 4068–4072, Oct. 2022.

[24] R. Paludo and L. Sousa, "NTT architecture for a linux-ready RISC-V fully-homomorphic encryption accelerator," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 69, no. 7, pp. 2669–2682, Jul. 2022.

[25] Y. Su, B. Yang, C. Yang, Z. Yang, and Y. Liu, "A highly unified reconfigurable multicore architecture to speed up NTT/INTT for homomorphic polynomial multiplication," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 8, pp. 993–1006, Aug. 2022.

[26] P. Duong-Ngoc, S. Kwon, D. Yoo, and H. Lee, "Area-efficient number theoretic transform architecture for homomorphic encryption," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 70, no. 3, pp. 1270–1283, Mar. 2023.

[27] Y. Su, B. Yang, C. Yang, and L. Tian, "FPGA-based hardware accelerator for leveled ring-LWE fully homomorphic encryption," *IEEE Access*, vol. 8, pp. 168008–168025, 2020.

[28] I. Yoon, N. Cao, A. Amaravati, and A. Raychowdhury, "A 55 nm 50nJ/encode 13nJ/decode homomorphic encryption crypto-engine for IoT nodes to enable secure computation on encrypted data," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Apr. 2019, pp. 1–4.

[29] C. Gentry, S. Halevi, C. Peikert, and N. P. Smart, "Ring switching in BGV-style homomorphic encryption," in *Security and Cryptography for Networks*. Amalfi, Italy: Springer, Sep. 2012, pp. 19–37.

[30] Keccak Team. (2018). *SHA3 (Keccak)*. [Online]. Available: https://keccak.team/hardware.html

[31] A. C. Mert, E. Karabulut, E. Öztürk, E. Savas, M. Becchi, and A. Aysu, "A flexible and scalable NTT hardware : Applications from homomorphically encrypted deep learning to post-quantum cryptography," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 346–351.

[32] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Advances in Cryptology–CRYPTO*. Santa Barbara, CA, USA: Springer, Aug. 2012, pp. 868–886.

[33] (Apr. 2020). *Microsoft SEAL (Release 3.5)*. [Online]. Available: https://github.com/Microsoft/SEAL

[34] M. Dworkin, "SHA-3 standard: Permutation-based hash and extendable-output functions," U.S. Dept. Commerce, Nat. Inst. Standards Technol., Tech. Rep. FIPS PUB 202, 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[35] D. J. Bernstein, "ChaCha, a variant of Salsa20," in *Proc. Workshop Rec. SASC*, Lausanne, Switzerland, vol. 8, no. 1, 2008, pp. 3–5.

[36] S. Heron, "Advanced encryption standard (AES)," *Netw. Secur.*, vol. 2009, no. 12, pp. 8–12, Dec. 2009.

[37] R. Agrawal, L. Bu, and M. A. Kinsy, "A post-quantum secure discrete Gaussian noise sampler," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Dec. 2020, pp. 295–304.

[38] G. Xin et al., "VPQC: A domain-specific vector processor for post-quantum cryptography based on RISC-V architecture," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 8, pp. 2672–2684, Aug. 2020.

[39] M. Albrecht et al., "Homomorphic encryption standard," *Protecting Privacy Through Homomorphic Encryption*. Springer, 2021, pp. 31–62. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-77287-1_2

[40] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—A new hope," in *Proc. USENIX Secur. Symp.*, 2016, pp. 1–11.

[41] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Proc. Conf. Theory Appl. Cryptograph. Techn.* Cham, Switzerland: Springer, 1986, pp. 311–323.

[42] Z. Azad, G. Yang, R. Agrawal, D. Petrisko, M. Taylor, and A. Joshi, "RACE: RISC-V SoC for en/decryption acceleration on the edge for homomorphic computation," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, Aug. 2022, pp. 1–6.

[43] 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.

[44] S. R. Thummaluru, M. Ameen, and R. K. Chaudhary, "Four-port MIMO cognitive radio system for midband 5G applications," *IEEE Trans. Antennas Propag.*, vol. 67, no. 8, pp. 5634–5645, Aug. 2019.