IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 30 May 2023, accepted 10 July 2023, date of publication 13 July 2023, date of current version 20 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3295245

RESEARCH ARTICLE

VLSI Design and FPGA Implementation of an NTT Hardware Accelerator for Homomorphic SEAL-Embedded Library

STEFANO DI MATTEO, MATTEO LO GERFO, AND SERGIO SAPONARA
Department of Information Engineering, University of Pisa, 56126 Pisa, Italy

Corresponding author: Stefano Di Matteo (stefano.dimatteo@phd.unipi.it)

This work was supported in part by the European Union's Horizon 2020 Research and Innovation Program "European Processor Initiative" (EPI) under Grant 101036168 (EPI SGA2), and in part by the "Toward EXtreme Scale Technologies and Accelerators for euROhpc hw/Sw Supercomputing Applications for exascale" (TEXTAROSSA) under Grant 956831.

ABSTRACT Homomorphic Encryption (HE) allows performing specific algebraic computations on encrypted data without the need for decryption. For this reason, HE is emerging as a strong privacy-preserving solution in cloud computing environments since it allows to keep data secure even in the case the cloud server is not trusted. HE libraries such as Microsoft SEAL have been recently released; however, such libraries are not specifically designed for resource-constrained platforms and they are often expensive in terms of computational resources and memory consumption, which limits their usage in edge devices. This limitation is contained by the SEAL-Embedded library, the first C-based HE library specifically designed for embedded platforms. In this article, we propose a hardware accelerator specifically designed for the SEAL-Embedded library and its implementation of the CKKS scheme: the proposed hardware presents a configurable Number Theoretic Transform (NTT) unit for all the polynomial degrees available on the SEAL-Embedded, a memory architecture able to reduce the I/O latency and a dedicated module for the generation of roots of unity. A complete system that includes a 32-bit RISC-V (RI5CY) processor has been implemented on a Xilinx ZCU106 FPGA board to test the functionality of the hardware accelerator and to measure performance improvements. The results showed a speed-up of around x1000 with the hardware acceleration respect to the pure software implementation of the SEAL-Embedded library for the symmetric encryption function.

INDEX TERMS Hardware accelerator, homomorphic encryption, number theoretic transform, ring learning with errors, RISC-V, SEAL-Embedded, FPGA.

I. INTRODUCTION

Storing private data in encrypted form on server or cloud can be considered secure if no one has access to the secret key. But when our data need to be processed or updated it is necessary to decrypt them, opening a window of possibility for cyber attackers to take them over as long as they are in plain form. Homomorphic Encryption (HE) is born with the intention of keeping data encrypted while performing specific operations: given two messages m_1, m_2 and their encryption c_1, c_2, with HE is possible to obtain a new ciphertext $c_3 = c_1 \star c_2$ which is the encryption of $m_3 = m_1 \star m_2$, where $\star$ is a generic operation applied to data. HE is nowadays considered a solution that allows users to share data on cloud or on any non-trusted server denying any chance for the attackers or even cloud owners to learn anything about them since the private key is unknown to everyone except for the data owner. In the last decade, many open-source HE libraries have been developed, including lattice-based libraries such as Microsoft SEAL [1], [2] or PALISADE, whose security relies on the assumption that lattice problems are considered intractable for both quantum and classical computers. Their proven hardness and resistance even against quantum computer attacks make lattice-based cryptographic constructions

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

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

a valid alternative to modern cryptosystems; in fact, three out of four of the standardized algorithms for the NIST Post-Quantum Cryptography (PQC) competition are lattice-based: CRYSTALS-Kyber [3], CRYSTALS-Dilithium [4], and FALCON [5].

Even if HE guarantees data security, on the other hand, it requires high computational resources and memory consumption, which limits its use on resource-constrained devices; edge devices, for instance, could not be able to run HE libraries, or the execution time and the energy consumption of HE encryption functions could be very high in such devices where computational and energy resources could be constrained. This fact limits edge-cloud interoperability and suggests the need for designing ad hoc libraries and/or integrating dedicated hardware accelerators for the most computing-intensive operations of HE encryption functions. From the aforementioned Microsoft SEAL, an embedded-oriented spin-off library has been developed in 2021, the SEAL-Embedded (SE), which is the first lattice-based HE library specifically designed for embedded devices. Following the *Ring Learning With Errors* (RLWE) decision problem, which is recognized to be quantum resistant, this library implements a particular HE scheme called CKKS [6] that allows encryption over floating point numbers. The RLWE algorithm is built on arithmetic of polynomials where coefficients belong to a cyclotomic ring R modulo Q; given a secret key s, the ciphertext is composed of two polynomials $(a, -a \cdot s + m + e)$ where a is sampled from a discrete gaussian distribution over R, e is sampled from a uniform distribution and m is the plaintext. All the modular polynomial operations employed to evaluate the ciphertext require time and resources to be performed. In order to evaluate fast polynomial multiplication SE library employs the *Number Theoretic Transform* (NTT), which is a specialized Discrete Fourier Transform. Different acceleration strategies for HE and NTT can be found in the literature; an example is the Intel Homomorphic Encryption Acceleration Library (Intel HEXL), which is a C++ library that provides optimized implementations of polynomial arithmetic for Intel processors exploiting the Intel Advanced Vector Extensions 512 (Intel AVX512) [7]. This solution concerns Intel 64-bit processors and can be applied to HE schemes on the cloud side but is not suitable for the edge side where low-power microcontrollers with limited computational resources are usually employed. Other Hardware/Software solutions exploit Instruction Set Extensions dedicated to NTT such as [8], [9], and [10]. These lead to limited hardware resource consumption but also limited performance gain. Another approach is to design dedicated hardware accelerators connected to the microcontroller as peripherals through standard interfaces (e.g. AXI4 or others): this solution can lead to excellent gains in terms of execution time and energy efficiency as reported in [11], but hardware resource consumption can be high depending on the adopted design strategies. Some examples of this approach can be found in [12], [13], and [14]. In this case, most of the hardware resources are devoted to storing polynomial coefficients, performing arithmetic operations on them, and interfacing the hardware accelerator with the processor/microcontroller. In this paper, we propose a dedicated hardware accelerator for symmetric encryption of the SE library able to support all the polynomial degrees of the SE. The main contributions of this paper include but are not limited to:

- Design and implementation of the first hardware accelerator for the SE library: it shows a speed-up of around x1000 with respect to the SW implementation of the SE on a RISC-V 32-bit processor; the proposed hardware accelerator includes a standard AXI4 slave interface, so it can be connected to any CPU or microcontroller;
- A parametrizable (at synthesis level) architecture to support all or only a subset of the polynomial degrees of the SE library. The proposed memory scheduling for NTT-based operations allows parallelizing the data exchange between the processor-accelerator and the data processing and allows saving memory thanks to a dedicated module for the generation of the roots of unity for NTT.
- A complete characterization of the SE library for all the supported parameters on a 32-bit RISC-V processor, which includes a software-only version, and a hardware-accelerated version with and without the DMA support for data exchange.

The paper is organized as follows: Section II provides an overview of the Microsoft SE library, the RLWE encryption, and the arithmetic operations involved in it. Section III explains the architecture of the proposed hardware accelerator and the design optimizations introduced. Section IV presents the implementation results on FPGA and a comparison with existing solutions, and Section V draws the conclusions of this work.

II. HOMOMORPHIC ENCRYPTION SEAL-EMBEDDED LIBRARY

Due to high memory consumption and computational overhead, it is not easy or even possible to run a homomorphic encryption library on embedded devices since they are often resources constrained. Recently a new embedded-oriented library called *SEAL-Embedded* (SE) [15] has been developed; this library, which is a spin-off of the Microsoft *SEAL*, allows users to perform encryption over complex and real numbers following the CKKS scheme, and to implement different configurations for both asymmetric and symmetric encryption. It is not possible to perform decryption or any kind of homomorphic evaluation through the SE itself, therefore SE requires the usage of the SEAL library plus an additional *adapter* module, which is available together with the SE database. The *adapter* is necessary to convert the SE ciphertext into a usable format for the SEAL and also to generate private and public keys. A possible SE application scenario may be composed of the following parties, as reported in Fig. 1:

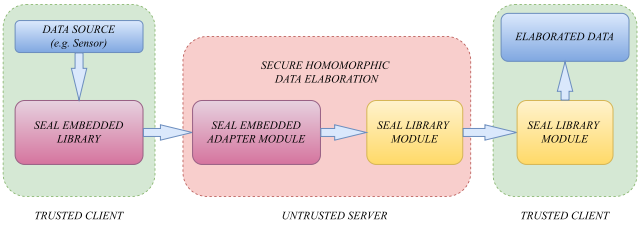

FIGURE 1. An application scenario for the SE homomorphic encryption library.

- An external trusted party that owns an instance of the adapter module.
- An embedded device that owns a SE instance.
- A server (it could be a non-trusted server) that owns both the *adapter* module and the SEAL installed on it.
- A client, which is able to decrypt everything the server will send using the private key previously generated by the external trusted party with the adapter. This party can also take the place of the former one.

Considering for example symmetric encryption, the trusted party generates through the adapter the private key and distributes it to the client and the embedded device, which is now able to encrypt and send data to the server. The server convert the received ciphertext and is capable of performing homomorphic operations over it through the SEAL. When the computation is complete, the resulting ciphertext is sent to the client. We remark that in such a scenario the server knows nothing about the secret key. Referring to Fig. 1, the trusted client could be an edge device with limited energy and computational budget, so the hardware acceleration for the most intensive HE operations can improve both energy consumption and performance in term of encryption latency/throughput.

A. RING LEARNING WITH ERRORS

SE encryption is based on the RLWE algorithm presented in [16]: given R_Q^n a ring of integer modulo Q with degree less than n, a distribution over R from which a key s is sampled, the ciphertext can be computed as a couple of integer polynomials $(a, b) \in R$ with degree less than n as stated in (1).

$$b = s \cdot a + e \ \ (mod\ Q) \tag{1}$$

Polynomial $a \in R_Q$ is uniformly random, while e is an error perturbation sampled independently from an error distribution over R. In SE polynomial degree n is chosen as a power of two and the selected ring is defined as $R = \mathbb{Z}[x]/(x^n + 1)$, where x^n+1 is the $2n$-th cyclotomic polynomial. All elements are polynomials represented as $n-$length vectors of their unsigned integer coefficients, whose values are in the set $\{0, \ldots, Q-1\}$.

B. ENCODING IN SEAL EMBEDDED

SE encrypts data following the CKKS scheme, allowing encryption over floating point values. Since encryption and decryption work on polynomial rings, it is necessary to convert the floating point message $z \in \mathbb{C}^{n/2}$ into an unsigned integer polynomial $m \in R$ without information loss. Following the work on [17], a polynomial in R is transformed into a complex vector $z \in \mathbb{C}^{n/2}$ when evaluated with the canonical embedding $\sigma : R \rightarrow \mathbb{H}$ and projected using the natural projection $\pi : \mathbb{H} \rightarrow \mathbb{C}^{n/2}$, where $\mathbb{H} = \{z_j : z_{-j} = \bar{z}_j, \forall j \in \mathbb{Z}_{2n}^*\}$ is a subring of $\mathbb{C}^n$. The encoding operation is performed evaluating the inverse of the canonical embedding and π projection and, in order to convert real numbers into integers, the resulting polynomial coefficients are multiplied by a scale factor Δ and rounded to the nearest integer. Scale factor Δ is taken high enough to prevent information loss from the round operation.

$$m = \lceil \Delta \cdot \sigma^{-1}(\pi^{-1}(z)) \rfloor_R \in R \tag{2}$$

In SE the π^{-1} projection takes as input a vector of single-precision numbers and outputs a vector of n single-precision values, made of the original values and their conjugates, re-ordered following a precomputed set of *index map*. It can be set in 4 different configurations:

- compute *on-the-fly*: values are computed onto the device when needed during encoding;
- compute *persistent*: same as the *on-the-fly*, but values are stored in RAM for the program lifetime;
- *load*: all indices are precomputed by the adapter, stored in FLASH, and loaded in RAM when message encoding is performed;
- *load persistent*: same as *load*, but values are stored in RAM for the program lifetime.

The σ^{-1} projection is evaluated as an Inverse Fast Fourier Transform, evaluating the polynomial at the powers of the $2n-$th root of unity of $x^n + 1$. It takes as input n real single-precision floating point values and outputs a n double-precision elements vector. IFFT can be set in 2 different configurations:

- compute *on-the-fly*: IFFT roots are computed during the execution of an encoding procedure
- *load*: all roots are calculated by the adapter, stored in FLASH, and loaded in RAM when needed

C. ENCRYPTION IN SEAL EMBEDDED

SE supports both symmetric and asymmetric encryption. All the operations must occur in R_Q but in SEAL the modulo is often quite large, resulting in huge memory consumption and execution time, especially in embedded devices. For this reason, the SE adopts the *Residue Number System* (RNS) basing on [18], which allows representing modulus Q as a product of smaller co-primes q_i following (3).

$$Q = \prod_{i=0}^{M-1} q_i = q_0 \times q_1 \times \cdots \times q_{M-1} \tag{3}$$

Any operation performed modulus Q is instead divided in $M-1$ operations for each modulus q_i. SE provides a set of precomputed moduli that can be represented on 32-bit, whose

TABLE 1. Number of co-primes assigned to each polynomial degree *n*.

Polynomial Degree (n)	Number of moduli
1024	1
2048	1
4096	3
8192	6
16384	13

number depends on the chosen polynomial degree. Table 1 reports the number of co-primes for each polynomial degree supported by the SE.

With regard to the RLWE algorithm, ciphertext in SE is evaluated as a couple of vectors of 32-bit unsigned integers, such that:

$$\begin{aligned} c_0 &= -a \cdot s + m + e \\ c_1 &= a \end{aligned} \tag{4}$$

Vector a is randomly sampled from a uniform distribution R_q while the vector e is sampled from a centered binomial distribution $\mathcal{B}$. Vector s is the secret key. In order to optimize performance, vectors are evaluated in NTT form before polynomial operations.

D. NUMBER THEORETIC TRANSFORM

Considering the overhead caused by polynomial multiplication, in SE those operations are optimized utilizing the *Number Theoretic Transform* (NTT), computed following the Harvey Butterfly operations, reported in Fig. 2. In a polynomial multiplication between two $(n-1)$ degree polynomials, evaluating the NTT of both allows multiplying their coefficient in a point-wise manner.

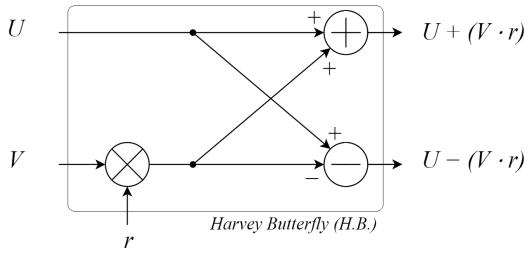

FIGURE 2. Harvey butterfly configuration.

Inputs in NTT are multiplied with twiddle factors (roots of unity) and combined with regard to the chosen butterfly. NTT is computed as a particular FFT algorithm and can be very efficient when the ring dimension is equal to $n = 2^i$ and when each polynomial coefficient is taken modulo q_i, where $q = 1 \ (mod \ 2n)$. Both these conditions are met in SE, and the negacyclic convolution on the inputs is performed using a special transform whose twiddle factors are powers of a primitive $2n - th$ root of unity, called the "negacyclic NTT" [15]. Twiddle factors in SE are evaluated as powers of a primitive $2n - th$ root of unity and ordered in bit-reversed form in RAM or FLASH with regard to the NTT chosen configuration, while such primitives are pre-calculated following (5).

$$1 \equiv \omega^{2n-1} \ (mod \ q) \tag{5}$$

SE gives 4 different options to compute the NTT and roots generation:

- *on-the-fly*: roots are computed only when requested by the NTT algorithm, once at a time;
- *one-shot*: all roots are computed before encryption begins and stored in RAM;
- *load*: roots are evaluated by the adapter, stored in FLASH and loaded in RAM during encryption;
- *load fast*: a high performance version of the NTT. Every root is computed by the adapter and coupled with an auxiliary integer in order to perform a fast integer modular multiplication. Both elements are stored in FLASH, therefore this configuration requires double memory space than the *load* version.

The NTT computation is divided into a number of $log_2(n)$ *stages* depending on the input length n. Each element in the input vector is divided into a number of *groups* and *pairs* that increase and decrease in every stage. An example is shown in Fig. 3, for a polynomial degree set to 8 which corresponds to 3 stages; the first stage starts with a single group and 4 pairs, the second stage continues with 2 groups and 2 pairs per group and finally, the third stage ends with 4 groups and 1 pair per group. Each stage of the NTT requires a number of twiddle factors equal to the number of groups composing such a stage; for instance, a stage with four groups requires four roots of unity to be computed. Despite bringing performance optimization, NTT is the encryption bottleneck due to the high amount of operations required.

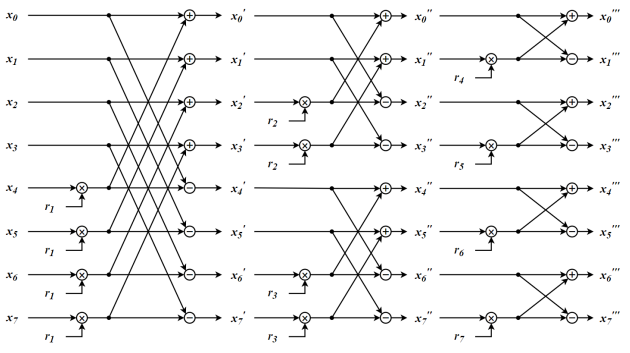

FIGURE 3. NTT execution for a hypothetical polynomial degree set to 8, counting 3 stages. To different groups are associated different roots, for a total of 7 roots of unity.

E. BARRETT REDUCTION

SE uses the *barret reduction* algorithm for the modular reduction after multiplications. This operation requires a modulus $q < 2^{30}$ and a constant parameter (the library refers to as

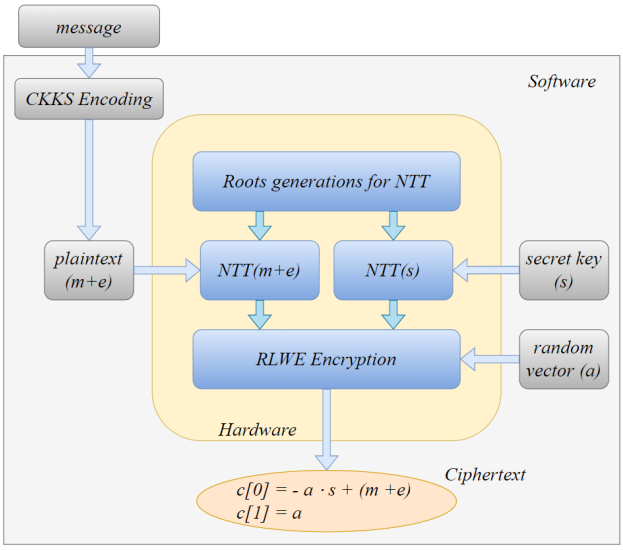

FIGURE 4. Hardware-Software partitioning of the symmetric encryption algorithm for the SE.

constant ratio), which is evaluated as shown in (6).

$$cr = \left\lfloor \frac{2^{64}}{q} \right\rfloor \tag{6}$$

The reduced result of a multiplication is calculated as shown in (7).

$$(x \cdot y)_q = x \cdot y - \left\lfloor \frac{x \cdot y \cdot cr}{2^{64}} \right\rfloor \cdot q \tag{7}$$

III. HARDWARE ACCELERATOR FOR SEAL-EMBEDDED SYMMETRIC ENCRYPTION

The proposed hardware accelerator aims to execute the symmetric encryption function reported in (4), with the message encoding deferred to the software. The hardware-software partitioning can be seen in Fig. 4.

Referring to (4) and Fig. 4, the hardware acquires the vectors $a, s, m + e$ from the software and proceeds with the encryption, evaluating *NTT(s)*, *NTT(m+e)* and performing all the polynomial operations to obtain the ciphertext. The architecture of the proposed hardware accelerator, that can be seen in Fig. 5, features the following main blocks:

- Two dual port RAMs (`DPRAM1` and `DPRAM2` in Fig. 5): they are used to store the polynomial coefficients up to the maximum supported polynomial degree (i.e. $n = 16384$) for SE. The size of these memories range from 8KB to 128KB depending on the maximum polynomial degree the hardware accelerator is able to support (from 1024 to 16384), which can be configured at synthesis time.
- A dual port RAM (`shared DPRAM` in Fig. 5): it is used as shared memory between the hardware accelerator and the main processor or DMA. Its size ranges from 4KB to 64KB (can be configured at synthesis time depending on the supported polynomial degree).
- An Arithmetic Logic Unit (`ALU Butterfly` in Fig. 5): it performs the butterfly operations stage by stage and the polynomial operations needed to obtain the ciphertext.
- The Roots Generator module (`Roots Generator` in Fig. 5): it computes and stores the roots of unity for each co-prime, according to what expressed in Section II-C.
- A finite state machine (`ALU NTT Fsm` in Fig. 5) that manages the RLWE algorithm and the memory accesses.

To be noted that the proposed hardware accelerator can be configured (at synthesis time) to support all the supported parameters of the SE library.

A. ALU BUTTERFLY MODULE

The `ALU butterfly`, depicted in Fig. 6, performs both Harvey Butterfly and polynomial operations. Within the ALU a `Barrett Reduction` block is instantiated, which is designed to perform both multiplication and modular reduction as explained in Section II-E. Due to the complexity of the combinational logic, the ALU includes four stages of pipeline. The ALU has a dual function:

- During the NTT phase, it takes as inputs the polynomial coefficients of the vectors s and $m + e$ stored into the dual port RAMs, plus the roots of unity from the `Roots RAM`. The ALU then performs modular multiplications, sums, and subtractions following the Harvey Butterfly algorithm sending the outputs to the corresponding dual port RAM. This procedure will be better explained in Section III-C.
- During the Encryption phase the ALU computes the RLWE algorithm. Inputs are now $NTT(s)$, $NTT(m + e)$ and a. The encryption result, which is a n-length vector, is taken from the $(U - V \cdot R)_q$ output and stored inside the `shared DPRAM`, which is accessible by the processor.

B. ROOT GENERATOR MODULE

As reported in Table 1, the SE requires a number of co-primes that depends on the selected polynomial degree, and for each modulus corresponds a different set of roots of unity for the NTT. For instance, in case the polynomial degree is 16384, 64KB of memory is required for each co-prime (to store the roots of unity), which means 832 KB of data to be stored to save the complete set of roots of unity for 13 co-primes. To overcome this issue a dedicated module for the on-the-fly generation of roots of unity has been implemented: at the beginning of the NTT computation, the Root Generator module starts the generation of the roots of unity and stores the complete set of roots (for the corresponding co-prime) inside the Roots RAM. After that, the NTT computation can take place. This module needs three parameters to work properly: modulus q, constant ratio cr, and the polynomial degree n. Each evaluated root is stored in the Roots RAM in a *bitreverse* manner from address 1 to address $n-1$. The *bitreverse* operation is evaluated on $log_2(n)$ *bits*, which means that $bitreverse(1)_{n=1024} = 512$ but $bitreverse(1)_{n=4096} = 2048$.

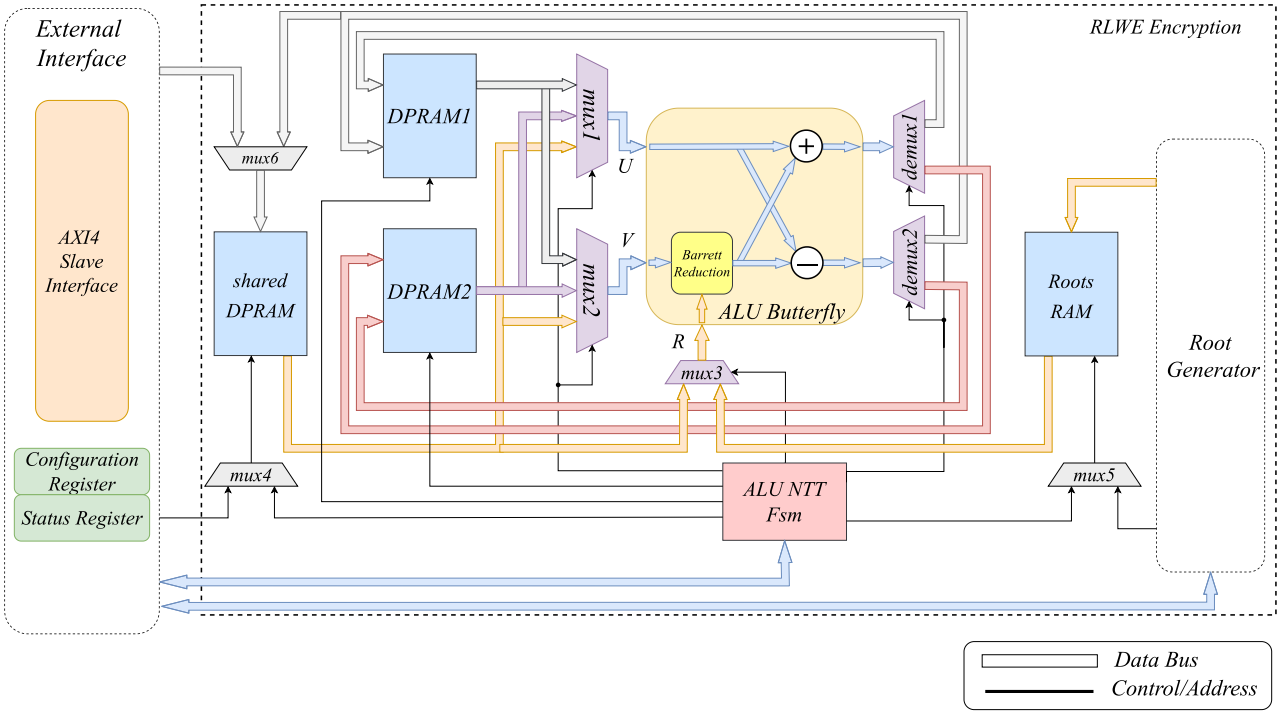

FIGURE 5. Architecture of the hardware accelerator for RLWE symmetric encryption for SE.

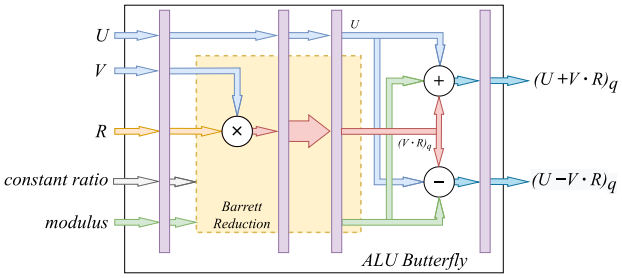

FIGURE 6. Architecture of the ALU butterfly.

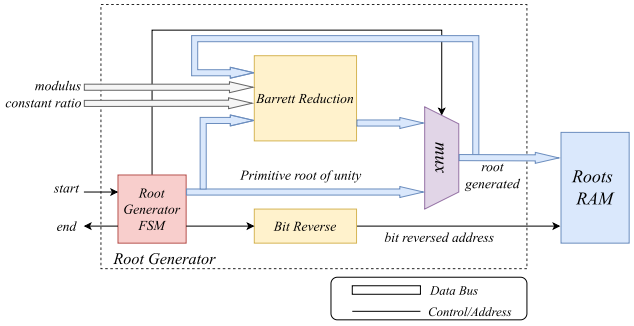

FIGURE 7. Roots generator module.

The first root stored in RAM at the address of $n/2$ is exactly the primitive root of unity already loaded as a parameter inside the hardware accelerator; the subsequent roots are evaluated exponentiating this first value $n-1$ times by itself, reducing the result modulo q every time. Referring to Fig. 7, all these arithmetic operations are computed by the Barrett Reduction module. Except for the first root ω_1, each root is ready to be stored after three clock cycles, because of the pipeline. A full set of roots is then accessible after almost $3n$ clock cycles since the start of the computation. For instance, let us consider a polynomial degree $n = 4096$, one of the moduli given by the SE, such as $q = 1053818881$, and the primitive root of unity associated, computed externally from our design and also given by the SE, in this very case equal to $\omega = 503422$.

The primitive roots of unity coincides with the first usable root for the NTT and it is stored in RAM without any evaluation at the address of 2048. The second root is evaluated as the multiplication of the primitive for itself reduced by q and stored after 3 clock cycles at the address of 1024, and so are the remaining roots.

$$\begin{aligned} |\omega^2|_q &= 517178644 && \text{stored at address 1024} \\ |\omega^3|_q &= 506942146 && \text{stored at address 3072} \\ &\vdots \\ |\omega^{4095}|_q &= 1053315459 && \text{stored at address 4095} \end{aligned}$$

C. MEMORY ORGANIZATION, OPERATIONS SCHEDULING, AND DATAFLOW

The polynomials a, s, and $(m+e)$ need to be stored inside the hardware accelerator memory to execute the RLWE encryption function, according to (4). Depending on the performance of the source that will write/read data to/from

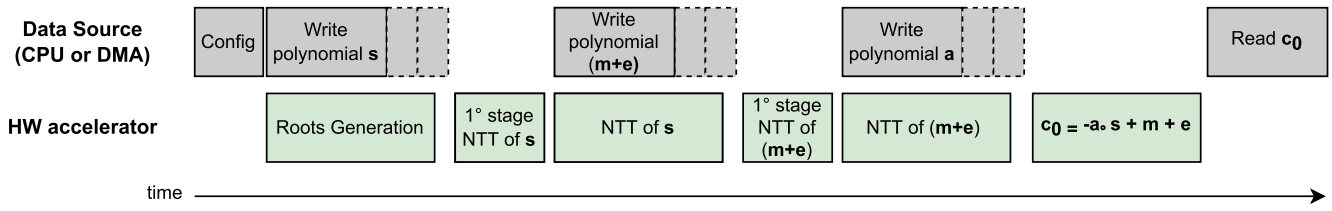

FIGURE 8. Timeline of the RLWE encryption function per co-prime.

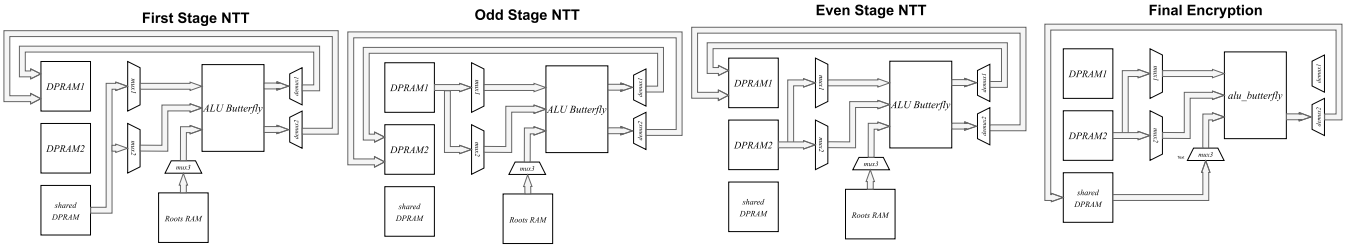

FIGURE 9. Dataflow and memory organization.

the accelerator (i.e., CPU or DMA), the data exchange can bring a significant overhead in the whole encryption function. For this reason, the `shared DPRAM` has been included in the architecture; this allows to parallelize the data transfer from the CPU to the accelerator and the NTT operation as reported in Fig. 8. At the beginning, a configuration stage of the hardware accelerator is needed to indicate the modulus, the polynomial degree, and the primitive root of unity. When configured, the accelerator starts the generation of the roots of unity according to the configuration parameters; simultaneously, the polynomial s can be written inside the `shared DPRAM` by the CPU or DMA. As soon as s is completely written (and the generation of the roots of unity is ended), the hardware accelerator executes the first stage of NTT of s. After the first NTT stage is ended, the `shared DPRAM` is no longer requested for the execution of the *NTT(s)*; in this way, it can be used to store the next polynomial $m + e$, parallelizing the execution of the NTT and the data store. The same process is repeated also for the last polynomial a. The computation of the complete encryption function is divided into four types of processes with different dataflow among the memories, as reported in Fig. 9:

1) During the first stage of the NTT, data are read from the `shared DPRAM`. After the `ALU Butterfly` elaboration, the results are written into the `DPRAM1`. When this first stage is over, the current NTT will no longer include any other operations on the `shared DPRAM`. After the first stage, *odd stages* and *even stages* are repeated until all $log_2(n)$ stages are over.
2) During the odd stages of the NTT, data are read from the `DPRAM1` and written into the `DPRAM2` after the elaboration.
3) During the even stages of the NTT, data are read from the `DPRAM2` and written into the `DPRAM1`.

TABLE 2. Arrival memory of the *NTT*(*s*) and *NTT*($m + e$).

Polynomial Degree	Stages	Arrival memory
1024	10	`dpram2`
2048	11	`dpram1`
4096	12	`dpram2`
8192	13	`dpram1`
16384	14	`dpram2`

4) During the *Final Encryption*, data are read from the `DPRAM1` or `DPRAM2` and `shared DPRAM`, processed by the `ALU Butterfly` and sent to the `shared DPRAM`, ready to be read from the CPU/DMA. During the last NTT stage, data will be stored inside a different DPRAM depending on the polynomial degree currently in use, as shown in Table 2.

An example can be done for a polynomial degree of 4096. In this case, the arrival memory for $NTT(s)$ and $NTT(m + e)$ is the `DPRAM2`, respectively in address $[0 : n - 1]$ and $[n : 2n - 1]$. The random vector a is stored in the `shared DPRAM`. After the *Final Encryption* operation, which involves the computation of $(-a)\cdot NTT(s)+NTT(m+e)$, the cyphertext is stored in the `shared DPRAM`. Thanks to the pipeline, the *Final Encryption* is executed in n clock cycles on average; actually, data need 4 clock cycles to elaborated by the `ALU Butterfly` and 2 clock cycles to be correctly stored in the arrival memory after being loaded from the corresponding memory, so 6 clock cycles are needed to fill the pipeline. A complete NTT requires then $log_2(n)\cdot n/2+6 \approx log_2(n)\cdot n/2$ clock cycles to be completed, while *Final Encryption* requires $n+6 \approx n$ clock cycles. We also must consider the roots generation latency of $L_{rg} = 3$ clock cycles (per single root of unity), and the I/O latency $L_{I/O}$ (per single polynomial coefficient to

be written/read in/from the `shared DPRAM`) that cannot be exactly predicted before the implementation of the system. This latency depends on the AXI4 CPU/DMA interface and on the interconnect logic. For this reason, we need to consider three different scenarios for the evaluation of the per-prime encryption latency (in clock cycles):

1) In case the latency of writing/reading a polynomial in/from the accelerator ($nL_{I/O}$) is greater than both the latency of the NTT computation and the latency of the roots generation, the per-prime encryption latency is:

$$\begin{aligned} Enc_{c.c.} &\approx 2 \cdot \underbrace{(n/2)}_{FirststageNTT} + \underbrace{4nL_{I/O}}_{I/O} + \underbrace{n}_{Enc.} \\ &\approx n \cdot \left(2 + 4L_{I/O}\right) \end{aligned} \tag{8}$$

2) In case the latency of writing/reading a polynomial in/from the accelerator ($nL_{I/O}$) is greater than the latency of the root generation but lower than the NTT computation, the per-prime encryption latency is:

$$\begin{aligned} Enc_{c.c.} &\approx 2 \cdot \underbrace{(log_2(n) \cdot n/2)}_{NTT} + \underbrace{n}_{Enc.} + \underbrace{2nL_{I/O}}_{I/O} \\ &\approx n \cdot \left(log_2(n) + 1 + 2L_{I/O}\right) \end{aligned} \tag{9}$$

3) In case the latency of writing/reading a polynomial in/from the accelerator ($nL_{I/O}$) is lower than both the latency of the NTT computation and the latency of the root generation, the per-prime encryption latency is:

$$\begin{aligned} Enc_{c.c.} &\approx 2 \cdot \underbrace{(log_2(n) \cdot n/2)}_{NTT} + \underbrace{n}_{Enc.} + \underbrace{nL_{I/O}}_{I/O} + \underbrace{nL_{rg}}_{rootgen} \\ &\approx n \cdot \left(log_2(n) + 1 + L_{I/O} + L_{rg}\right) \end{aligned} \tag{10}$$

The term *FirststageNTT* in (8) indicates the number of clock cycles required for the first stage of the NTT only; in (8), (9), and (10) the term *Enc.* refers to the computation of the *Final Encryption*. In addition, we considered the I/O latency symmetric (latency of write operations from the RAM memory to the accelerator equal to the latency of the read operations from the accelerator to the RAM memory), and we neglected the 6 clock cycles given by the `ALU Butterfly` pipeline and the clock cycles needed for the initial configuration of the accelerator. We remark that (8), (9), and (10) consider only the amount of clock cycles per prime encryption; for full encryption, the hardware accelerator must repeat the per-prime encryption for each co-prime, as reported in Table 1.

IV. FPGA IMPLEMENTATION RESULTS AND COMPARISON WITH THE STATE OF THE ART

The proposed hardware accelerator has been tested in a simulation environment that encompasses an AXI4 Master emulator for the generation of the AXI4 stimuli and transactions. The whole design (reported in Fig. 5) has been synthesized on the Zynq-Ultrascale+ MPSoC included in the Xilinx ZCU106 FPGA board, and Table 3 reports the synthesis results.

TABLE 3. Synthesis results on the target FPGA for different polynomial degrees.

Poly	Freq.	LUTs	REGs	BRAMs	URAMs	DSPs
1024	185 MHz	3168	1440	19.5	0	42
2048	180 MHz	3216	1460	16.5	2	42
4096	180 MHz	3320	1480	29.5	1	42
8192	180 MHz	3480	1500	51	0	42
16384	180 MHz	3742	1524	87	0	42

A. DEMOBOARD AND BENCHMARK CAMPAIGN

To test the functionality of the proposed hardware accelerator on FPGA and to measure the performance improvements on the SE thanks to the hardware acceleration a demoboard has been implemented, which encompasses the following components:

- The 32-bit RISC-V RI5CY [19] (also named CV32E40P) processor, which is a 32-bit 4-stage in-order processor.
- 1GB of DDR4 memory.
- A Central Direct Memory Access (CDMA) by Xilinx.
- An AXI4 interconnect and standard peripherals (JTAG and UART). JTAG peripheral is used to program the memory at the start-up of the system and the UART allows to communicate with the host PC.

Fig. 10 reports the architecture of the RISC-V-based system that includes the proposed hardware accelerator for SE. The entire system runs at @100MHz while the hardware accelerator runs at @150MHz (its maximum frequency is 180 MHz, as reported in Table 3) and is configured to support all the parameters of the SE library. Table 4 reports the performance results of the symmetric encryption function (per-prime) of the SE library on the proposed system, including the SW-only results (execution of the SE on the RI5CY processor) and SW plus the hardware acceleration with and without the support of the Xilinx DMA. The performance gain of the hardware accelerated version (without the DMA support) with respect to the SW-only solution ranges from 21.5x (Poly Degree equal to 1024) to 28.2x (Poly Degree equal to 16384). The speed gain of the hardware accelerated version with the DMA support for data exchange with respect to the SW-only version ranges from 1202.5x (Poly Degree equal to 1024) to 1306.4x (Poly Degree equal to 16384). Similar results have been obtained for the full encryption function, as reported in Table 5.

B. COMPARISON WITH THE STATE OF THE ART

A direct comparison with the State-of-the-Art is not simple since few works in the literature have concentrated on speeding up edge-side operations for HE. The work in [24] utilizes an FPGA-based accelerator for the BGV HE scheme. However, their BGV accelerator only supports small scheme

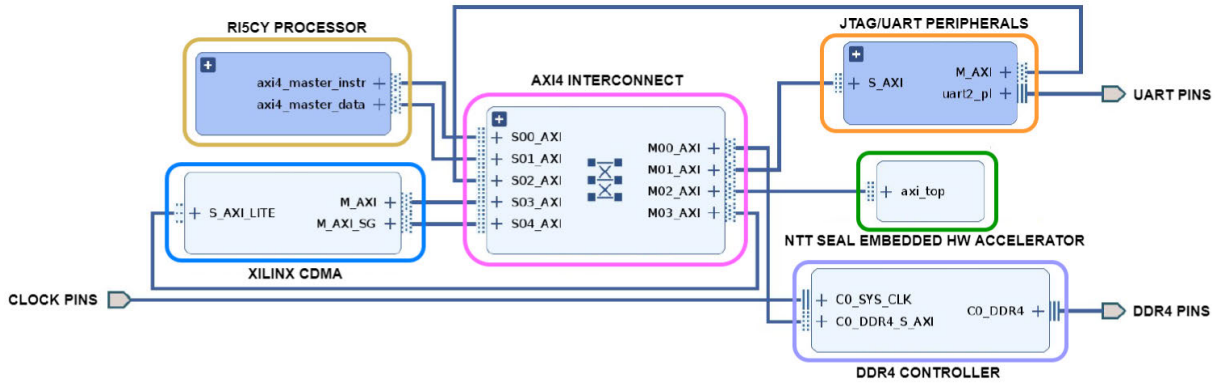

FIGURE 10. Simplified block Design of the proposed RISC-V-based system on Xilinx Vivado 2020.2. Reset synchonizers, clock generator and push buttons are not included for readability reasons.

TABLE 4. Benchmark of the SE (per-prime) symmetric encryption function on the RISC-V-based system. The "SW only" column refers to the execution time on the RI5CY processor, the "HW acc." and "HW acc. + DMA" columns refer to the execution time using the proposed hardware accelerator without and with Xilinx DMA support, respectively.

Poly Degree	per-prime encryption [ms]		
	SW only	HW acc.	HW acc + DMA
1024	168.35	7.84	0.14
2048	364.53	15.68	0.29
4096	781.45	31.24	0.62
8192	1664.47	62.47	1.29
16384	3527.45	124.93	2.70

TABLE 5. Benchmark of the SE (full encryption) symmetric encryption function on the RISC-V-based system. The "SW only" column refers to the execution time on the RI5CY processor, the "HW acc." and "HW acc. + DMA" columns refer to the execution time using the proposed hardware accelerator without and with Xilinx DMA support, respectively.

Poly Degree	full encryption [ms]		
	SW only	HW acc.	HW acc + DMA
1024	169.2	7.84	0.14
2048	366.72	15.68	0.29
4096	2352.74	93.72	1.86
8192	10032.13	374.83	7.79
16384	45856.85	1624.12	35.19

parameters (N = 128, modulus of 27-bit) that are not practical for HE computation. Although the authors claim that their accelerator can be extended to larger polynomial degrees to support higher security levels, support for larger parameters is not yet available. The authors of [25] present a RISC-V-based SoC (named RISE) to accelerate the asymmetric encryption of the SEAL Embedded library. The work in [25] which is very recent, shares some design choices with our work, but the authors did not report any implementation results on FPGA technology and hence a direct comparison is not possible. To the best of our knowledge, no other works targeting the acceleration of HE for constrained devices can be found in the literature; for this reason, we evaluated generic hardware accelerators for RLWE, and to make a fair comparison we compared the performance related to the execution of the NTT. The work in [20] presents a reconfigurable NTT architecture (named MCNA) for polynomial multiplication which employs different processing elements. The reported results consider a fixed number of processing elements (eight) and polynomial degrees equal to 1024 and 4096. As reported in Table 6, the latency of this architecture is very low but the resource consumption is higher than our work (about 420% more LUTs and 580% more FFs for a polynomial degree of 4096). For this reason, the architecture in [20] seems mainly oriented to the cloud side rather than the edge side. The authors in [21] propose a custom coprocessor for the Fan-Vercauteren (FV) HE scheme, which can accelerate homomorphic computations in cloud installations. They followed a hardware-software codesign approach and implemented their architecture in a Xilinx ZCU102 FPGA Board. The results reported in Table 6 refer to the NTT applied to 180-bit coefficients (they employ RNS to transform each 180-bit coefficient into six 32-bit coefficients). So, to make a fair comparison, the execution time for the NTT of the work in [21] must be divided by 6. The execution time they achieved is better than our work (11x in speed) but the resource consumption is enormous (about 1912% more LUTs and 1731% more FFs for a polynomial degree of 4096). It should be noted that the work in [21] presents a hardware accelerator not only for NTT but also for the computation of expensive operations of the (FV) HE scheme. For this reason, the resource consumption is so high. The work in [22] describes the implementation on FPGA of a hardware accelerator for encryption/decryption of the Brakerski/Fan-Vercauteren (BFV) homomorphic encryption scheme with high-performance polynomial multipliers. The architecture is targeted for cloud applications. They compute the NTT in 1.7 μs only (for the polynomial degree of 4096 and coefficients of 32-bit), but also in this case the resource consumption is very high with respect to our solution. The work in [23] provides an exhaustive study of design methods for implementing NTT. The authors propose software, High-Level Synthesis (HLS), and manual hardware

TABLE 6. Performance comparison of NTT.

Work	Platform	n	q	LUTs	REGs	DSPs	BRAMs	URAMs	Freq. [MHz]	Latency [us]
Our	Zynq Ultrascale+	1024	32-bit	3168	1440	42	19.5	0	185	27.71
Our	Zynq Ultrascale+	4096	32-bit	3320	1480	42	29.5	1	180	136.58
[20]	Virtex 7	1024	32-bit	10272	6704	80	87	0	250	2.60
[20]	Virtex 7	4096	32-bit	14004	8662	80	87	0	250	12.30
[21]	Zynq Ultrascale+	4096	180-bit	64000	–	200	193	0	250	73
[22]	Virtex 7	4096	32-bit	80000	–	952	325.5	0	200	1.7
[23]	Virtex 7	1024	29-bit	966	–	7	21.5	0	–	–

implementations of NTT. A direct comparison can be made with the NTT hardware implementation with 29-bit coefficients and a polynomial degree of 1024. They report a fairly modest resource consumption (966 LUTs, 7 DSPs, and 21.5 BRAMs), but they do not report the maximum synthesis frequency.

V. CONCLUSION

This work presented the implementation of a hardware accelerator for the symmetric encryption function of the SEAL-Embedded library. The proposed design can be configured to support polynomial degrees from 1024 to 16384 and presents a memory architecture able to reduce the I/O latency and a dedicated module for the generation of roots of unity. The proposed hardware provides an x1000 encryption speed-up compared to the sole software performance on a RISC-V (RI5CY) processor, also the possibility to generate at run-time the roots of unity during the encryption can save up to 832 KB of FLASH memory.

REFERENCES

[1] K. Laine. (2017). Simple encrypted arithmetic library 2.3.1. Microsoft Research. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf

[2] *Microsoft SEAL (Release 4.0)*, Microsoft Res., Redmond, WA, USA, Mar. 2022. [Online]. Available: https://github.com/Microsoft/SEAL

[3] R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle. (2020). *Crystals-Kyber Algorithm Specifications and Supporting Documentation (Version 3.0)*. [Online]. Available: https://pq-crystals.org/kyber/data/kyber-specification-round3.pdf

[4] S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehle. (2020). *Crystals-Dilithium Algorithm Specifications and Supporting Documentation. NIST Post-Quantum Cryptography Standardization Round 3*. [Online]. Available: https://pq-crystals.org/dilithium/data/dilithium-submission-nist-round3.zip

[5] P.-A. Fouque. (2020). *Falcon: Fast-Fourier Lattice-Based Compact Signatures Over NTRU*. [Online]. Available: https://falcon-sign.info/falcon.pdf

[6] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology—ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham, Switzerland: Springer, 2017, pp. 409–437.

[7] F. Boemer, S. Kim, G. Seifu, F. D. M. de Souza, and V. Gopal, "Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52," in *Proc. 9th Workshop Encrypted Comput. Appl. Homomorphic Cryptogr.* New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 57–62, doi: 10.1145/3474366.3486926.

[8] 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.

[9] E. Karabulut and A. Aysu, "RANTT: A RISC-V architecture extension for the number theoretic transform," in *Proc. 30th Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2020, pp. 26–32.

[10] 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.

[11] L. Baldanzi, L. Crocetti, S. Di Matteo, L. Fanucci, S. Saponara, and P. Hameau, "Crypto accelerators for power-efficient and real-time on-chip implementation of secure algorithms," in *Proc. 26th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Nov. 2019, pp. 775–778.

[12] J. Mu, Y. Ren, W. Wang, Y. Hu, S. Chen, C.-H. Chang, J. Fan, J. Ye, Y. Cao, H. Li, and X. Li, "Scalable and conflict-free NTT hardware accelerator design: Methodology, proof, and implementation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 42, no. 5, pp. 1504–1517, May 2023.

[13] K. Derya, A. C. Mert, E. Öztürk, and E. Savaş, "CoHA-NTT: A configurable hardware accelerator for NTT-based polynomial multiplication," *Microprocessors Microsyst.*, vol. 89, Mar. 2022, Art. no. 104451. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933122000254

[14] M. Bisheh-Niasar, R. Azarderakhsh, and M. Mozaffari-Kermani, "High-speed NTT-based polynomial multiplication accelerator for crystals-kyber post-quantum cryptography," Dept. Comput. Elect. Eng. Comput. Sci., Florida Atlantic Univ., Boca Raton, FL, USA, Tech. Rep. 2021/563, 2021. [Online]. Available: https://eprint.iacr.org/2021/563

[15] 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. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8991

[16] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, pp. 1–35, Nov. 2013, doi: 10.1145/2535925.

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

[18] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," in *Proc. Int. Conf. Sel. Areas Cryptogr.* Seoul, South Korea: Seoul National Univ., 2019, pp. 347–368.

[19] *RY5CY: User Manual*, Univ. Bologna, ETH Zürich, Zürich, Switzerland, Apr. 2019. [Online]. Available: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf

[20] Y. Su, B.-L. Yang, C. Yang, Z.-P. Yang, and Y.-W. 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.

[21] 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. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 387–398.

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

[23] A. C. Mert, E. Karabulut, E. Öztürk, E. Savas, and A. Aysu, "An extensive study of flexible design methods for the number theoretic transform," *IEEE Trans. Comput.*, vol. 71, no. 11, pp. 2829–2843, Nov. 2022.

[24] 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.
[25] Z. Azad, G. Yang, R. Agrawal, D. Petrisko, M. Taylor, and A. Joshi, "RISE: RISC-V SoC for en/decryption acceleration on the edge for homomorphic encryption," 2023, *arXiv:2302.07104*.

STEFANO DI MATTEO received the Ph.D. degree (cum laude) in information engineering from the University of Pisa, in 2023. He is currently a Postdoctoral Researcher with the University of Pisa. His research interests include digital and VLSI design, and embedded systems for cybersecurity and cryptography in different application fields (automotive, HPC, and the IoT).

MATTEO LO GERFO received the M.Sc. degree in electronic engineering from the University of Pisa, in 2022. His thesis focused on the design of a hardware accelerator for homomorphic encryption. He is currently a Digital Designer with Global High-Tech Company. His research interests include digital design, RISC-V processors architecture, and cryptography.

SERGIO SAPONARA received the master's degree (cum laude) and the Ph.D. degree from the University of Pisa. In 2012, he was a Marie Curie Research Fellow of imec. He is currently a Full Professor of electronics with the University of Pisa. He is the Director of I-CAS Laboratory, Crosslab Industrial IoT, Summer School Enabling Technologies for the IoT. He has coauthored more than 300 scientific publications and 18 patents. He is the Leader of many funded projects by EU and others companies like Intel, Magneti Marelli, Ericsson, and PPC. He is an IEEE Distinguished Lecturer and the Co-Founder of the Special Interest Group on IoT for both IEEE CAS and SP Societies. He is an Associate Editor of several IEEE and Springer Journals.

Open Access funding provided by 'Università di Pisa' within the CRUI CARE Agreement