Client-Server Framework for FPGA Acceleration of Fan-Vercauteren-based Homomorphic Encryption

Simon Bothe*, Hassan Nassar*†, Lars Bauer, Jörg Henkel*†

* *Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems (CES), Germany*
† {hassan.nassar, henkel}@kit.edu

***Abstract*—Cloud computing is becoming increasingly popular and widely used in daily services. However, traditional computation on the cloud involves decryption and processing plaintext on the server side, raising privacy and security concerns. Homomorphic encryption enables computations on encrypted data without decryption, but is significantly slower. Accelerating these computations through hardware and arithmetic optimizations, like FPGA usage, is essential. Previous works focused on implementing building blocks for constructing a homomorphic encryption system, but do not provide a full framework to perform full operations. In this work, we go in a different direction: We design a client-server framework with a communication channel and an FPGA-based acceleration for the Fan-Vercauteren algorithm, as it has several open-source implementations and can be easily deployed. Using the framework we evaluate the end-to-end performance of the algorithm. The client employs software for encryption, decryption, and key generation, while the server utilizes FPGA hardware to accelerate the computation. We evaluate the timing needed and compare it to executing a full software version of the system.**

***Index Terms*—Homomorphic Encryption, FPGA, Security**

I. Introduction

Cloud computing has become more popular in recent years. The benefits of using cloud computing over traditional IT infrastructure are obvious. Organizations and businesses no longer have to own and maintain their own IT infrastructure and can instead use cloud computing services on demand as long as they have Internet access.

But cloud computing also has its flaws. Especially privacy and security concerns are raised when talking about cloud computing [1]. To understand this point, it is useful to look at how computation on the cloud is performed in Figure 1. This immediately shows us two things: The cloud provider has access to public and private keys to perform encryption and decryption operations. Moreover, at some point the data we send, as well as the results of our computations, are present on the cloud system in a decrypted state. The data might be encrypted when transported between the two parties, but the user has to trust the cloud provider without any possibilities to protect the data themselves as soon as they arrive at the cloud computing system. This is a problem, as malicious cloud providers could access the data.

This work was partially funded by the "Helmholtz Pilot Program for Core Informatics (kikit)" at Karlsruhe Institute of Technology, and the German Federal Ministry of Education and Research (BMBF) through grant 01IS23066 as part of the Software Campus Project "HE-Trust".

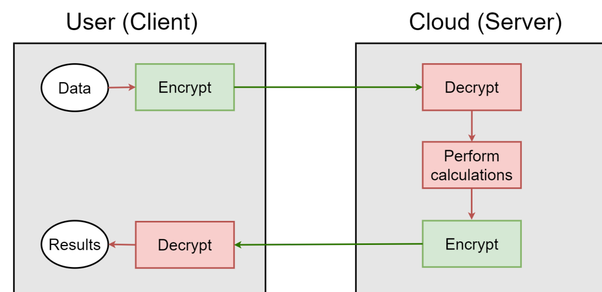

Fig. 1. Traditional cloud computing infrastructure for performing calculations. The operations are either red (encrypted), or green (decrypted) depending on the encryption status of the result. The same color coding applies to the data paths depending on the encryption status of the transferred data.

Homomorphic Encryption (HE) is one solution for this dilemma. Unlike traditional encryption methods, which require data to be decrypted before performing any operations, homomorphic encryption enables the execution of computations directly on data that remains encrypted [2]. This ensures that sensitive information remains protected throughout the entire process. The use cases for such a technology are diverse [1], including but not limited to: (i) **Artificial Intelligence over medical data**, (ii) **Action recognition and smart meters**, (iii) **Voting schemes**, and (iv) **Encrypted search engines**.

However, the computational overhead for performing homomorphic operations compared to performing the same operations on decrypted data is enormous [3]–[5]. In order to use homomorphic encryption under feasible conditions, it is necessary to speed up those computations. Aside from algorithmic optimizations in recent years, the most promising results come from the usage of hardware accelerators. They come mainly in two variants: Using a GPU and its ability to perform Single Instruction Multiple Data (SIMD) or using Field Programmable Gate Array (FPGA) is needed [3], [4].

Most of the previous work dealing with FPGA accelerators for homomorphic encryption applications like [6]–[8] usually only focus on the server side. But building a full homomorphic encryption framework also requires a client to be able to generate the needed keys, as well as to encrypt and decrypt the data. Without it, the implemented hardware accelerator cannot be used practically, and the benchmark times for the whole

framework remain theoretical. Therefore, in this work, we focus on building a client-server framework to enable acceleration of algorithms running fully in HE. We release our client-server framework as open source[1].

II. Homomorphic Encryption

Homomorphic encryption describes a type of encryption that enables computation on encrypted data without decrypting it. This is achieved by following a few steps [9], [10]. We assume that our plaintext is an element of an algebraic group. The plaintext gets encrypted into a ciphertext by mapping it from one algebraic group into another using a key. Decryption is the same transformation, but reversed.

It is possible to divide homomorphic encryption schemes in three categories : (i) **Partially homomorphic encryption** (PHE): Operates on groups and supports just one operation type. Schemes of this category can apply a single operation for an arbitrary amount of times without losing the ability to decrypt the data. (ii) **Somewhat homomorphic encryption** (SHE): Operates on rings and supports multiple operation types. Schemes of this category can apply multiple operations, but only for limited amounts of time. If the limits are exceeded, the ability to decrypt the data is lost. (iii) **Fully homomorphic encryption** (FHE): Operates on rings and supports multiple operation types with mitigation strategies for noise growth. Schemes of this category can apply multiple operations in any order for an arbitrary amount of times.

There are several algorithms for HE. We chose to use the Fan-Vercauteren (FV) algorithm as its hardware implementation is already available [11] open source and software like Microsoft SEAL [12] can be used to encrypt, decrypt and evaluate data without exact knowledge of the characteristics of the FV scheme. FV belongs to the Somewhat Homomorphic Encryption group. With each multiplication operation, the data become more noisy. Therefore, after a certain multiplicative depth, the data have to be decrypted to eliminate the noise before processing can continue. It can be fully homomorphic (infinite multiplication depth) by employing a bootstrapping mechanism [2]. The security of the FV scheme [2] and its key generation is rooted in the decision form of the Ring Learning with Errors (RLWE) problem.. This is considered a practical and efficient way to achieve security even in the age of post-quantum cryptography [13]. FV (as described in [2]) has eight main building blocks as follows.

(I) Secret key generation: Sample $s \leftarrow R_2$ (not from χ as in the original RLWE problem) and set the secret key $sk = s$.

(II) Public key generation: Sample $a \leftarrow R_q$ and $e \leftarrow R_q$. The public key calculation is similar to the one in the RLWE problem, but we swap the order of the parameters and add a sign change before applying the operation $modulo$. The public key pk is therefore given as the pair of the form

$$pk = ([-(a \cdot s + e)]_q, a) \tag{1}$$

(III) Encryption: Let R_t be the plaintext space for $t > 1$. Let m be the message that should be encrypted with $m \in R_t$. Sample $e_1 \leftarrow \chi$, $e_2 \leftarrow \chi$, sample $u \leftarrow R_2$ and let $\triangle = \lfloor q/t \rfloor$. Denote $p_0 = pk[0]$ as the first element in pk and $p_1 = pk[1]$ as the second element. The encrypted ciphertext ct can now be calculated as the pair of

$$ct = ([p_0 \cdot u + e_1 + \triangle \cdot m]_q, [p_1 \cdot u + e_2]_q) \tag{2}$$

(IV) Decryption: Let $s = sk$, $c_0 = ct[0]$ be the first element in ct and let $c_1 = ct[1]$ be the second element in ct. The decrypted plaintext d can be calculated by

$$d = [\lfloor \frac{t \cdot [c_0 + c_1 \cdot s]_q}{q} \rceil]_t \tag{3}$$

(V) Addition: Represents a coefficient-wise addition on the plaintexts. Given two ciphertexts ct_1 and ct_2 the sum ct_3 can simply be calculated by a pairwise addition

$$ct_3 = ([ct_1[0] + ct_2[0]]_q, [ct_1[1] + ct_2[1]]_q) \tag{4}$$

(VI) Multiplication: Represents a coefficient-wise multiplication on the plaintexts. Given two ciphertexts ct_1 and ct_2 the product ct_4 is given as a triple $(ct_4[0], ct_4[1], ct_4[2])$ with

$$\begin{aligned} ct_4[0] &= [\lfloor \frac{t \cdot (ct_1[0] \cdot ct_2[0])}{q} \rceil]_q \\ ct_4[1] &= [\lfloor \frac{t \cdot (ct_1[0] \cdot ct_2[1] + ct_1[1] \cdot ct_2[0])}{q} \rceil]_q \\ ct_4[2] &= [\lfloor \frac{t \cdot (ct_1[1] \cdot ct_2[1])}{q} \rceil]_q \end{aligned} \tag{5}$$

Since the resulting product is a triple of polynomials instead of a pair, we are unable to perform other operations as defined above on the result. This problem gets solved by a step called relinearisation which transforms a triple of polynomials into a pair without changing the encrypted plaintext. The relinearisation step requires a relinearisation key that can be generated after the secret key is known. Note that relinearisation is just a special kind of key switching, meaning it transforms a ciphertext that is encrypted with one secret key into a ciphertext encrypted with another secret key.

(VII) Relinearisation key generation: Given an integer base T. Sample $a_i \leftarrow R_q$, $e_i \leftarrow R_q$ and set $l = \lfloor log_T(q) \rfloor$. The relinearisation key is just a masked version of T

$$rlk = [([-(a_i \cdot s + e_i) + T^i \cdot s^2]_q, a_i) : i \in [0..l]] \tag{6}$$

(VIII) Relinearisation: Given a freshly calculated product $ct_4 = (ct_4[0], ct_4[1], ct_4[2])$ and the relinearisation key rlk, we can calculate the pair ciphertext $ct'_4 = (ct'_4[0], ct'_4[1])$

$$\begin{aligned} ct'_4[0] &= [ct_4[0] + \sum_{i=0}^{l} rlk[i][0] \cdot ct_4[2]^{(i)}]_q \\ ct'_4[1] &= [ct_4[1] + \sum_{i=0}^{l} rlk[i][1] \cdot ct_4[2]^{(i)}]_q \end{aligned} \tag{7}$$

The FV scheme uses some parameters that can be chosen freely. These parameters can influence the capabilities, performance and security of the scheme when used. In an effort to standardise and ensure security it is highly advised to follow the homomorphicencryption.org guidelines [14] when setting these parameters.

[1] https://github.com/DAL3X/somewhat-homomorphic-encryption-system

III. Architecture description

We aim to construct a complete homomorphic encryption framework. The encryption framework in this context means having an interface capable of homomorphic key generation, encryption, and decryption, and capable of performing homomorphic multiplication and addition, as well as relinearisation on the encrypted data. Since we want to have a real-world scenario, we split our framework into two parts. The client side is responsible for key generation, encryption, and decryption. The server side is responsible for homomorphic addition, multiplication, and relinearisation. The communication between those two has to be able to transmit over long distances while still making sure it is fast and reliable. A TCP/IP stack was chosen for this purpose, since IP allows fast long-range communication, while TCP ensures that no data is lost along the way. An overview of this architecture is given in Figure 2. However, there are a few things to note when implementing this architecture. The TCP protocol uses network congestion avoidance. This means that when packages are taking too long or they get lost, the number of packages sent per fixed time interval gets reduced. Since performing a computation is dependent on the data being transferred to the server, a slow or unstable Internet connection can lead to slower data transfers and therefore potentially slower calculations. We will look at the sending overhead and compare it with the time needed to perform the calculation in Section IV to see if the influence of this is negligible. Another thing to consider is the relinearisation. It should be performed on the server side because it is computationally expensive compared to other operations. However, performing the relinearisation step on the server side brings further challenges with it. The server needs access to the relinearisation key. Since all key switching keys are considered public, this does not influence the security of our encryption scheme, and they can be freely exchanged [2].

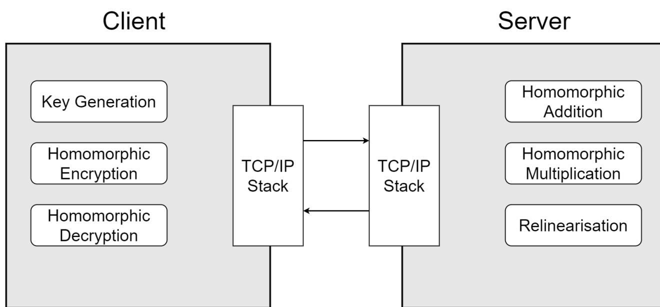

Fig. 2. The proposed Client-Server-Architecture with a TCP/IP stack for communication. The client generates keys, performs encryption and description. The homomorphic operations are done by the server.

A. Client Implementation

The client has to implement homomorphic encryption, homomorphic decryption and the corresponding key generation. The software client further has to implement a way of communicating with the server and packing the data to the same structure expected by the server. Communication and data packing/unpacking has to be implemented by hand. But the other functionalities are already accessible through open-source libraries like the OpenFHE repository [15]. Microsoft SEAL [12] is used as it is the nearest to the implemented hardware.

The client architecture shown in Figure 3 comprises three main components: (i) SEALInterface, (ii) Packer, and (iii) SocketClient. SEALInterface serves as the primary interface for interacting with the Microsoft SEAL library, facilitating the generation of encryption and relinearization keys, encryption, and decryption. The packer is responsible for packing and unpacking ciphertexts and relinearization keys to match the format that the hardware accelerator needs on the server side. Additionally, SocketClient communicates with the server over sockets, offering methods for transmitting and receiving ciphertexts and relinearization keys.

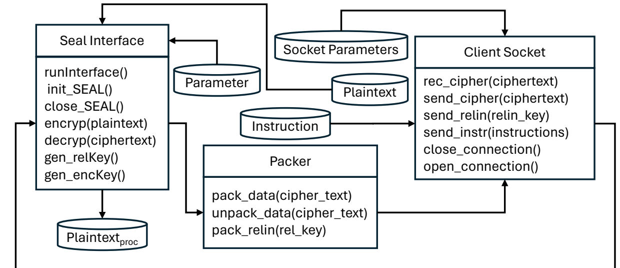

Fig. 3. Client architecture for homomorphic encryption. The diagram illustrates the interactions and dependencies between the SEALInterface, Packer, and SocketClient classes, highlighting their roles in facilitating secure computation and communication within the framework.

Algorithm 1 shows the steps that the client needs to perform. It starts by generating an encryption key (key_{en}) to encrypt plain text and the relinearization key (key_{rel}) for relineratization on the server. Subsequently, it sends the instructions and (key_{rel}) to the server for processing. The algorithm then enters a loop in which it repeatedly performs the following steps until the processing is complete. Within each iteration, using the encryption key, it encrypts the plaintext to produce a ciphertext. This ciphertext is then sent to the server for processing. After receiving the processed ciphertext ($ciphertext_u$) from the server, it decrypts it using the encryption key to obtain the processed plaintext ($Plaintext_{proc}$).

B. Server Implementation

For reproducibility, we aim to use open-source components for building the complete framework. The only accessible works, at the time of writing this paper, that describe the implementation of an FPGA accelerator used for accelerating homomorphic operations and share the source code are from [11], [16]. We use them as the starting point and we build the rest of our homomorphic encryption framework around it.

The hardware accelerator implements the FV scheme with a multiplicative depth of four. Since we are talking about a somewhat homomorphic encryption implementation, the parameters are set to allow four multiplications before the result fails to decrypt correctly. The ciphertexts consist of

Algorithm 1: Homomorphic en/decryption on client

Input: Plaintext
Output: Plaintext_{proc}
Generate(key_{en})
Generate(key_{rel})
Send(instructions)
Send(key_{rel})
while *!done* **do**
  Ciphertext $\leftarrow$ **encryption**(plaintext, key_{en})
  Ciphertext_{pack} $\leftarrow$ **pack**(Ciphertext)
  Send(ciphertext_{pack})
  Receive(ciphertext_{u})
  Ciphertext $\leftarrow$ **unpack**(ciphertext_{u})
  Plaintext_{proc} $\leftarrow$ **decrypt**(ciphertext, key_{en})
  Plaintext $\leftarrow$ Plaintext_{proc}

polynomials with **4096** coefficients. The selected modulus q is a **180-bit** number. When multiplication is performed, the polynomials are shifted from q to Q. The modulus Q should be at least a **372-bit** integer [2].

Algorithm 2 shows the full operation on the server side. It iterates over the ciphertext according to the instructions provided while ensuring that the multiplications remain below the multiplicative depth. It consists of a loop where it sequentially executes each instruction in the ciphertext. If the instruction involves multiplication, it performs a reliniarization operation on the ciphertext using the provided reliniarization key. The counter is incremented after each multiplication operation to track the number of iterations. Once all instructions have been processed or the counter reaches the threshold, the resulting processed ciphertext (**ciphertext**$_u$) is sent back to the client.

Algorithm 2: Homomorphic Processing on Server side

Input: Key_{rel}, Ciphertext, Instructions
Output: Ciphertext_{u}
Initialize counter: **ctr** $\leftarrow 0$
Receive key_{rel}, ciphertext, instructions
Initialize output: **ciphertext**$_u$ $\leftarrow$ ciphertext

while *instructions* $\neq$ *empty* ***and*** *ctr* $<$ *mul_depth* **do**
  `// Execute instruction`
  ciphertext$_u$ $\leftarrow$ **Instruction**(ciphertext_u)
  if *instruction is* ***multiply*** **then**
    `// Perform reliniearization`
    ciphertext$_u$ $\leftarrow$ **relin**(ciphertext_u,key_{rel})

    `// Increment counter`
    Increment counter: **ctr** $\leftarrow$ **ctr** $+ 1$

Send ciphertext_u

IV. Results

The experimental setup, shown in Figure 4, consists of the client (local computer) and the server (Xilinx Zynq UltraScale+ ZCU102 board). All measurements are timed multiple times on our client and server, with the average of those timings taken as the measurement. Operations on the client side and software library operations are performed by an Intel i7-2600 Quad-Core CPU, while operations on the server side run either on the FPGA, or get performed by the integrated ARM Cortex-A53 Quad-Core processor.

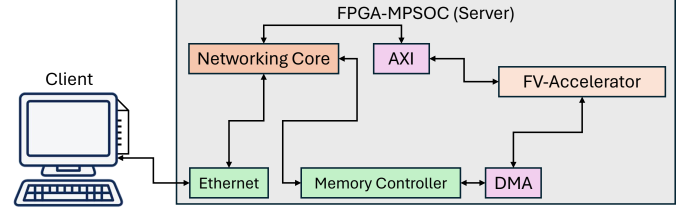

Fig. 4. The implemented system. The opensource FV-Accelerator is integrated with communication infrastructure. The client runs on a PC sends the ciphertext and receives processed ciphertext which it decrypts back to plaintext.

The data have to arrive completely and in the correct order to ensure correct results for the calculations on server side. Over long distances, this is only possible if we pair the IP with the TCP protocol. The client software can use sockets for an easy TCP/IP implementation, since the machine runs an operating system. But since our server does run bareback without an operating system, we are forced to use a slightly different approach. We first need to implement/enable an Ethernet interface using Processing System (PS), or implemented using Programmable Logic (PL). So we can either let a Gigabit Ethernet Controller (GEM) handle the Ethernet packages, or program the FPGA to do so. We choose the PS option with the GEM3 through Multiplexed Input/Output (MIO). Implementing the Ethernet interface in this way allows for up to one gigabit per second of data transfer. Both devices were connected to the same network with a ping around 0.2 ms between them to avoid turnaround times being a factor.

TABLE I
Overhead timings for transferring a ciphertext or relinearisation key and the packing/unpacking operation.

Ciphertext transfer	Packing	Unpacking
0.18 ms	2.54 ms	1.29 ms

We start with the timings on the client side and communication. This includes the transfer of the ciphertext, as well as the packing/unpacking procedure (transforming data to be compliant with server implementation). The measurements can be seen in Table I. The timings for data transfers have to be put in context. They can change drastically depending on the used network card, network speed/bandwidth, and how many packages are already in the network. Even in the same environment, the timings varied by more than 40ms while taking the measurements. It is also important to note that the data transfer overhead would also be present when implementing a software server without the use of a hardware accelerator. These stats are therefore not directly helpful when comparing a software and hardware implementation, but they do

give a clear indication for a potential communication bottleneck. To put this into perspective and gain insight into the differences, Table II provides an overview of the calculation times.

TABLE II
CALCULATION TIMINGS FOR ENCRYPTION, ENCODING, DECRYPTION, DECODING, MULTIPLICATION, RELINEARISATION, AND ADDITION.

Performed by	Encrypt and Encode	Decrypt and Decode	Multiplication and Relinearisation	Addition
Microsoft SEAL	3.26 ms	1.16 ms	14.74 ms	0.23 ms
OpenFHE	7.68 ms	0.59 ms	10.41 ms	0.05 ms
FPGA	-	-	4.46 ms	0.03 ms

It shows the calculation times for operations performed by the homomorphic encryption libraries Microsoft SEAL [12] and OpenFHE [15] in comparison to the hardware accelerator. Since neither encryption nor decryption is performed by the FPGA, the times are missing and the software timings only bear relevance for the client implementation. Note that decryption and encryption were never supposed to run on the FPGA in our architecture, because it is the exclusive responsibility of the client to avoid any secret keys or decrypted data being present on the server. We can see that FPGA is faster in homomorphic addition and homomorphic multiplication than either of the software implementations.

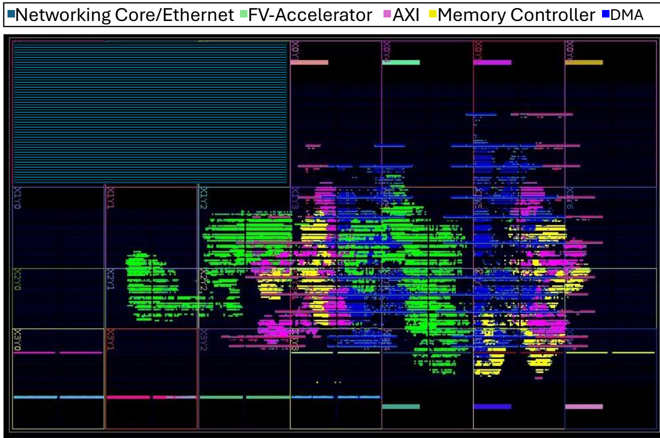

Fig. 5. The Floor plan on the FPGA. The ethernet and the networking core are implemented in the PS. The resource utilization is dominated by the open source FV-accelerator [3].

Although timing and accelerating execution are our main focus, we also report the utilization of resources to implement the server on FPGA. Figure 5 shows the floorplan of the implemented system. We are using both PS and PL. For communication with the client, we rely on the PS as it is already equiped with the needed infrastructure. On the PL, we implement the open source FV-accelerator [3] along with the AXI interface to the PS and the memory interface. Most of the area used on the PL are used by the open source accelerator.

We also look at the full detailed implementation results on the PL as shown in Table III. In general, we do not use much resources, we have a relatively high utilization of BRAm but

TABLE III
RESOURCE UTILIZATION FOR IMPLEMENTING OUR SERVERSIDE ON THE FPGA-MPSOC.

LUT	FF	BRAM	DSP
58534 (21%)	25549 (5%)	389 (43%)	208 (8%)

still less than 50%. Moreover, for, LUT utilization we are even lower than 25%. For FF and DSP the utilization is significantly less. Therefore, the full system can be used and even extended if needed, e.g., to implement bootstrapping to transform the system from SHE to FHE.

V. CONCLUSION

In this work, we design a client-server framework for Homomorphic Encryption. We implement open-source FPGA hardware accelerators on the server side along with the necessary communication. Our hardware accelerator outperformed their software counterparts (SEAL and OpenFHE) in multiplication and linearization by 2.3× and 3.3×, and addition by 1.7× and 7.7× highlighting the need for hardware optimization. The current implementation supports only Somewhat Homomorphic Encryption. In future work, we intend to implement a bootstrapping mechanism to support Fully Homomorphic Encryption as well.

REFERENCES

[1] D. Archer *et al.*, "Applications of Homomorphic Encryption", in *Crypto Standardization Workshop, Microsoft Research*, 2017.
[2] J. Fan *et al.*, "Somewhat Practical Fully Homomorphic Encryption", Cryptology ePrint Archive, Paper 2012/144, 2012.
[3] S. S. Roy *et al.*, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation", *IEEE TC*, 2018.
[4] W. Wang *et al.*, "Accelerating fully homomorphic encryption using GPU", in *2012 IEEE Conference on High Performance Extreme Computing*, 2012, pp. 1–5.
[5] H. Nassar *et al.*, "HBMorphic: FHE Acceleration via HBM-Enabled Recursive Karatsuba Multiplier on FPGA", in *IEEE FCCM*, 2024.
[6] Y. Yang *et al.*, "FPGA Acceleration of Rotation in Homomorphic Encryption Using Dynamic Data Layout", in *IEEE FPL*, 2023, pp. 174–181.
[7] Y. Yang *et al.*, "Poseidon: Practical Homomorphic Encryption Accelerator", in *IEEE HPCA*, 2023, pp. 870–881.
[8] B. Bulut *et al.*, "HW/SW Co-Design of TFHE Homomorphic OR Gate via NTT-based Polynomial Multiplication on a programmable SoC", in *IEEE Innovations in Intelligent Systems and Applications Conference*, 2022.
[9] X. Yi *et al.*, *Homomorphic Encryption*. Cham: Springer International Publishing, 2014, pp. 27–65.
[10] I. Chillotti *et al.*, "Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks", Zama AI, 2020. [Online]. Available: https://github.com/zama-ai/tfhe-rs
[11] S. Sinha Roy *et al.*, "FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data", in *IEEE HPCA*, 2019.
[12] "Microsoft SEAL (release 4.1)", Jan. 2023, microsoft Research.
[13] I. Blanco-Chacón, "Ring Learning with Errors: a Crossroads between Post-Quantum Cryptography, Machine Learning and Number Theory", *Irish Math. Soc. Bull*, vol. 86, pp. 17–46, 2020.
[14] M. Albrecht *et al.*, "Homomorphic Encryption Security Standard", HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., 2018.
[15] OpenFHE, "An open-source project that provides efficient extensible implementations of the leading post-quantum Fully Homomorphic Encryption (FHE) schemes", December 2022.
[16] F. Turan *et al.*, "HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA", *IEEE TC*, 2020.