Journal of Circuits, Systems, and Computers

FPGA Acceleration of Homomorphic Encryption Assisted Federated Learning

Mohamed Aboelenien[§,*] Mohamed Alsharkawy[§*], Simon Bothe[§*], Hassan Nassar[§], Lars Bauer[†], Jörg Henkel[§]

[§] *Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems (CES) Karlsruhe, Germany*
mohamed.ahmed3@kit.edu, mohamed.alsharkawy@kit.edu, simon.bothe@student.kit.edu, hassan.nassar@kit.edu, henkel@kit.edu

[†] *Ubitium GmbH, Karlsruhe, Germany*
lars.bauer42@gmx.de

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)
Published (Day Month Year)

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. Furthermore, we evaluate the timing needed and compare it with executing a full software version of the system. We further extend and adapt the framework to use additive homomorphic encryption in the federated learning use case, where the cloud server is responsible for aggregation of the clients' models. We evaluate our extended framework and record the aggregation time and resource utilization of the XI-based Paillier adder.

Keywords: Federated Learning, FPGA, Homomorphic Encryption

1. Introduction

Deep learning has achieved outstanding performance in various domains, such as vision, medical imaging, text, and recommendation systems. This impressive

*All three authors contributed equally

performance is attributed to the ability to learn complex representations from training on a large amount of data. Traditional training approaches are performed in a centralized manner that requires the collection of data from multiple sources into a single data center, raising concerns about data privacy, ownership, and security. This creates a gap between the potential of deep learning models and the constraints of real-world environments, such as healthcare, finance, and personalized services, where the data is sensitive and often distributed across multiple institutions or user devices and cannot be easily centralized for training.

Federated learning [1,2,3,4] bridges this gap by enabling collaborative model training across distributed data sources. Instead of sending sensitive data to a central cloud server, federated learning allows local devices or institutions to compute model updates and only share the updated models with a central server for aggregation. Nevertheless, the shared models can still reveal information about training data, which can be exploited by malicious cloud providers [5,6].

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 [7].

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[8]. This ensures that sensitive information remains protected throughout the entire process.

However, the computational overhead for performing homomorphic operations compared to performing the same operations on decrypted data is enormous [9,10,11,12]. 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 [13,14].

Most of the previous work dealing with FPGA accelerators for homomorphic encryption applications like [9,10,15,16,17] 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

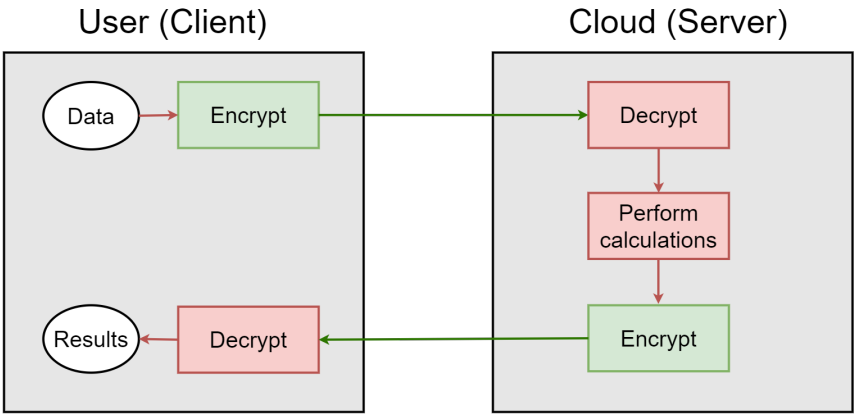

Fig. 1. Traditional cloud computing infrastructure for performing calculations. The operations are either red (encrypted), or green (decrypted) depending on the encryption status of the result. The same color coding applies to the data paths depending on the encryption status of the transferred data.

fully in HE. We released our client-server framework as open source [18]. Furthermore, we extend this framework to evaluate on the federated learning scenario, where the server performs additive homomorphic operations (i.e., using Paillier encryption) for model aggregation.

2. 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 [19,20]. 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.

Let $(P, \diamond, C, \circ, e, d)$ be our homomorphic encryption scheme, where P denotes the plaintext group with the group operation $\diamond$ and C the ciphertext group with the group operation $\circ$. e and d denote the encryption and decryption algorithm respectively. By this definition $e : P \to C$, $d : C \to P$, $\diamond : P \to P$ and $\circ : C \to C$ applies. Note that $\diamond$ is an arbitrary operation on P like addition or multiplication, while $\circ$ is provided by the selected homomorphic encryption scheme.
Given two plaintexts $a \in P$ and $b \in P$, the encryption scheme satisfies the following equation:

$$e(a) \circ e(b) = e(a \diamond b) \tag{1}$$

By doing this, it allows us to perform our modified operation $\circ$ on encrypted data. The decrypted result will be the same as performing our intended operation $\diamond$ on

the plaintext, but without any data knowledge during the calculation step.

$$d(e(a) \circ e(b)) = a \diamond b \tag{2}$$

Equations (1) and (2) are only satisfied, if the encryption function e is a homomorphism from $(P, \diamond)$ to $(C, \circ)$. Hence, the name homomorphic encryption. The above concept only allows support for a single operation. But two operations can be supported by applying the concept on rings instead of groups [19]. Unfortunately, with each computation on the encrypted data some noise is accumulated. Thus, over multiple operations the noise will accumulate and eventually the noise level becomes too high to correctly decrypt the data.

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.

2.1. *Fan-Vercauteren Algorithm*

There are several algorithms for HE. We chose to use the Fan-Vercauteren (FV) algorithm as its hardware implementation is already available [21] open source and software like Microsoft SEAL [22] 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 [8]. The security of the FV scheme [8] 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 [23]. FV (as described in [8]) 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

$$pk = ([-(a \cdot s + e)]_q, a) \tag{3}$$

(III) Encryption: Let R_t be the plaintext space for $t > 1$. Let m be the message that should be encrypted with $m \epsilon R_t$. 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

$$ct = ([p_0 \cdot u + e_1 + \triangle \cdot m]_q, [p_1 \cdot u + e_2]_q) \tag{4}$$

(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

$$d = [\lfloor \frac{t \cdot [c_0 + c_1 \cdot s]_q}{q} \rceil]_t \tag{5}$$

(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

$$ct_3 = ([ct_1[0] + ct_2[0]]_q, [ct_1[1] + ct_2[1]]_q) \tag{6}$$

(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

$$\begin{aligned} ct_4[0] &= [\lfloor \frac{t \cdot (ct_1[0] \cdot ct_2[0])}{q} \rceil]_q \\ ct_4[1] &= [\lfloor \frac{t \cdot (ct_1[0] \cdot ct_2[1] + ct_1[1] \cdot ct_2[0])}{q} \rceil]_q \\ ct_4[2] &= [\lfloor \frac{t \cdot (ct_1[1] \cdot ct_2[1])}{q} \rceil]_q \end{aligned} \tag{7}$$

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

$$rlk = [([-(a_i \cdot s + e_i) + T^i \cdot s^2]_q, a_i) : i \epsilon [0..l]] \tag{8}$$

(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 cipher-

text $ct'_4 = (ct'_4[0], ct'_4[1])$

$$\begin{aligned} ct'_4[0] &= [ct_4[0] + \sum_{i=0}^{l} rlk[i][0] \cdot ct_4[2]^{(i)}]_q \\ ct'_4[1] &= [ct_4[1] + \sum_{i=0}^{l} rlk[i][1] \cdot ct_4[2]^{(i)}]_q \end{aligned} \tag{9}$$

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 [24] when setting these parameters.

2.2. *Paillier Algorithm*

In addition to the FV-algorithm we chose to implement the Pailier algortithm as well to represent the partially homomorphic algorithms subset of homomorphic encryption. The Paillier cryptosystem supports additive homomorphism. Specifically, given two ciphertexts $C_1 = E(m_1)$ and $C_2 = E(m_2)$, their product modulo n^2 yields the encryption of the sum of the corresponding plaintexts:

$$E(m_1) \cdot E(m_2) \bmod n^2 = E\big((m_1 + m_2) \bmod n\big).$$

This property naturally extends to the addition of k ciphertexts, where the encrypted sum is obtained via repeated modular multiplications:

$$\prod_{i=1}^{k} E(m_i) \bmod n^2 = E\left(\left(\sum_{i=1}^{k} m_i\right) \bmod n\right).$$

Paillier enables an unbounded number of additions in theory, constrained only by the accumulated plaintext value modulo n. With appropriately chosen parameters, millions of additions can be performed securely and efficiently, highlighting its suitability for privacy-preserving aggregation in distributed learning frameworks.

3. Federated Learning

Federated learning (FL) has gained attention due to the ability of devices to collaboratively train models without sharing their local data. It has been applied in various applications such as finance [25], recommendation systems [26], and healthcare [27,28] to leverage distributed data without sending them. FL can be categorized into horizontal and vertical based on data partitioning. Horizontal FL [1] involves multiple clients/devices with datasets having the same features but different samples, while vertical FL [29] involves clients with datasets that share the same samples but have different features. In this work, we focus on the standard horizontal FL setting.

The federated training includes multiple clients and is coordinated by a server. Each client c has its local data $\mathcal{D}_c$. The process is iterative and consists of multiple

training rounds. In each round r, the server sends the model parameters (θ) to the devices. Each device then trains its local model for some local iterations given its $\mathcal{D}_c$. The devices then upload the updated model parameters again to the server. The server aggregates the model parameters received from the devices to obtain a new model and starts a new FL round by broadcasting the model to the clients. This process is repeated for multiple rounds until the model converges.

There are multiple challenges that face this federated learning process when deployed in resource-constrained environments. Techniques like [30,31] use the dropout mechanism to reduce the computation burden of training on the client side. For memory-constrained environments, optimized freezing mechanisms [32] and low-rank adaptation [33] are proposed for training from scratch and fine-tuning scenarios, respectively. Model compression techniques [34,35] are proposed for efficient communication between the server and clients.

Furthermore, this distributed setting opens up security vulnerabilities. From the client side, poisoning attacks [36] involve malicious clients manipulating their local data labels or model updates, aiming to corrupt the global model and introduce targeted vulnerabilities or backdoors. Countermeasures such as model-based analysis and robust aggregation are proposed to mitigate these types of attacks [37,38]. FL is also vulnerable to byzantine attacks to destabilize the federated process, where countermeasures include the detection of attackers and byzantine resilient aggregation [39,40]. HE has also been explored for privacy-preserving FL on the server side, particularly from a server that aims to infer information about clients [41,42].

4. 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 6 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 [8].

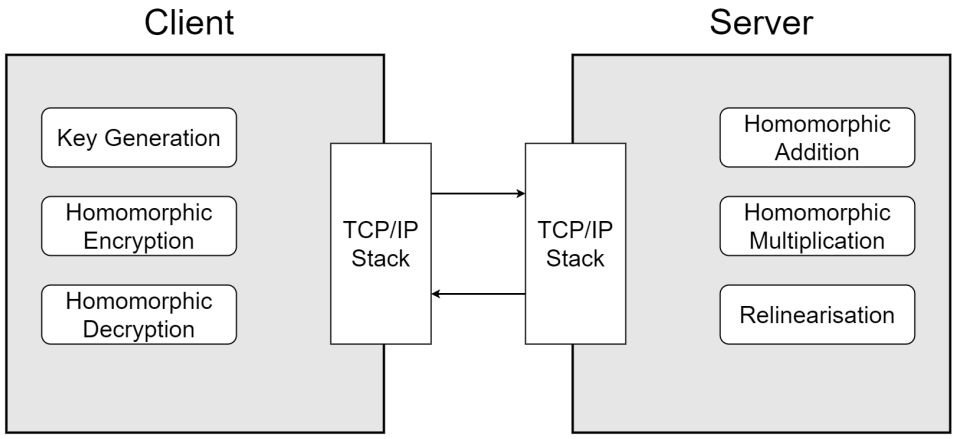

Fig. 2. The proposed Client-Server-Architecture with a TCP/IP stack for communication. The client generates keys, performs encryption and description. The homomorphic operations are done by the server.

4.1. *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 [43]. Microsoft SEAL [22] 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.

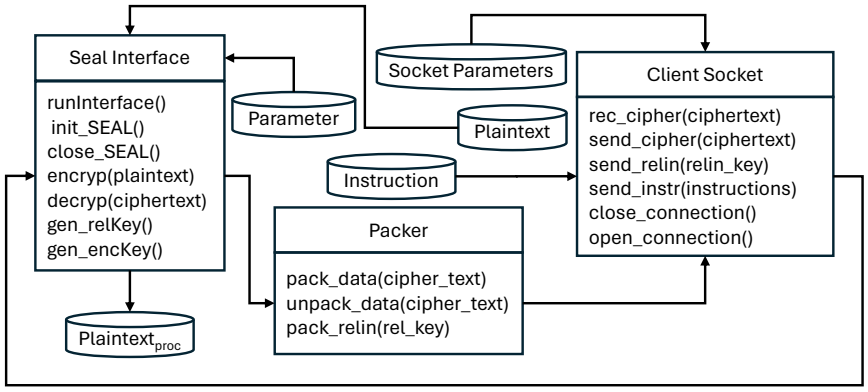

Fig. 3. Client architecture for homomorphic encryption. The diagram illustrates the interactions and dependencies between the SEALInterface, Packer, and SocketClient classes, highlighting their roles in facilitating secure computation and communication within the framework.

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 ($Plaintext_{proc}$).

4.2. *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 [21,44]. 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 polynomials with **4096** coefficients. The selected modulus q is a **180-bit** number. When multiplication is

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 ← encryption(plaintext, key_{en})
  Ciphertext_{pack} ← **pack**(Ciphertext)
  Send(Ciphertext_{pack})
  Receive(Ciphertext_{u})
  Ciphertext ← **unpack**(Ciphertext_{u})
  Plaintext_{proc} ← **decrypt**(ciphertext, key_{en})
  Plaintext ← Plaintext_{proc}

performed, the polynomials are shifted from q to Q. The modulus Q should be at least a **372-bit** integer [8].

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 ($\textbf{ciphertext}_u$) is sent back to the client.

Algorithm 2 Homomorphic Processing on Server side

Input: Key_{rel}, Ciphertext, Instructions
Output: Ciphertext_{u}
Initialize counter: **ctr** ← 0
Receive key_{rel}, ciphertext, instructions
Initialize output: $\textbf{ciphertext}_u$ ← ciphertext
while instructions ≠ empty **and** ctr < mul_depth **do**
  $\textbf{ciphertext}_u$ ← **Instruction**(ciphertext_{u}) {Execute instruction}
  If instruction is **multiply**:
    $\textbf{ciphertext}_u$ ← **relin**(ciphertext_{u},key_{rel}) {Perform reliniearization}
    Increment counter: **ctr** ← **ctr** + 1
Send ciphertext_{u}

5. Framework Adaptation for Federated Learning

As discussed, in the federated learning process, the clients share their model with the server to aggregate them and broadcast them again to the client. The shared models from the clients in each round can still leak information about the data of the clients. To incorporate homomorphic encryption to ensure privacy in this process, clients encrypt their updated local model parameters using a homomorphic encryption scheme before uploading them to the server. The server then performs aggregation directly on the encrypted updates, without the need to decrypt them. Once aggregation is complete, the server broadcasts the encrypted global model to all clients. Each client then decrypts the aggregated model using its private key and proceeds with the next round of local training. Figure 4 illustrated the flow discussed in the federated learning round.

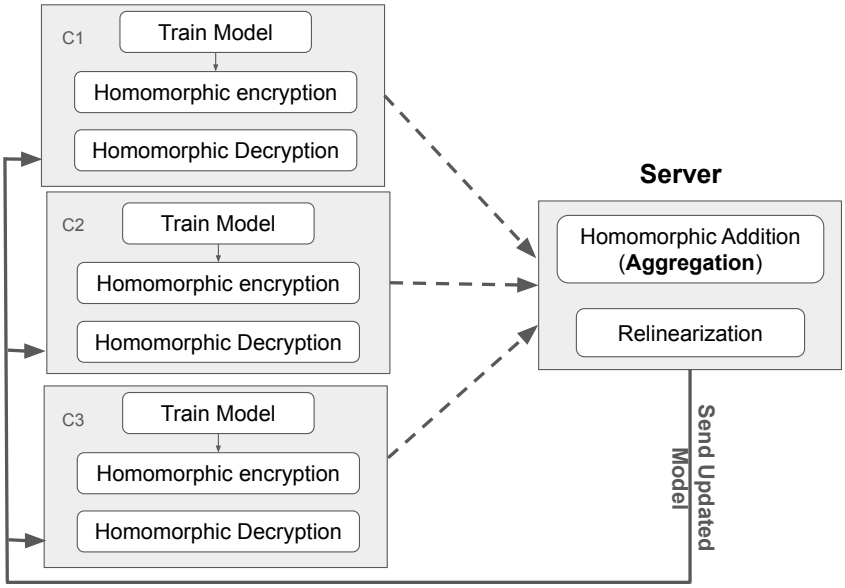

Fig. 4. Flow of the federated learning round using HE on the server.

Since the aggregation process consists mainly of addition, using FV is not the optimal strategy. Therefore, we use Paillier [45], an additive and less computationally expensive homomorphic encryption scheme on the server. The server would add all parameter values from the received models and return the encrypted model values. Each client would then decrypt the received model and then divide each parameter by the number of participating clients. The number of participating clients can be known prior to the FL process to the clients themselves or shared by the server given the number of clients that participated.

In our setup, we consider horizontal federated learning, where data is partitioned across clients. We consider MNIST, a digit image classification task, and employ LeNet-5 [46] as our model. Each client would train for 10 local iterations per round using stochastic gradient descent.

6. Results

We begin our evaluation with the client-server framework and subsequently extend it to incorporate the federated learning use case with homomorphic encryption.

6.1. *Client-server framework*

The experimental setup, shown in Figure 5, 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.

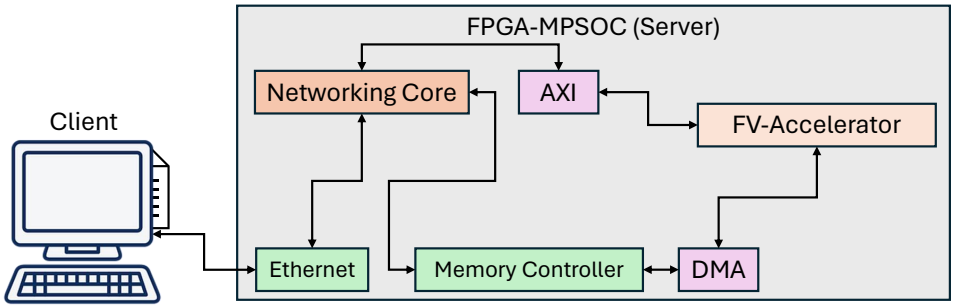

Fig. 5. The implemented system. The opensource FV-Accelerator is integrated with communication infrastructure. The client runs on a PC sends the ciphertext and receives processed ciphertext which it decrypts back to plaintext.

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 1. 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 1. 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 2 provides an overview of the calculation times.

Table 2. 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 [22] and OpenFHE [43] 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.

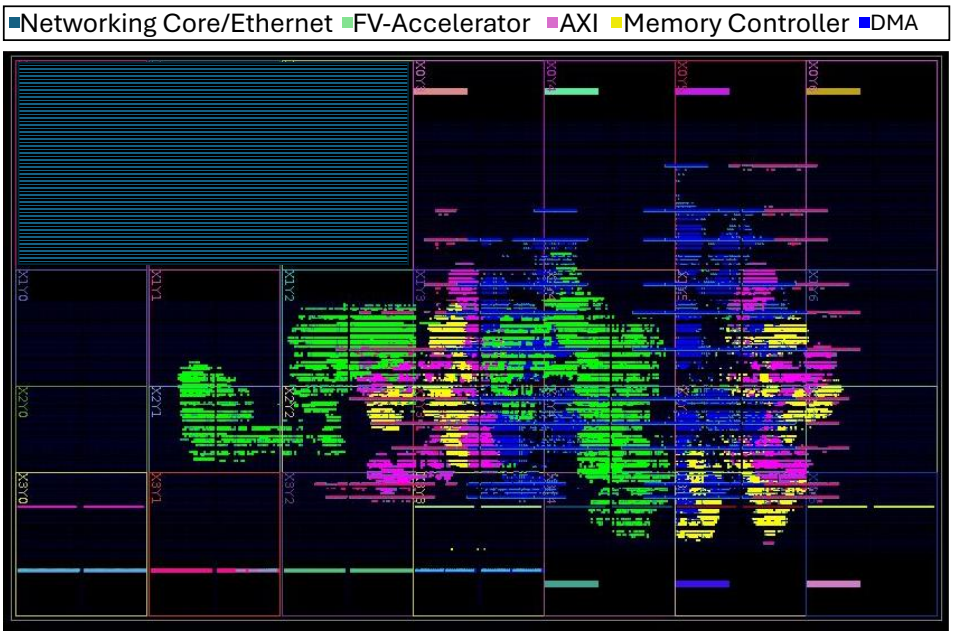

Fig. 6. The Floor plan on the FPGA. The ethernet and the networking core are implemented in the PS. The resource utilization is dominated by the open source FV-accelerator.

Although timing and accelerating execution are our main focus, we also report the utilization of resources to implement the server on FPGA. Figure 6 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 [9] 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.

Table 3. Resource utilization for implementing our serverside on the FPGA-MPSoC using the opem source FV-accelerator.

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

We also look at the full detailed implementation results on the PL as shown in Table 3. In general, we do not use much resources, we have a relatively high utilization of BRAm but 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.

6.2. *Federated learning Framework*

We extend our evaluation to a federated learning (FL) scenario. The setup consists of one central server and three federated clients, all running on local PCs. Communication between the server and the clients is established via Ethernet. We explore two variants of the server: (i) a ZCU104 board employing an accelerated Paillier-based additive homomorphic encryption (HE) scheme, and (ii) a baseline implementation using a standard Intel i5 Quad-Core CPU for plaintext model averaging. In the hardware-accelerated setup, we integrate a single AXI-based Paillier adder, which is controlled by the ARM core on the ZCU104 board. The ARM core is responsible for managing all communication and controlling the AXI-based Paillier adder. The FPGA resource utilization of this design is summarized in Table 4, and the corresponding floorplan implementation is shown in Figure 7.

LUT	LUTRAM	FF	DSP	BUFG
59698 (2%)	70 (0.07%)	2687 (0.58%)	50 (2.89%)	1 (0.18%)

Table 4. Resource utilization for AXI-based paillier adder on ZCU104 board

We record the normalized time for the aggregation of one parameter given three clients models in Figure 8. Our accelerated homomorphic encryption (HE) setup takes approximately 123.0× more time than standard plaintext averaging on the CPU. This substantial overhead is expected, as HE schemes—despite hardware acceleration—are inherently more computationally intensive due to the encrypted domain arithmetic and modular operations they require.

7. 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,

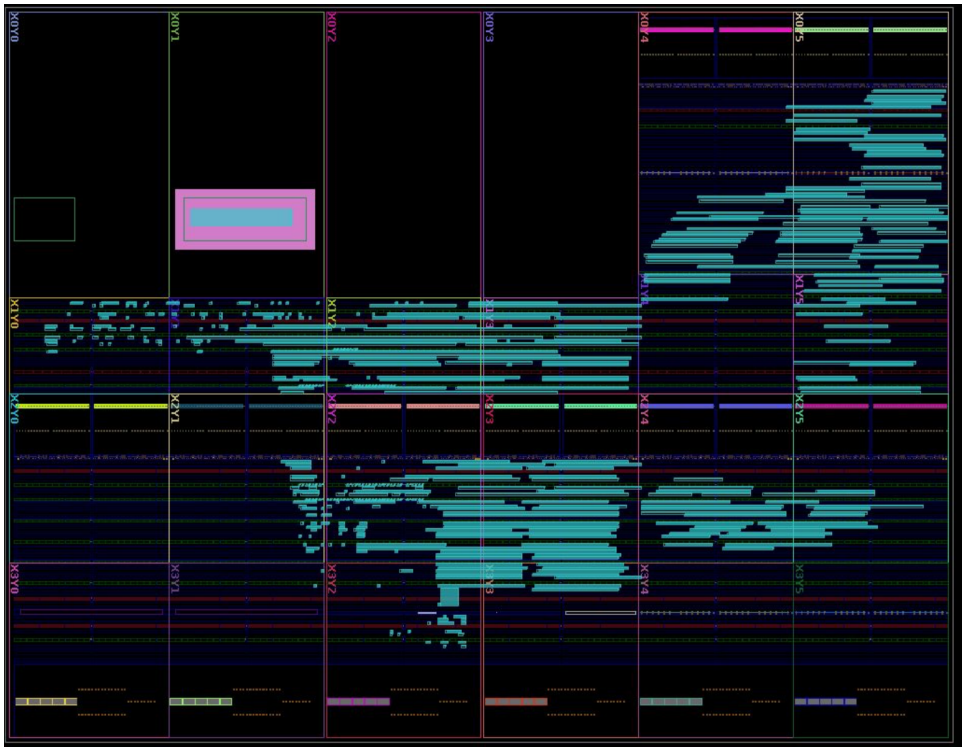

Fig. 7. The floor plan for AXI-based Paillier adder on ZCU104 board.

we intend to implement a bootstrapping mechanism to support Fully Homomorphic Encryption as well.

Acknowledgments

A preliminary version of this work has been published in ICM 2024 [18]. This work was supported in part by BMBF through the Software Campus Projects HE-Trust under Grant 01IS23066.

References

1. B. McMahan, E. Moore, D. Ramage, S. Hampson and B. A. y Arcas, Communication-efficient learning of deep networks from decentralized data, in *Artificial intelligence and statistics*, (PMLR, 2017), pp. 1273–1282.
2. F. Hu, M. Jin, Y. Zhang, X. Fang and M. Guizani, Federated learning-based traffic flow prediction model in intelligent transportation systems, *Journal of Circuits, Systems and Computers* **34**(03) (2025) p. 2550074.
3. J. Du and K. Yang, Nids-flgdp: Network intrusion detection algorithm based on

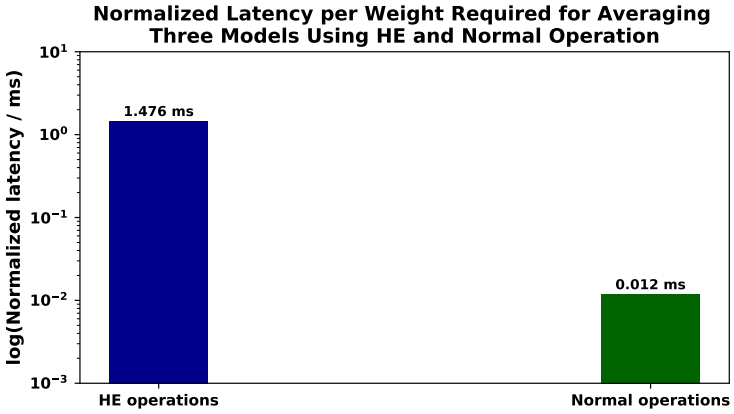

Fig. 8. Normalized latency per parameter for averaging three models using HE and normal operations

gaussian differential privacy federated learning, *Journal of Circuits, Systems and Computers* **33**(03) (2024) p. 2450048.

4. K. Pfeiffer, M. Rapp, R. Khalili and J. Henkel, Federated learning for computationally constrained heterogeneous devices: A survey, *ACM Comput. Surv.* **55**(July 2023).
5. Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang and H. Qi, Beyond inferring class representatives: User-level privacy leakage from federated learning, in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, (IEEE, 2019), pp. 2512–2520.
6. L. Melis, C. Song, E. De Cristofaro and V. Shmatikov, Exploiting unintended feature leakage in collaborative learning, in *2019 IEEE symposium on security and privacy (SP)*, (IEEE, 2019), pp. 691–706.
7. C. Huang, S. Wei and A. Fu, An efficient privacy-preserving attribute-based encryption with hidden policy for cloud storage, *Journal of Circuits, Systems and Computers* **28**(11) (2019) p. 1950186.
8. J. Fan and F. Vercauteren, Somewhat Practical Fully Homomorphic Encryption Cryptology ePrint Archive, Paper 2012/144, (2012).
9. S. S. Roy, K. Järvinen, J. Vliegen, F. Vercauteren and I. Verbauwhede, HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation, *IEEE TC* **67** (2018).
10. W. Wang, Y. Hu, L. Chen, X. Huang and B. Sunar, Accelerating fully homomorphic encryption using GPU, in *Conference on High Performance Extreme Computing*, (IEEE, 2012), pp. 1–5.
11. H. Nassar, L. Bauer and J. Henkel, HBMorphic: FHE Acceleration via HBM-Enabled Recursive Karatsuba Multiplier on FPGA, in *FCCM*, (IEEE, 2024).
12. H. Nassar, L. Bauer and J. Henkel, Turbo-fhe: Accelerating fully homomorphic encryption with fpga and hbm integration, *IEEE Design & Test* **42**(3) (2025) 86–93.
13. J.-B. QIAN, H.-B. XU, Y.-S. DONG, X.-J. LIU and Y.-L. WANG, Fpga acceleration window joins over multiple data streams, *Journal of Circuits, Systems and Computers* **14**(04) (2005) 813–830.
14. G. CAFFARENA, C. PEDREIRA, C. CARRERAS, S. BOJANIC and O. NIETO-TALADRIZ, Fpga acceleration for dna sequence alignment, *Journal of Circuits, Sys-*

tems and Computers **16**(02) (2007) 245–266.
15. Y. Yang, W. Long, R. Kannan and V. K. Prasanna, FPGA Acceleration of Rotation in Homomorphic Encryption Using Dynamic Data Layout, in *FPL*, (IEEE, 2023), pp. 174–181.
16. Y. Yang, H. Zhang, S. Fan, H. Lu, M. Zhang and X. Li, Poseidon: Practical Homomorphic Encryption Accelerator, in *HPCA*, (IEEE, 2023), pp. 870–881.
17. B. Bulut, S. N. Bicakci and I. San, HW/SW Co-Design of TFHE Homomorphic OR Gate via NTT-based Polynomial Multiplication on a programmable SoC, in *Innovations in Intelligent Systems and Applications Conference*, (IEEE, 2022).
18. S. Bothe, H. Nassar, L. Bauer and J. Henkel, Client-server framework for fpga acceleration of fan-vercauteren-based homomorphic encryption, in *2024 International Conference on Microelectronics (ICM)*, (IEEE, 2024), pp. 1–5.
19. X. Yi, R. Paulet and E. Bertino, *Homomorphic Encryption*, in *Homomorphic Encryption and Applications*, (Springer International Publishing, Cham, 2014), Cham, pp. 27–65.
20. I. Chillotti, M. Joe and P. Paillier, Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks (2020).
21. S. Sinha Roy, F. Turan, K. Jarvinen, F. Vercauteren and I. Verbauwhede, FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data, in *HPCA*, (IEEE, 2019).
22. Microsoft SEAL (release 4.1)(January 2023), Microsoft Research.
23. I. Blanco-Chacón, Ring Learning with Errors: a Crossroads between Post-Quantum Cryptography, Machine Learning and Number Theory, *Irish Math. Soc. Bull* **86** (2020) 17–46.
24. M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai and V. Vaikuntanathan, Homomorphic Encryption Security Standard, tech. rep., HomomorphicEncryption.org (Toronto, Canada, 2018).
25. Y. Li, C. Chen, N. Liu, H. Huang, Z. Zheng and Q. Yan, A blockchain-based decentralized federated learning framework with committee consensus, *IEEE Network* **35**(1) (2020) 234–241.
26. Z. Jie, S. Chen, J. Lai, M. Arif and Z. He, Personalized federated recommendation system with historical parameter clustering, *Journal of Ambient Intelligence and Humanized Computing* **14**(8) (2023) 10555–10565.
27. I. Dayan, H. R. Roth, A. Zhong, A. Harouni, A. Gentili, A. Z. Abidin, A. Liu, A. B. Costa, B. J. Wood, C.-S. Tsai *et al.*, Federated learning for predicting clinical outcomes in patients with covid-19, *Nature medicine* **27**(10) (2021) 1735–1743.
28. X. Xu, H. Peng, M. Z. A. Bhuiyan, Z. Hao, L. Liu, L. Sun and L. He, Privacy-preserving federated depression detection from multisource mobile health data, *IEEE transactions on industrial informatics* **18**(7) (2021) 4788–4797.
29. S. Hardy, W. Henecka, H. Ivey-Law, R. Nock, G. Patrini, G. Smith and B. Thorne, Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption, *arXiv preprint arXiv:1711.10677* (2017).
30. D. Wen, K.-J. Jeon and K. Huang, Federated dropout—a simple approach for enabling federated learning on resource constrained devices, *IEEE wireless communications letters* **11**(5) (2022) 923–927.
31. M. Rapp, R. Khalili, K. Pfeiffer and J. Henkel, Distreal: Distributed resource-aware learning in heterogeneous systems, in *Proceedings of the AAAI Conference on Artificial Intelligence*, **36**(7)2022, pp. 8062–8071.
32. K. Pfeiffer, R. Khalili and J. Henkel, Aggregating capacity in fl through successive

layer training for computationally-constrained devices, *Advances in Neural Information Processing Systems* **36** (2023) 35386–35402.
33. S. Babakniya, A. R. Elkordy, Y. H. Ezzeldin, Q. Liu, K.-B. Song, M. El-Khamy and S. Avestimehr, Slora: Federated parameter efficient fine-tuning of language models, *arXiv preprint arXiv:2308.06522* (2023).
34. S. M. Shah and V. K. N. Lau, Model compression for communication efficient federated learning, *IEEE Transactions on Neural Networks and Learning Systems* **34**(9) (2023) 5937–5951.
35. R. Hönig, Y. Zhao and R. Mullins, Dadaquant: Doubly-adaptive quantization for communication-efficient federated learning, in *International Conference on Machine Learning*, PMLR2022, pp. 8852–8866.
36. G. Xia, J. Chen, C. Yu and J. Ma, Poisoning attacks in federated learning: A survey, *Ieee Access* **11** (2023) 10708–10722.
37. X. Li, Z. Qu, S. Zhao, B. Tang, Z. Lu and Y. Liu, Lomar: A local defense against poisoning attack on federated learning, *IEEE Transactions on Dependable and Secure Computing* **20**(1) (2021) 437–450.
38. S. Awan, B. Luo and F. Li, Contra: Defending against poisoning attacks in federated learning, in *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*, Springer2021, pp. 455–475.
39. X. Ma, Q. Jiang, M. Shojafar, M. Alazab, S. Kumar and S. Kumari, Disbezant: Secure and robust federated learning against byzantine attack in iot-enabled mts, *IEEE Transactions on Intelligent Transportation Systems* **24**(2) (2022) 2492–2502.
40. J. Xu, Z. Zhang and R. Hu, Achieving byzantine-resilient federated learning via layer-adaptive sparsified model aggregation, in *2025 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, IEEE2025, pp. 1508–1517.
41. W. Liu, N. A. Koca and C.-H. Chang, Efficient fast additive homomorphic encryption cryptoprocessor for privacy-preserving federated learning aggregation, in *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2024, pp. 1–6.
42. Q. Xie, S. Jiang, L. Jiang, Y. Huang, Z. Zhao, S. Khan, W. Dai, Z. Liu and K. Wu, Efficiency optimization techniques in privacy-preserving federated learning with homomorphic encryption: A brief survey, *IEEE Internet of Things Journal* **11**(14) (2024) 24569–24580.
43. OpenFHE, An open-source project that provides efficient extensible implementations of the leading post-quantum Fully Homomorphic Encryption (FHE) schemes(December 2022).
44. F. Turan, S. S. Roy and I. Verbauwhede, HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA, *IEEE TC* **69** (2020).
45. P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in *International conference on the theory and applications of cryptographic techniques*, (Springer, 1999), pp. 223–238.
46. Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, Gradient-based learning applied to document recognition, *Proceedings of the IEEE* **86**(11) (1998) 2278–2324.