FHEMaLe:Framework for Homomorphic Encrypted Machine Learning

B PRADEEP KUMAR REDDY, Indian Institute of Technology, India
SAMEEKSHA GOYAL, Indian Institute of Technology, India
RUCHIKA MEEL, Indian Institute of Technology, India
AYANTIKA CHATTERJEE, Indian Institute of Technology, India

Machine learning (ML) has revolutionized various industries by leveraging predictive models and data-driven insights, often relying on cloud computing for large-scale data processing. However, this dependence introduces challenges such as bandwidth constraints and network latency. Edge computing mitigates these issues by enabling localized processing, reducing reliance on continuous cloud connectivity, and optimizing resource allocation for dynamic workloads. Given the limited computational capacity of sensory nodes in ML systems, edge devices provide an effective solution by offloading processing tasks. However, a critical challenge in this paradigm is to ensure user privacy while handling sensitive data both in the cloud and in edge processing. To address this, we propose a Fully Homomorphic Encryption (FHE) enabled framework that enables ML computations directly on encrypted data, eliminating need for decryption. The main challenge to design such framework is that ML complex implementation steps need to be revisited with suitable optimizations to match FHE processing requirements. There are different standard libraries to support basic computation blocks on which encrypted ML processing is to be developed. These libraries vary in supported computation operators, computational complexity and memory demands. Those in-turn introduces latency and throughput challenges, especially on resource-constrained edge nodes. For example, in general HE library CKKS(Cheon-Kim-Kim-Song) with packing and approximate homomorphic operation support is known to be the best choice for privacy preserving AI algorithm implementation. However, analysis shows leveled CKKS is limited in implementing complex operators and hence not suitable for few specific ML algorithms like KNN, Logistic Regression or general activations in NN etc without any approximation. To avoid accuracy drops associated with approximations, Torus based FHE library (TFHE) can be a better choice to make certain ML implementations feasible. Moreover, our study shows compared to TFHE, CKKS with huge memory requirement is not suitable for resource constrained edge. Thus, underlying library choice to design such framework is crucial considering the trade-off between latency and accuracy. In this work, we propose an integrated framework FHEMaLe for encrypted ML processing which takes model architecture, desired accuracy, and platform preference as inputs and based on that appropriate execution environment is selected: a cloud platform leveraging the CKKS homomorphic encryption library or an edge platform using the TFHE library. Further, analysis shows the limitation of performing FHE ML on a single edge device and hence our framework partitions encrypted data, transmits it via a fabric API, and performs distributed encrypted ML computations across the edge cluster. We implement distributed ML inference for algorithms such as K-Nearest Neighbors (KNN) (Cloud CKKS=248 sec, Edge TFHE=37 min), Support Vector Machine (SVM) (Cloud CKKS=18 sec, Edge TFHE=4.15 min), and Logistic Regression (LR) (Cloud CKKS=17 sec, Edge TFHE=7.82 min) on a cluster of 11 edge nodes. This work explains why KNN suffers from a major performance bottleneck in encrypted domain and may not be a great choice for encrypted ML processing. Furthermore, our encrypted operators are capable of supporting encrypted NN processing (Cloud CKKS= 57 sec), but we explain why CKKS is a preferred choice in this case. The distributed nature of our implementation shows a promise of further improvement and scalability with the support of larger cluster.

Additional Key Words and Phrases: Machine Learning, IoT, Security, Fully Homomorphic Encryption, Distributed Computing, Privacy

1 INTRODUCTION

The rapid proliferation of Internet of Things (IoT) devices in smart environments has catalyzed a paradigm shift from centralized cloud computing to localized edge processing, commonly referred to as edge Machine Learning (ML). Edge ML enables data analysis close to its source, offering significant advantages in terms of energy efficiency, cost-effectiveness, and scalability when processing large datasets. This is particularly beneficial in scenarios with limited or intermittent connectivity, such as industrial, remote, or mobile deployments, where uninterrupted operation is critical. In domains like healthcare and clinical applications, edge-based IoT facilitates sensor-driven treatments and monitoring, using robust edge nodes to execute complex ML algorithms while keeping

Authors' addresses: B Pradeep Kumar Reddy, Indian Institute of Technology , , Kharagpur, , India, ; Sameeksha Goyal, Indian Institute of Technology, , Kharagpur, India; Ruchika Meel, Indian Institute of Technology, , Kharagpur, India; Ayantika Chatterjee, Indian Institute of Technology, , Kharagpur, , India.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
Manuscript submitted to ACM

data local. These capabilities underscore the vital role of edge ML in delivering low-latency, scalable, and efficient solutions for diverse applications. Despite its benefits, edge ML faces substantial security and privacy challenges. Hence, considering the present requirements, it is important to revisit both cloud and edge ML to provide a complete privacy-preserving AI solution. Since privacy enabled cloud solutions are mostly explored [1], [6], [25], [32], [33], [54], we primarily focus on privacy-preserving ML solutions on edge in this work. Finally, we propose an integrated framework FHEMaLe, suitable for both cloud and edge privacy preserving ML.

The massive volume and sensitivity of data generated by IoT devices increase the risk of data breaches, adversarial attacks, and leakage [2], [27]. Edge devices are inherently resource-constrained in terms of memory, compute, and energy, making it difficult to apply computationally intensive security mechanisms. Furthermore, their distributed nature complicates secure coordination and data sharing, increasing the risk of exposure during transmission and processing if not robustly protected. These factors demand privacy-preserving solutions that balance security with the tight resource budgets of edge devices. For example, low cost sensor nodes are placed to collect information from the remote locations, but actual processing on data requires cloud or high-end edge device support, as low-end edge nodes are not sufficient for costly ML processing. Hence, ensuring privacy is important while tranferring and processing data from one IoT node to another.

Several techniques have been proposed to address privacy in distributed ML in cloud as well as edge. Differential Privacy (DP) and Multi-Party Computation (MPC) are notable examples. DP injects noise into data or model outputs, reducing accuracy, which may not be acceptable for critical applications [31]. MPC, though privacy-preserving, introduces high communication and computation overhead, requiring frequent interaction between parties, which is impractical in bandwidth-limited edge environments. Homomorphic Encryption (HE) [22], [55] emerges as a promising alternative, as it enables computation directly on encrypted data without the need for decryption, preserving accuracy while eliminating plaintext exposure. Fully Homomorphic Encryption (FHE) further allows to compute arbitrary encrypted processing theoretically. However, its practical deployment on devices as privacy-preserving edge ML is challenging.

While translation of traditional algorithms into the FHE domain depends on the libraries present in the FHE literature [38], the main challenge lies in selecting the library for FHE processing. Torus-based FHE library [18] and its variants NuFHE/OpenFFHE [41], [43] support precise Boolean operations, which are ideal for gate level modeling of exact encrypted ML operators. However, gate level homomorphic processing involves costly denoising module, Bootstrapping [18] and makes overall ML prediction slower. Hence, CKKS [17] is considered efficient and the best choice with approximate arithmetic and packing support. However, ML models such as KNN, SVM, decision trees, Nave Bayes, and LR demand precise operations: exact comparisons, selection of minimum/maximum, and discrete decision rules that cannot be efficiently or accurately represented in the leveled CKKS scheme. CKKS is tailored for approximate, SIMD-parallel real-number arithmetic and lacks native support for bit-level logic and branching. While it is formally possible to approximate comparisons using high-degree polynomials, such techniques introduce significant approximation error and computational overhead, and fail to scale in circuits requiring repeated exact decisions, ultimately compromising both performance and accuracy. On the other hand, TFHE variants inherently supports Boolean logic, including gates like AND, XOR, and multiplexers, through fast functional bootstrapping that preserves exactness of bit-level operations, making it a better fit for algorithms that depend on exact comparison and conditional flows.

With these motivations, we implement common ML algorithms like KNN, SVM, and LR predictions with the same encrypted baseline so that they can be easily extended to the ensemble framework. However, it is to highlight that our work is not limited to the mentioned ML algorithms, and all the reported operators are sufficient to realise SDG, Naive Bayes, Decision Tree, and other basic ML algorithms. We demonstrate why encrypted KNN is distinct and can not be the first choice for encrypted domain implementation. We keep a separate discussionscussion on DNN, where libraries with separate SIMD supported FHE processing are more suitable to reduce latency. That leads to our discussion on the choice of library that differs with different requirements.

Unlike prior studies that focus only on specific ML operations [33], [1], [54], [32], we present a complete framework to support different ML algorithms in their encrypted form both in cloud and edge. Analysing the memory overhead, we justify the usage of TFHE over CKKS on Raspberry Pi clusters to horizontally scale FHE-based computations, enabling secure, low-latency machine learning on resource-constrained edge devices. The computational demands of FHE ML (including basic operations like bootstrapping, encrypted polynomial evaluations) require significant processing power, far beyond what a single Pi can sustain with acceptable latency. Since, vertical scaling of a Pi (i.e., upgrading its hardware resources) is not feasible due to its design limits, we adopt a horizontally scalable architecture, a distributed cluster of Raspberry Pi nodes, which enables parallel execution of encrypted tasks. This approach accelerates processing, improves fault tolerance, and allows incremental scaling without large capital investments as required to perform continuous processing in costly cloud architecture. However, implementing ML algorithms under FHE in a distributed edge cluster introduces new challenges as existing ML algorithms are not inherently designed for encrypted or parallel execution. For example, KNN requires encrypted distributed sorting, SVM requires encrypted kernel evaluations across nodes, and LR needs distributed encrypted division and sigmoid approximation, which are computationally heavy under FHE. These tasks demand operator-level redesign to ensure they are FHE-compatible with parallelization and optimized for edge constraints, without compromising privacy or significantly inflating latency.

Our implementation also shows the feasibility of NN processing with the same encrypted TFHE baseline. However, NN exhibits fault tolerance and can continue functioning correctly or with minimal performance loss even when parts of the network experience errors due to some approximation. Hence, CKKS is a better choice with suitable approximations to maintain faster prediction. However, each linear or nonlinear layer (e.g., convolutions, activations) must be supported by encrypted counterparts, often relying on polynomial approximations of functions like ReLU or sigmoid. These approximations dramatically increase multiplicative depth and ciphertext size, placing heavy demands on both memory and compute resources with CKKS. Multiplicative depth refers to the maximum number of sequential ciphertext multiplications (i.e., multiplication layers) that an encrypted computation can support before decryption fails due to excessive noise. On the other hand, TFHE requires bootstrapping and noise management for deep circuits, significantly increasing latency, but squeezing memory requirements on constrained edge devices. Considering all these aspects, we propose a **F**ramework for **H**omomorphic **E**ncrypted **Ma**chine **Le**arning (FHEMaLe), where our actual contributions are as follows:

(1) First, we analyze how the proper choice of FHE library differs according to the choice of ML models and affects the overall performance strongly. We explore the realization of the basic ML operators on encrypted data, where the algorithm needs to be realized in the circuit-based representation, and computations should be performed using FHE gates. Our observation shows that CKKS is claimed to be faster for encrypted ML implementations in general. However, few common ML algorithms are not feasible to implement with SIMD support in CKKS in their exact form. In these cases, bitwise homomorphic TFHE variants are a better choice. Otherwise, suitable approximations are required to make the ML implementations feasible, which may lead to a drop in accuracy. Further, CKKS based implementations require huge memory overhead, which is not suitable for edge ML processing.

(2) Considering all the above constraints, we propose an end-to-end framework to realize encrypted ML/NN algorithms where the user can input the model details, minimum accuracy requirement and platform details as shown in Fig. 1. The initial decision block evaluates the input model in plaintext domain to check if the required operators can be realized by SIMD supported CKKS library for faster performance. Choice of platform also has an immediate constraint in case of selecting the underlying FHE library. If some operator implementations are not feasible or the platform requirement is on edge, further approximation or TFHE based implementation will be chosen in the automated framework.

(3) Finally, evaluation and minimization of computational overhead introduced by encrypted data operations with distributed and concurrent computing on the edge devices network is also explored. However, our analysis shows direct adaptation of encrypted ML algorithms implemented on single node implementations is not suitable for distributed clusters and we need to revisit the algorithm design considering distributed processing underneath. Implementation of the encrypted framework is evaluated using different HE libraries: NuFHE TFHE library [41], OpenFHE TFHE library [43] and OpenFHE CKKS [16], where NuFHE can exploit GPU advantages and OpenFHE claims to support faster bootstrapping. For the cloud environment, the selection of CKKS [14] or TFHE [41], [18] is preferred depending on accuracy and latency requirements. In the edge environment, OpenFHE TFHE [43] is chosen for its memory efficiency, addressing the challenges posed by ciphertext expansion in resource-constrained settings.

(4) In edge cluster development, the system adopts a star topology in which a master node orchestrates worker assignments and monitors progress. Heartbeat signals are periodically exchanged, and upon detecting a node failure, the master reassigns the workload to a standby or available node. This design ensures robustness in low-cost edge clusters, where node dropout is more likely than in data centres.

The overall paper is organized as follows: Section 2 presents a brief introduction to related works and Section 3 presents the essential preliminary concepts. Section 4 explains system and threat model, proposed encrypted edge machine learning framework and challenges and limitations of adapting single-edge implementation of encrypted ML on a distributed platform. Section 4.4,

Table 1. ML Encrypted Models Training and Prediction Operations Complexity

	SVM	DT	LR	NB	RF
Prediction					
FHE_Addition	✓	×	✓	✓	×
FHE_Subtraction	×	×	×	✓	×
FHE_Multiplication	✓	×	✓	✓	×
FHE_Division	×	×	✓	✓	×
FHE_Comparison	✓	✓	✓	×	✓
FHE_Absolute	×	✓	×	×	✓
FHE_Square root	×	×	✓	×	×
FHE_Exponent	×	×	×	✓	×
FHE_Mean	×	×	×	✓	×
Training					
FHE_Addition	✓	✓	✓	✓	✓
FHE_Subtraction	✓	✓	✓	✓	✓
FHE_Multiplication	✓	✓	✓	✓	✓
FHE_Division	✓	✓	✓	✓	✓
FHE_Comparison	✓	✓	✓	×	✓
FHE_Log	×	✓	×	×	✓
FHE_Square root	×	×	×	✓	×
FHE_Exponent	×	×	✓	✓	✓
FHE_Mean	×	×	×	✓	×
FHE_Variance	×	×	×	✓	×

4.5 details the implementation of encrypted SVM and encrypted LR for the distributed platform respectively. Section 5 details the feasibility of encrypted KNN in an edge environment. Section 6, we explain how to select library for DL and LLM based on platforms. Section 7 elucidates the security analysis of the proposed framework. Finally, Section 8 demonstrates the timing requirement of the proposed framework and Section 9 mentions the conclusion and some future directions of this work.

2 RELATED WORKS AND MOTIVATION

ML on edge devices and encrypted ML algorithm implementation are active areas of research. In [40], researchers explore the deployment of ML and DL systems at the edge, discussing common architectures used in DL. The authors in [61] implement plaintext Random Forest, SVM, and Multi-Layer Perceptron on Raspberry Pi to demonstrate the feasibility of running state-of-the-art ML algorithms on edge IoT devices. Similarly, studies such as [57], [59], and [56] investigate various ML applications on edge devices, showcasing their real-time efficiency. However, these works do not address the security of edge-based ML frameworks.

Research on encrypted ML has primarily focused on cloud-based solutions. In [7], an ensemble of hyperplane decision, Naive Bayes, and Decision Trees using AdaBoost is proposed, but the use of multiple encryption techniques introduces security overhead. The authors in [44] implement an encrypted NB classifier, which has inherent classification limitations. Studies on encrypted analytics mainly explore LR [24], [11], [26] and SVM [35], yet they are largely cloud-based. A privacy-preserving medical diagnosis mechanism, LPME, employing encrypted model parameters and a two-trapdoor public-key cryptosystem on edge devices, is introduced in [37]. Additionally, [12] presents an efficient scheme for updating and searching encrypted data.

Several works focus on privacy-preserving ML at the edge. In [20], PFMLP incorporates partially homomorphic encryption (HE) for secure MLP training. Article [48] introduces a privacy-preserving AI framework using the BGV FHE algorithm for encrypted QoS data processing in edge networks. HE techniques based on the Paillier and RSA cryptosystems are used in [58] for securing edge computing with blockchain, while [29] employs an advanced Paillier cryptosystem for cloud-based encrypted computations. The authentication-supported homomorphic encryption (AHEC) scheme in [52] focuses on partial HE for MEC-based IoT systems, remaining limited to cloud environments. Collaborative encrypted edge learning frameworks such as [28] focus on ML but overlook security considerations. Surveys like [53] highlight the role of HE in data privacy, contrasting traditional encryption methods. Further advancements include [60], which introduces quantum-resilient partial HE, and [36], which presents encrypted posture-tracking wearable systems. Security in IoT remains critical, as examined in [47], which identifies vulnerabilities and recommends robust protections.

Most of the existing works related to machine learning (ML) algorithms on encrypted data are either confined to cloud computation or tailored to specific schemes where partial homomorphic encryption (HE) suffices. However, these approaches are limited in scope and application. In this paper, we foucs on proposing a framework that will take user model, minimum accuracy requirement and platform choice as inputs to perform encrypted ML computations with suitable choice of underlying HE libraries. In subsequent sections, we detail how choice of library and platform affect encrypted ML processing.

3 PRELIMINARIES

3.1 FHE with Torus (TFHE)

We consider NuFHE [41] as the underlying library, which is the GPU variant of TFHE [55]. NuFHE using CUDA and OpenCL GPU backends. We have used PyOpenCL, which provides pythonic access to OpenCL [42], which is the open standard for parallel programming of heterogeneous systems, for example, CPUs, GPUs, DSPs, and FPGAs. The TFHE scheme is a variant of the LWE-based (Learning with Errors) FHE, which supports the fastest bitwise homomorphy. Moreover, this bitwise homomorphy makes the scheme flexible to be implemented with any input bit-size as per application requirements. This is the main motivation for choosing NuFHE for our framework. Following, we explain the mathematical formulations of some of the key algorithms in TFHE. Considering λ=128 is the the security parameter, $\mathbb{B}\ \epsilon\ [0,1]$, $\mathbb{T}(Torus) = \mathbb{R}(realnumbers)/\mathbb{Z}(integers)(Real\ modulo\ 1\ set)$, $\mathbb{Z}[x]$(ring of polynomials)= $\mathbb{Z}_N[x]/x^N + 1$, $\mathbb{T}_N[x] = \mathbb{R}[x]/x^N + 1\ mod\ 1$, $\mathbb{B}_N[x]$= polynomials in $\mathbb{Z}_N[x]$ with binary coefficients, message(M) ϵ [0,1], TRLWE= Ring LWE over torus ,TRGSW= Ring GSW over Torus, TLWE= LWE over Torus, TFHE basic algorithms are detailed as follows [18]:

- $KeyGen(\lambda)$ generates TRLWE secret key(s) $\boldsymbol{\epsilon}\ \mathbb{B}_N[X]$, TRGSW bootstrapping key and TRLWE cloud key(p).
- $Enc(u, v)$ picks a uniform random vector $\boldsymbol{a}\ \boldsymbol{\epsilon}\ \mathbb{T}_N[x]$, outputs ciphertext $\langle u, v\rangle$=$\langle a, \langle a, s\rangle + e + M\rangle$, where e (small Gaussian error) $\boldsymbol{\epsilon}\ \mathbb{T}_N[x]$, $\langle u, v\rangle \boldsymbol{\epsilon} TRLWE$, M= message $\boldsymbol{\epsilon}$ [0,1].
- $Dec(s, \langle u, v\rangle)$: By decryption function $\langle v\text{-}s.u\rangle$ is rounded to the nearest number$M$.
- $Eval(f, c_1, ... c_t)$: Eval function evaluates addition and multiplication (using homomorphic gates) over the ring taking input as TRLWE ciphertext. Bootstrapping key is generated from TRGSW cloud key, and resultant TRLWE ciphertext is returned. In the process of evaluation, bootstrap function is the main denoising module which extracts sample TRLWE ciphertext, takes inputs TRLWE switching-key, TRGSW bootstrap-key, and returns new TLWE ciphertext with lower noise level.

In TFHE and its variants, bootstrapping is the most costly operation, making its optimization a key research challenge. Despite various improvements, it remains computationally expensive. OpenFHE implements an improved bootstrapping method [39], reducing execution time to 75ms from 126ms, making it faster than NuFHE. While NuFHE leverages GPU acceleration, OpenFHE benefits from optimized bootstrapping. To develop the end-to-end ML framework Framework for Homomorphic Encrypted Machine Learning (FHEMaLe), we redesign ML operations using basic gates, ensuring compatibility with OpenFHE and NuFHE.

3.2 CKKS

We consider **CKKS** [17] as the underlying library for approximate arithmetic operations on encrypted data. CKKS is supported by libraries like SEAL, HElib, PALISADE, and OpenFHE [43]. CKKS is designed for efficient evaluation of vectorized approximate computations on complex or real-valued data, making it suitable for encrypted machine learning where exact comparisons are unnecessary.

CKKS enables homomorphic addition, multiplication, and scalar operations on packed ciphertexts representing vectors of complex numbers. Since CKKS is an approximate scheme, polynomial or piecewise polynomial approximations are mandatory for non-arithmetic operations like comparisons or activation functions. Following, we explain the mathematical formulation of key algorithms in CKKS [17]:

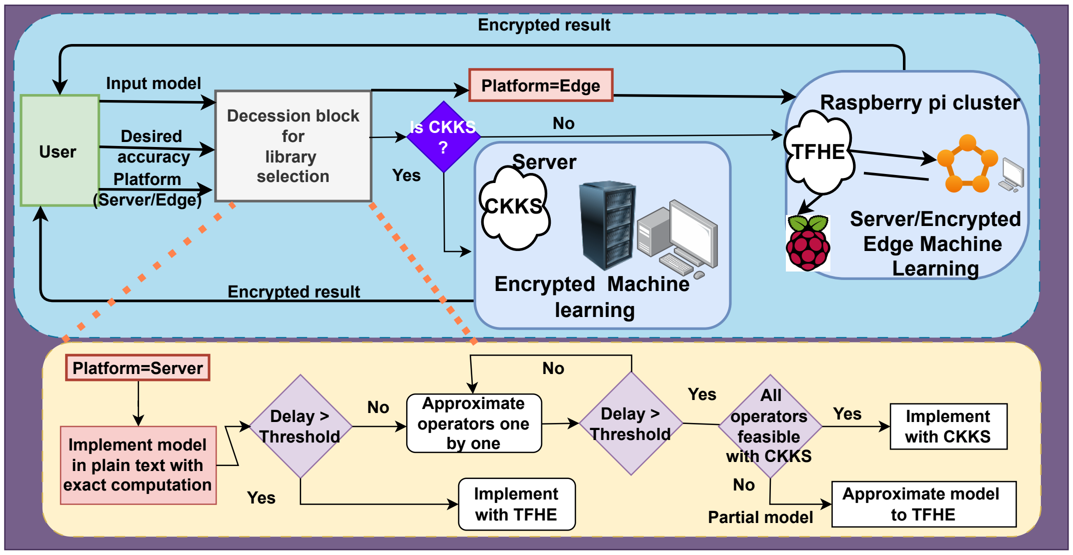


Fig. 1. FHEMaLe: Framework for Encrypted Machine Learning as Service

$\lambda = 128$ (security parameter)

$\mathbb{R}$ = real numbers, $\mathbb{C}$ = complex numbers

$\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$, q = ciphertext modulus

$\mathbb{Z}_q[X]/(X^N + 1)$ = ring of polynomials modulo cyclotomic polynomial

$\mathbb{C}^{N/2}$ = slots for packed complex numbers

- **KeyGen**(λ): Generates
  - Secret key $s \in \mathbb{Z}_q[X]/(X^N + 1)$ (from discrete Gaussian or ternary distribution)
  - Public key $pk = (b, a)$ where $a \leftarrow_R \mathbb{Z}_q[X]/(X^N + 1)$, $b = -as + e$
  - Relinearization keys and Galois keys for homomorphic operations
- **Enc**(m):
  - Map $m \in \mathbb{C}^{N/2}$ to polynomial $m'(X)$ via canonical embedding
  - Sample $a \leftarrow_R \mathbb{Z}_q[X]/(X^N + 1)$ and error e
  - Compute ciphertext:
$$c = (c_0, c_1) = (b + q \cdot \Delta m'(X), a)$$
- **Dec**(s, c):
  - Compute approximate message:
$$m'(X) \approx \frac{c_0 + c_1 s}{q}$$
  - Apply inverse canonical embedding to recover m
- **Eval**($f, c_1, \ldots, c_t$):
  - Supports homomorphic addition:
$$c_{\text{sum}} = c_1 + c_2$$
  - Supports homomorphic multiplication:
$$c_{\text{prod}} = \text{Relin}(c_1 \cdot c_2)$$
  - Rescale ciphertexts to manage scale growth:
$$\text{Rescale}(c) = \frac{c}{\text{scale factor}}$$

4 PROPOSED FRAMEWORK

In this section, we elaborate on the proposed framework FHEMaLe, which dynamically selects HE library based on user inputs and the characteristics of the machine learning model. First we detail how choice of library is important.

4.1 Choice of Library

We mostly compare two leading FHE libraries, **CKKS** [17] and **TFHE** [55], in terms of their suitability to be the baseline library on cloud and edge. As illustrated in Table 9, CKKS-based libraries such as *Microsoft SEAL*, *TenSEAL*, *HE Transformer*, and *HEAAN* offer support for approximate arithmetic, making them suitable for encrypted deep learning inference on real-valued data, an essential requirement in cloud-based workloads. Conversely, TFHE-based libraries like *Concrete*, *TFHE*, and *CUFHE* enable fast bit-level operations and efficient bootstrapping, with lower memory demands, rendering them ideal for logic-heavy or binarized computations common in resource-constrained edge devices. This contrast highlights the fundamental trade-offs: CKKS favors higher throughput and supports real-number computation with approximation, while TFHE focuses on precision and responsiveness in binary logic environments, critical for real-time edge scenarios.

- **Operator Support:** CKKS is an approximate arithmetic homomorphic encryption scheme that supports addition, multiplication, and slot rotations for parallel SIMD operations. However, it does not natively support bit-level operations such as comparisons, conditional operation (MUX), or sorting, which require evaluation of sign functions or Boolean logic. These absolute value operations require sign evaluation and conditional logic, essential for traditional ML implementation. Moreover, CKKS only supports approximate multiplications and additions, so division must be approximated using iterative methods like Newton-Raphson [62], which require deep polynomial circuits and large multiplicative depth. Multiplicative depth refers to the maximum number of consecutive homomorphic multiplication levels (or layers of multiplications) that a ciphertext can undergo in an encrypted computation before decryption fails due to noise overflow and increased multiplicative depth in turn increases computational latency and memory demands.
  In addition, comparisons, sorting, and exact non-linear functions cannot be efficiently packed, and each requires its own high-degree circuit. Consequently, ciphertext counts, overall memory consumption, and computational depth scale poorly in memory-limited edge environments. Thus, for edge-deployed systems such as a Raspberry Pi cluster with constrained RAM and computation, CKKS remains suitable only for matrix multiplications, dot-products, and basic linear activations. However, any conditional logic, branching, or precise non-linear operation forces approximation strategies that dramatically increase ciphertext size and circuit depth. For example, encrypted KNN and SVM require comparisons and sorting, and these are not implementable in native CKKS without expensive workarounds. LR involves divisions and absolute value operations, which are also outside the efficient scope of CKKS. Deep neural networks and LLMs often rely on non-linear operations like exponentials, logarithms, or square roots. CKKS lacks support for these directly, and polynomial or piecewise approximations are mandatory [8]. Thus, while neural networks and LLMs can be efficiently implemented via CKKS using SIMD packing with tolerable approximations, traditional ML models face steep memory and performance barriers under CKKS on the edge. To summarize, the CKKS scheme designed for approximate arithmetic, cannot directly handle several operations critical for our encrypted ML algorithms, limiting its applicability. These include:
  - **FHE_Comparison:** Required for KNN sorting, SVM, and LR predictions, as CKKS struggles with exact comparisons.
  - **FHE_Sorting:** Used in KNN for distance ordering, reliant on comparisons.
  - **FHE_Mux:** Used in SVM and LR for conditional selections, not natively supported.
  - **FHE_Division, FHE_Absolute:** Used in LR sigmoid approximation, requiring approximations in CKKS.
  - **FHE_Exponent, FHE_Log, FHE_SquareRoot:** Used in DNNs, requiring approximations that introduce errors. The CKKS scheme, efficient for approximate arithmetic on encrypted data, can impact ML model accuracy in high-precision scenarios.
- **Keysize and memory requirement:** In CKKS, a collection of distinct keys facilitates secure and efficient operations on encrypted data. The secret key (s) plays a critical role in decrypting ciphertexts and creating additional keys required by the system. The public key (p) is responsible for encrypting plaintext data, ensuring its confidentiality during computation. To maintain efficiency, relinearization keys are utilized to control the growth of ciphertexts following multiplication operations. Meanwhile, rotation keys, also referred to as Galois keys, enable cyclic shifts of the data packed in ciphertexts, supporting

SIMD (Single Instruction, Multiple Data) operations for parallel processing. CKKS key sizes vary widely, with multiplicative keys from 2.6 MB to 2.6 GB, public keys from 1.3 MB to 850 MB, and rotation keys for processing vary from 10.5 MB to 10 GB. At depth 150, the mult key reaches 2.6 GB, and the rotation key 10.2 GB, reflecting high computational demands. OpenFHE TFHE, however, has a stable key size, with a 113 MB public key and a 4 KB secret key, making it more efficient for edge computing. For OpenFHE TFHE with 128-bit security, Bootstrapping key size is also around 25 MB. This trade-off between computational complexity and memory efficiency highlights OpenFHE's advantage in edge environments, where reduced key sizes lower storage and processing demands.

The next section details the system and threat model of the proposed framework.

4.2 System and Threat model for Proposed Encrypted ML Processing Framework

The proposed framework as shown in Fig. 1 enables secure ML processing on multiple platforms. Before detailing the actual ML implementation steps, we mention the relevant system and threat model. In this framework, the system comprises three main entities: (1) the **Client** (e.g., sensor node), (2) the **Edge Nodes** (ENs), and (3) the **Master Node** (MN) or Cloud Server. The client generates cryptographic keys for homomorphic encryption and securely shares necessary keys (e.g., public, relinearization, and Galois keys) with the computational nodes. Encrypted data and computation requests are sent to edge nodes or the cloud server, ensuring data confidentiality during processing. **Edge system:** Sensor nodes collect data from the environment but are vulnerable to physical capture and interception. Moreover, sensor nodes are limited in processing power to perform complex ML processing. Hence, edge nodes process encrypted data near the source but requires encrypted transmission and encrypted processing. Particular high end nodes may act as the master node that coordinates tasks, stores encrypted data, and aggregates encrypted partial results, and is susceptible to unauthorized access or breaches. The FHEMaLe system mitigates these threats through IND-CPA-secure [22] FHE and secure task distribution.

Cloud system: In the cloud setting, encrypted machine learning allows clients to outsource both training and inference over encrypted data. The cloud infrastructure is assumed to be honest-but-curious, with threats including data breaches, model extraction, and insider access attempts. FHE ensures that all computations are performed on encrypted data, protecting the confidentiality of both the model and the data.

Cryptographically secure ensemble learning further strengthens privacy by distributing encrypted workloads across multiple models and aggregating encrypted predictions. This reduces the risk that attackers could infer sensitive information from any single model or computation [34, 23]. The proposed system dynamically selects the execution platform and underlying FHE scheme (e.g., CKKS or TFHE) based on user preferences and task requirements, ensuring a balance between accuracy, efficiency, and security.

4.3 Encrypted ML Processing in the Framework

To discuss ML processing, we start with TFHE as the underlying library as it supports exact computation and suitable for edge computing for low memory overhead. Gradually, we shall discuss about the feasibility of operators and possible optimizations related to CKKS implementation for faster implementation. It is to note that FHE ML implementations on single edge device degrades the performance heavily, hence direct adaptation of the FHE implementations as reported in [46] is not possible and ML implementations should be revisited for distributed framework. For distributed processing, encrypted datasets from node$_1$ (the initial recipient) to other nodes (Node$_2$, Node$_3$, ..., Node$_n$) for independent encrypted partial predictions. Since Node$_1$ communicates with all other nodes without inter-node communication, a star topology ensures scalability while minimizing complexity. Overlapping communication and computation further reduces overhead. The next section discusses the limitations of single-edge encrypted ML for distributed platforms.

Limitations of Single Edge Encrypted ML for Distributed Platforms

In distributed edge environments, implementing ML algorithms presents challenges due to data fragmentation and inconsistencies. For KNN, the issue lies in distributing and retrieving data points across nodes, which affects efficiency. In centralized systems, data is easily accessible, but in a distributed setup, these issues hinder performance. Similarly, SVM faces difficulties in kernel computation and model synchronization, as coordinating across multiple nodes adds overhead and potential delays. LR also struggles with parameter estimation and model convergence, as iterative optimization may be slowed by communication delays and network latency. Thus, revisiting the framework to support encrypted ML processing in a distributed manner is necessary. The following section outlines the implementation of this framework, which builds on basic encrypted operators to enable distributed encrypted models.

4.4 Encrypted SVM

The SVM algorithm is widely used for classification and regression tasks in domains like image classification, text classification, and bioinformatics. It works by identifying the optimal hyperplane that maximizes the margin between support vectors i.e. the closest data points to the decision boundary of different classes. This maximization enhances generalization, improving accuracy on unseen data. The space between the support vectors and the hyperplane is called the margin, and the hyperplane with the largest margin is termed the optimal hyperplane.

In this work, we consider a linear hyperplane, which can be expressed as $w^T x + b = 0$. where the vector w represents the normal to the hyperplane, defining the perpendicular direction, and b denotes the offset of the hyperplane from the origin along w. After training on the dataset, the learnt parameters w and b are used to determine the decision function, which assigns labels to test data based on their position relative to the hyperplane. In this implementation, the SVM prediction steps were performed on encrypted data in a distributed edge network, which are as follows:

- Encrypted decision function(DF) computation: $DF = w^T x + b$, where x is the input vector for encrypted test data.
- Class label prediction based on decision function (DF)

4.4.1 *Distributed Encrypted SVM.*

In this section, the steps of distributed encrypted SVM implementation are discussed. The prediction steps are distributed on n edge nodes as depicted in Fig. 2. On the client side, test data vector x $(x_1, x_2, \ldots, x_m)$, w $(w_1, w_2, \ldots, w_m)$ vector (model parameters) and bias (b) etc. are encrypted with the help of secret key and sent to master edge node ($Node_1$) along with cloud key, from there this data is distributed among n edge nodes for further encrypted prediction computations.

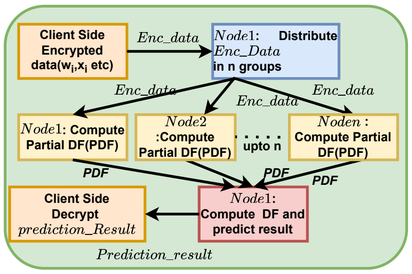

Client Side Encrypted data(w_i,x_i etc)
Enc_data
Node1: Distribute Enc_Data in n groups
Enc_data
Enc_data
Enc_data
Node1: Compute Partial DF(PDF)
Node2 :Compute Partial DF(PDF)
. . . . upto n
Noden : Compute Partial DF(PDF)
PDF
PDF
PDF
Client Side Decrypt prediction_Result
Node1: Compute DF and predict result
Prediction_result

Fig. 2. Encrypted Distributed SVM algorithm on Edge.

If we have m number of features in the input data and to find the decision function, we need to multiply w^T and x, and then b is added.

$$w^T x = [w_1 w_2 \cdots w_m]^T * [x_1 x_2 \cdots x_m] \tag{1}$$

$$w^T x = w_1 x_1 + w_2 x_2 + \ldots + w_m x_m \tag{2}$$

$$DF = w_1 x_1 + w_2 x_2 + \ldots + w_m x_m + b \tag{3}$$

$$DF = \underbrace{w_1 x_1 + \ldots + w_6 x_6}_{\text{PDF}_1} + \underbrace{w_7 x_7 + \ldots + w_m x_m + b}_{\text{PDF}_2} \tag{4}$$

$$DF = \underbrace{\sum_{i=1}^{k} w_i x_i}_{\text{PDF}_1} + \underbrace{\sum_{i=k+1}^{2k} w_i x_i}_{\text{PDF}_2} + \ldots + \underbrace{\sum_{i=(n-1)k+1}^{m} w_i x_i + b}_{\text{PDF}_n} \tag{5}$$

We have to perform encrypted multiplications and additions to compute decision function DF for encrypted data. These operations are distributed among edge nodes as partial decision function (PDF) computations to reduce computational overhead as shown in Fig 2. For example, for two nodes ($n = 2$) we can divide DF computation in PDF_1 and PDF_2 as illustrated in equation 4. If we have n edge nodes, then DF computation will be divided as shown in equation 5, where PDF_1, PDF_2,. . . ,PDF_n will be computed on nodes 1,2,...,n respectively as parallel processes reducing significant timing overhead. Once these PDF computations are completed

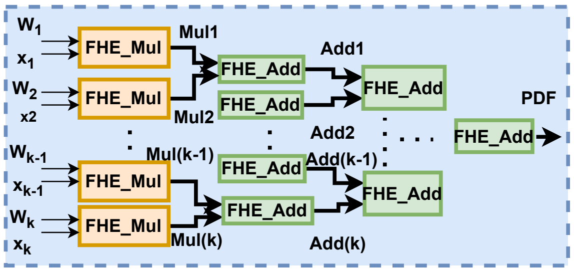


Fig. 3. Partial Decision Function Computation on Edge.

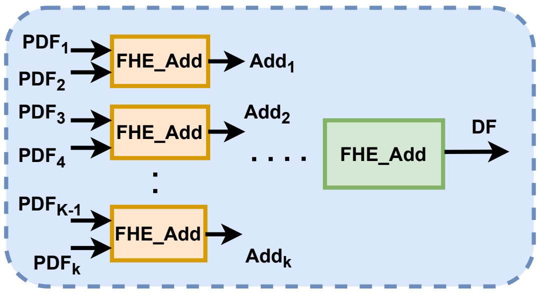


Fig. 4. Decision Function Computation on Edge.

on edge nodes, these partial results are sent to $Node_1$, and DF is computed after adding all $PDFs$. After which label prediction is performed on Node_1 and predicted results are sent to the client side, where the label is decrypted with the help of the secret key.

4.4.2 Encrypted Partial Decision Function Computation. As the first step in encrypted SVM prediction, DF is to be computed with distributed computations of partial decision functions (PDF) on n edge nodes. We are distributing model parameter vector (w) and test vector (x) on n edge nodes, where each node gets k elements of these vectors. For the computation of PDF ($PDF = w_1x_1 + w_2x_2 + \ldots + w_kx_k$), we need to perform k multiplications on encrypted data ($w_i * x_i$) using FHE_Multiplier circuit [10] and $(k-1)$ addition operations using FHE_Adder. Once the parallel computation of $PDFs$ is completed on n edge nodes, the results are sent to Node_1, where these partial results are accumulated to compute the final resultDF. These computations are performed as shown in Fig. 3.

4.4.3 Encrypted Decision Function Computation and Prediction. After getting $PDFs$ results for n individual nodes, DF is computed with the addition of all $PDFs$ using the FHE_Adder circuit as illustrated in Fig 4. Once the value of DF is calculated, the next task is to predict the label of the test data input. In our case we have two classes of labels, that is positive (+1) and negative (−1). With the help of the decision function (DF), we can tell which side of the hyperplane test data lies on. In our case, if the value of $DF > 0$, then the encrypted label is $Enc(+1)$ otherwise it is $Enc(-1)$. To perform this comparison on encrypted data, we have utilized FHE_Mux to decide the label according to its selection line. We have given the most significant bit of $DF(msb(DF))$ as the selection bit of FHE_Mux.

$$msb(DF) = \begin{cases} Enc(0) & \text{if } DF \geq 0 \\ Enc(1) & \text{if } DF < 0 \end{cases} \tag{6}$$

$$\text{FHE\_Mux}(result) = \begin{cases} Enc(+1) & \text{if } \text{msb}(DF) = Enc(0) \\ Enc(-1) & \text{if } \text{msb}(DF) = Enc(1) \end{cases} \tag{7}$$

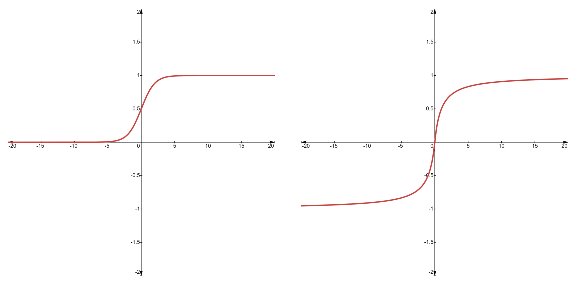
According to FHE_Mux ($result$), the encrypted label of test data input is predicted on the Node_1, this predicted label result is sent to the client side. The encrypted label is decrypted on the client side with the help of the secret key. Prediction time taken for SVM implementation on different numbers of edge nodes is discussed in the results section 8.

Feasibility and Optimizations for CKKS based Implementation of SVM

In a CKKS-based implementation of SVM, computations such as inner products between feature vectors and model weights, as well as bias addition, can be efficiently performed homomorphically using the scheme support for approximate arithmetic. However, challenges arise during the final classification stage where SVM requires exact comparisons (e.g., checking the sign of the decision function output to determine class labels). Since CKKS is inherently an approximate scheme, it cannot directly perform exact sign checks or threshold comparisons within the ciphertext space. This reliance on approximate arithmetic significantly impacts model performance. The accumulated noise and approximation errors from CKKS computations can cause the decision function output to deviate slightly from its true value. When these deviations occur near the classification boundary (i.e., values close to zero in a binary SVM), the final comparison can produce incorrect class labels. Hence, in this work we have taken pre-final encrypted ourcomes to client side and performed the final prediction step after decryption. Though this method is suitable for binary classification or multiclass classification with smaller number of classes, it adds added overhead as number of class increases. With increased processing and memory overhead, this may not be suitable for multiclass classification with large number of classes, specially on edge platform.

4.5 Encrypted LR

LR, a supervised learning method, is used for binary classification by estimating the likelihood of an outcome. It analyzes the relationship between independent variables and classifies data into distinct groups, such as yes/no or 0/1. The LR model employs a sigmoid function to map real-valued inputs to a probability range of 0 to 1, producing an S-shaped curve. If the predicted probability exceeds a predefined threshold, the instance is assigned to a class; otherwise, it is classified as not belonging to that class.



(a) Sigmoid (b) Approximate Sigmoid

Fig. 5. Comparison between Sigmoid and Approximate Sigmoid Function (plaintext).

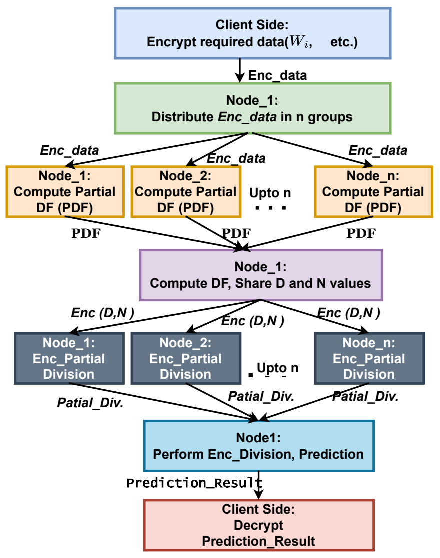


Fig. 6. Encrypted Distributed LR Framework on Edge.

In this implementation, LR prediction steps were performed on encrypted data in a distributed edge network, which are as follows:

- Encrypted decision function (DF) computation same as in SVM implementation discussed in section 4.4.2

$$DF = w^T x + b \tag{8}$$

  where x is the encrypted test data input vector, w is the model parameter vector and b is bias.
- Encrypted sigmoid calculation based on decision function (DF)
- Class label prediction based on sigmoid result

In the following few sections, we will discuss the encrypted prediction steps stated above in detail.

4.5.1 *Distributed Encrypted LR.*

In this section, we will discuss the distributed framework for the implementation of the encrypted LR algorithm. The block diagram shown in Fig. 6 shows the distributed prediction process on n edge nodes. On the client side, the test data vector x ($x_1, x_2,\dots, x_m$), model parameters w ($w_1, w_2, \dots, w_m$), and bias (b) are encrypted using a secret key and sent to Node$_1$ along with the cloud key. The node$_1$ then distributes these data among the edge nodes n for encrypted prediction.

The edge nodes compute the partial decision function, similar to the SVM framework, followed by the sigmoid function using an approximate method to handle encrypted data efficiently. Since division is computationally expensive in encryption, we propose an approximate division algorithm, which distributes operations across edge nodes n, with the final division result computed at node$_1$. The label prediction is then performed, and the encrypted result is sent back to the client for decryption and classification.

4.5.2 Encrypted PDF and DF Computation. After training on training dataset, the model parameters w and b are used to compute the decision function for classifying test data. The partial decision function (PDF) is computed concurrently on n edge nodes, following the approach discussed in Section 4.4.2 for SVM. Similarly, the encrypted decision function (DF) computation follows the method outlined in Section 4.4.3.

$$DF = \underbrace{\sum_{i=1}^{k} w_i x_i}_{\text{PDF\_1}} + \underbrace{\sum_{i=k+1}^{2k} w_i x_i}_{\text{PDF\_2}} + \ldots + \underbrace{\sum_{i=(n-1)k+1}^{m} w_i x_i + b}_{\text{PDF\_n}} \tag{9}$$

4.5.3 Encrypted Approximate Sigmoid Calculation. The next step after encrypted decision function (DF) computation is the evaluation of the sigmoid function. As sigmoid computation involves complex operations like exponentiation, which are challenging in an encrypted domain, we employ an approximate sigmoid function for efficient processing.

$$\text{sig}(DF) = \frac{1}{1 + e^{-DF}} \tag{10}$$

$$\text{ap\_sig}(DF) = \frac{DF}{1 + |DF|} \tag{11}$$

The sigmoid function ranges from 0 to 1 with a threshold of 0.5, while the approximate sigmoid function ranges from -1 to $+1$, using a threshold of 0 for classification (Fig. 5). Computing the approximate sigmoid, $ap\_sig(DF)$, requires an encrypted division of DF by $1 + |DF|$, a costly operation in encrypted domain [10]. To address this issue, we have proposed an approximate division algorithm, which is discussed in detail in the next section.

4.5.4 Proposed Encrypted Approximate Division for Single Node. In this section, we will discuss the computation steps of $ap\_sig(DF)$. As it requires encrypted division operation, we have proposed an approximate encrypted division Algorithm 1. The key idea of this approximation is that the denominator (D) is converted into its nearest power of two value. For example, 6 is converted into 8. Later, the numerator is right-shifted by the number of powers, which is equivalent to dividing by the power of two value.

Algorithm 1 Proposed Encrypted Approximate Division

```
1:  $N \leftarrow DF$                          # assign value to numerator
2:  $D \leftarrow 1 + |DF|$                    # assign value to denominator
3:  $PO2 \leftarrow [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, \ldots]$
4:  $EPO2 \leftarrow Enc(PO2)$                 # encrypted power of 2 array
5:  $abs\_sub[0] \leftarrow$ False
6:  $s[0] \leftarrow$ False
7:  for $i$ in $[1, \text{len(EPO2)}]$ do
8:      $sub[i] \leftarrow$ FHE_Subtraction$(EPO2[i], D, size)$
9:      $abs\_sub[i] \leftarrow$ FHE_Absolute$(sub[i])$
10:     $ssub[i] \leftarrow$ FHE_Subtraction$(abs\_sub[i-1], abs\_sub[i], size)$
11:     $s[i] \leftarrow sub[i][size-1]$
12:     $ss[i] \leftarrow ssub[i][size-1]$
13:     $sel \leftarrow s[i] + ss[i] \cdot s[i-1]$
14:     $rsft \leftarrow N >> 1$

    # Shifts if $D > EPO2[i]$ or if $D < EPO2[i]$ but closer than $EPO2[i-1]$
15:     $N \leftarrow$ FHE_Mux$(sel, rsft, N)$

16: end for
```

To implement these steps for encrypted data, we present Algorithm 1. The numerator and denominator of the ap_sig function are assigned to the variables N and D, respectively. An array of power-of-two values ($EPO2$) up to $2^{11} = 2048$ is created, encrypted on the client side, and sent to the edge nodes along with other Enc_data.

The next step is to find the nearest power of two to D from the $EPO2$ array. We iterate through the $EPO2$ array, and the numerator N is right-shifted by the index of the element $EPO2[i]$ that is closest to D. This right shift occurs if any of the following conditions hold:

- If D is greater than $EPO2[i]$

$$D > EPO2[i] \tag{12}$$

- If D is greater than $EPO2[i-1]$ and less than $EPO2[i]$ and $EPO2[i]$ is closer to D than $EPO2[i-1]$ (e.g: if $D = 14, |14-16| < |14-8|$)

$$\begin{gathered}(EPO2[i-1] < D < EPO2[i])\& \\ (|EPO2[i-1] - D| < |EPO2[i] - D|)\end{gathered} \tag{13}$$

To check these above two conditions, we will perform two FHE_Subtraction operations. First FHE_Subtraction ($sub[i]$) is between $EPO2[i]$ and D

$$sub[i] = EPO2[i] - D \tag{14}$$

The most significant bit ($s[i]$) of result14 of decides condition 12. If $s[i]$ is $true$ then 12 is $true$ and vice versa.

$$cond1 = s[i] \tag{15}$$

The Second FHE_Subtraction ($ssub[i]$) is between $abs\_sub[i-1]$ (absolute value of $(EPO2[i-1] - D)$) and $abs\_sub[i-1]$ (absolute values of $(EPO2[i] - D)$), whose most significant bit is $ss[i]$.

$$ssub[i] = abs\_sub[i-1] - abs\_sub[i] \tag{16}$$

The condition 13 will be $true$, if $ss[i]$ is true, $s[i]$ is false and $s[i-1]$ is true.

$$cond2 = ss[i] \cdot \overline{s[i]} \cdot s[i-1] \tag{17}$$

The numerator N is right-shifted by one if any one of the above conditions is true. In each iteration, we will either update N with right-shifted N ($rsft$) or keep it as it is (N). This selection choice is made with FHE_Mux where the selection line (sel) is given as follows:

$$sel = cond1 + cond2 \tag{18}$$

$$sel = s[i] + ss[i] \cdot \overline{s[i]} \cdot s[i-1] \tag{19}$$

$$sel = s[i] + ss[i] \cdot s[i-1] \tag{20}$$

After simplifying, sel input is given as shown in equation 20, where if sel is $true$, N is updated with $rsft$. Once the iterations are completed, the encrypted division result is obtained as updated N. In the next section, we will expand this algorithm for distributed edge network.

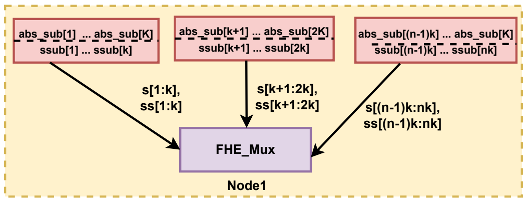


Fig. 7. Distributed Operations of Encrypted Approximate Division on Edge.

4.5.5 ***Proposed Encrypted Distributed Approximate Division.*** As explained earlier, the key operations for encrypted approximate division involve two types of FHE_Subtraction and FHE_Mux operations for updating N. To distribute the division workload, FHE_Subtraction operations are divided across n edge nodes, while the most significant bits (s, ss) are sent to Node$_1$, where FHE_Mux operations are performed. In Algorithm 1, lines [8-12] are distributed across the n edge nodes, as shown in Fig. 7, while lines [13-15] are executed on Node$_1$. Each edge node computes k elements of the s and ss arrays, which store the most significant bits of the FHE_Subtraction results. These bits are then used to calculate the sel bit of the FHE_Mux operation. After completing the approximate division, the result is assigned to the ap_sig function, which is used for predicting the test data label.

4.5.6 Encrypted Label Prediction. Once the approximate sigmoid (ap_sig) is computed, next the encrypted label is predicted. For prediction, the ap_sig value is compared with a threshold value that is $Enc(0)$ here. Since the ap_sig value varies from -1 to $+1$, if it is greater than the threshold value then the predicted label is $Enc(+1)$ otherwise $Enc(-1)$. For comparison, the most significant bit ($signbit$) of ap_sig is checked to determine if it is positive or negative since the threshold value is $Enc(0)$.

The encrypted label prediction is performed with the help of the FHE_Mux circuit, where $signbit$ is given as the selection line. The predicted label is given as below depending on $signbit$:

$$\text{Predicted label} = \begin{cases} \text{Enc}(-1) & \text{if signbit} = \text{Enc}(1) \\ \text{Enc}(+1) & \text{if signbit} = \text{Enc}(0) \end{cases}$$

This predicted label result is sent to the client side, where it is decrypted with the help of the secret key.

Feasibility and Optimizations for CKKS based Implementation of LR

In a CKKS based LR implementation, core operations such as dot products of feature vectors and model coefficients, plus bias addition, are performed efficiently using CKKS's approximate arithmetic operations. However, the sigmoid function, critical for mapping decision boundary output to probabilities, requires a polynomial or piecewise approximation [30] due to the limitation of CKKS to linear operations, which introduces error. This error arises at two levels: First, the polynomial approximation of the sigmoid and second, the final classification, where the approximate sigmoid output is compared to a threshold. As discussed previously in the CKKS based implementation of SVM, exact encrypted comparisons are infeasible, necessitating either decryption or additional polynomial approximations for the threshold comparison, both adding error. For samples near the decision boundary, small approximation errors can cause misclassification, producing false positives or false negatives. Post-decryption refinement is used to perform final comparison introduces security risks [8], [4]. When ciphertexts are decrypted to refine classification results, the plaintext outputs (e.g., probabilities or decision values) become accessible. An adversary with access to these decrypted results, such as through a compromised client or server, could exploit this information to infer sensitive details about the model or input data.

5 WHY KNN IS NOT SUITABLE IN ENCRYPTED DOMAIN?

The KNN algorithm is a supervised learning classifier recognized for its non-parametric approach. It classifies or predicts the grouping of an individual data point by utilizing the proximity of data points. In this method, a query data point is assigned a class label based on a plurality vote among its K (an integer) nearest neighbors, each representing a specific label. The primary steps in implementing the KNN algorithm are: (1) **Distance Computation**: calculating the distance between the test data points (T_i) and the training data points (Tr_i); (2) **Sorting Distances**: sorting the computed distances to identify the K nearest neighbors; (3) **Voting**: performing voting among the K nearest neighbors based on their class labels; and (4) **Class Label Assignment**: assigning a class label to the test data point (T_i) based on the plurality of votes. In the next few subsections, we discuss encrypted KNN implementation steps mentioned above. For that, we revisit the design of a few FHE operations (distance computation, encrypted sorting, etc.) to make them suitable for distributed platforms.

5.1 Encrypted KNN Algorithm for Parallel Processing Distributed Platform

This section introduces our specialized module designed to accelerate the KNN algorithm using distributed HE, as illustrated in Fig. 8 and Fig. 9. The framework facilitates encrypted prediction across distributed Raspberry Pi nodes. It consists of two main phases: initialization and prediction. During the initialization phase, the client generates a public-secret key pair. It encrypts the data using these keys and transfers the encrypted data to edge Node_1. Node_1 then partitions and distributes the data among the respective worker nodes. In the subsequent prediction phase, each worker node performs partial prediction steps using the encrypted data. After completing distributed processing, Node_1 aggregates and transmits these encrypted results back to the client. This process ensures that sensitive information remains protected during transmission and can be further processed or integrated into the overall model without compromising data confidentiality. To do this, basic FHE operations like distance computation and encrypted sorting need to be redesigned so that they work on distributed platforms. Before going to the actual implementation, we try to answer here the following pertinent question:

Why exisitng sorting [49] and [51] will not work in distributed scenario?

The core modules for implementing distributed encrypted KNN are primarily adapted from [49] and [51], which are developed in Python and operate using single-threaded distributed environments. Hence, we need to redesign sorting implementation, realisizing the distribution step is more crtical due to the inherent nature of sorting algorithims. It is to highlight that distribution of soritng steps are not straightforward and we cannot just take array of length n, divide the full data within d edge nodes and perform the soritng in parallel to get the final result. There should be suitalble merging step to get the top k values of the whole array. Here we detail how we are developing merging step to sort the outcome of distruibted distance compuation. In comparison to previous work,our work explores thread-level parallelism using OpenHFE in C++, enabling the implementation of KNN to leverage parallel processing capabilities. This approach allows for efficient sorting and computation across multiple threads, significantly improving execution time and making it more suitable for distributed or high-performance computing environments.

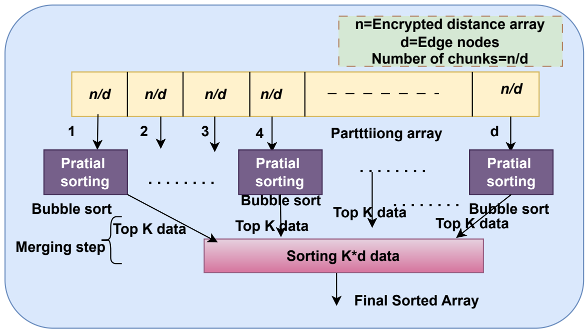


Fig. 8. Distributed Sorting on Edge Cluster.

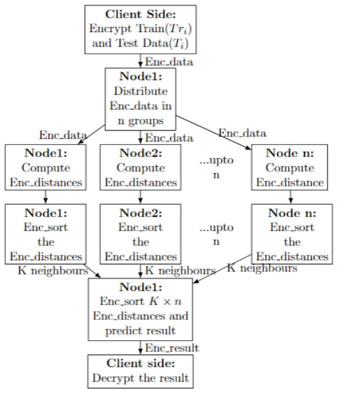


Fig. 9. Encrypted Distributed KNN algorithm

Distributed Distance Computation. To find the nearest data points to the test instance, the first step is to compute the distance from each train data point. Let us consider the train point vector as $Tr = \{C_1, C_2, C_3, \ldots, C_m\}$ and the test point vector as $T = \{Ct_1, Ct_2, Ct_3, \ldots, Ct_n\}$, where m and n are the number of features for train and test instances respectively and C_i, Ct_j indicate the feature instance of train data and test data respectively, where $i \in [1, m]$ and $j \in [1, n]$.

There are various types of distance available, such as Minkowski distance, Euclidean distance, Manhattan distance, Hamming distance, and Cosine distance. The Euclidean distance function is the most popular one among all of them, but since the multiplication operation is costly in the encrypted domain compared to other addition and subtraction operations, we consider the Manhattan distance here for the computation of the distance matrix. Manhattan distance is computed as follows:

$$Manhattan\ distance(Tr, T) = \sum_{i=1}^{m} |C_i - Ct_i|$$

In this work, we have distributed encrypted distance (Enc_distance) matrix computation operations across n number of Raspberry Pi edge devices. The encrypted train data points are split into n groups and sent over n edge nodes (Node$_1$, Node$_2$,...Node$_n$) along with the encrypted test data. For each group of test data, the Enc_distance submatrices $dist_1[i][j]$, $dist_2[i][j]$, ..., $dist_n[i][j]$ are calculated. Here, $dist_k[i][j]$ is the distance submatrix for node k, where i is the index of the Enc_train data and $j = 0$ shows the value of Enc_distance, and $j = 1$ shows the label of the Enc_train data.

For the computation of the Manhattan distance, two sub-operations are performed in the encrypted domain [50]: FHE subtraction (FHE_Subtraction) and encrypted absolute value computation of the FHE_Subtraction result. In case the FHE_Subtraction result is negative, to get the absolute value, we need to take the two's complement of the result.

To compute the two's complement in the encrypted domain, first, all bits are inverted for the encrypted data, and then *Enc(1)* is added to get the absolute value of the encrypted data [10].

Since it is important to check if the FHE_Subtraction result is positive or negative, this encrypted decision-making is done using FHE multiplexer (FHE_Mux) [50], where the most significant bit (MSB or sign bit (*sbit*)) of the subtraction result is given as the selection line. If the FHE_Subtraction result is positive, then the *sbit* is *Enc(0)*, selecting the FHE_Mux result as $(C_i - Ct_i)$. Otherwise, if the FHE_Subtraction result is negative, then the *sbit* is *Enc(1)*, selecting the FHE_Mux result as $-(C_i - Ct_i)$.

After computation of absolute values of encrypted distance for individual features, these distances are added with FHE_Adder [50] circuit to compute the final Enc_distance between two encrypted data points. Final computed Enc_distance values are stored in distance submatrices ($dist_k[i][j]$) of each node. Further, encrypted sorting is performed on these distance submatrices to find out K nearest neighbours from each distributed dataset.

Algorithm 2 Encrypted Sorting on Distributed Platform

```
    # Compute Enc_distances submatrices in
    $Node_1$,Node2,...Node n

    # Enc_sort on $Node_1$,Node2,...,Node n:
 1: for $k \leftarrow 0$ to $n$ do
 2:   for $i \leftarrow 0$ to len($dist_k$) do
 3:     for $j \leftarrow 0$ to len($dist_k$) $- i - 1$ do
    # create temporary variable v1,v2
 4:       $v1[0] \leftarrow dist_k[j][0]$
 5:       $v1[1] \leftarrow dist_k[j][1]$
 6:       $v2[0] \leftarrow dist_k[j+1][0]$
 7:       $v2[1] \leftarrow dist_k[j+1][1]$
 8:       $temp \leftarrow$ FHE_Subtraction($v1[0], v2[0], size$)
 9:       $sbit \leftarrow temp[size-1]$
10:       $snbit \leftarrow$ vm.gate_not($sbit$)
11:       $dist_k[j][0] \leftarrow$ FHE_Mux($sbit, v1[0], v2[0]$)
12:       $dist_k[j+1][0] \leftarrow$ FHE_Mux($snbit, v1[0], v2[0]$)
13:       $dist_k[j][1] \leftarrow$ FHE_Mux($sbit, v1[1], v2[1]$)
14:       $dist_k[j+1][1] \leftarrow$ FHE_Mux($snbit, v1[1], v2[1]$)
15:     end for
16:   end for
17: end for

    #get smallest $k$ (number of neighbours) elements from all $n$ nodes and store in $Dist[i][j]$ matrix
18: for $i \leftarrow 0$ to $n$ do
19:   for $i' \leftarrow 0$ to $K-1$ do
20:     $Dist[i \times K + i'][0] \leftarrow dist_i[i'][0]$
21:     $Dist[i \times K + i'][1] \leftarrow dist_i[i'][1]$
22:   end for
23: end for

    #Enc_sort $k \times n$ neighbours in Dist matrix with the same algorithm
24: for $i \leftarrow 0$ to len($Dist$) do
25:   for $j \leftarrow 0$ to len($Dist$) $- i - 1$ do
    # create temporary variable v1,v2
26:     $v1[0] \leftarrow Dist[j][0]$
27:     $v1[1] \leftarrow Dist[j][1]$
28:     $v2[0] \leftarrow Dist[j+1][0]$
29:     $v2[1] \leftarrow Dist[j+1][1]$
30:     $temp \leftarrow$ FHE_Subtraction($v1[0], v2[0], size$)
31:     $sbit \leftarrow temp[size-1]$
32:     $snbit \leftarrow$ vm.gate_not($sbit$)
33:     $Dist[j][0] \leftarrow$ FHE_Mux($sbit, v1[0], v2[0]$)
34:     $Dist[j+1][0] \leftarrow$ FHE_Mux($snbit, v1[0], v2[0]$)
35:     $Dist[j][1] \leftarrow$ FHE_Mux($sbit, v1[1], v2[1]$)
36:     $Dist[j+1][1] \leftarrow$ FHE_Mux($snbit, v1[1], v2[1]$)
37:   end for
38: end for
```

Sorting on Encrypted Data

In this section, we shall describe the aspects of sorting on encrypted data. The feasibility of partition-based sorting on encrypted data has been discussed in [9] and from that results it is evident that simple comparison-based sorting algorithms, such as Bubble Sort with order of $O(n^2)$ complexity is a better choice compared to divide and conquer based sortings. We also analyze the feasibility of implementing bucket sort on FHE data and then explain our proposed schemes for actually implementing bubble sort on encrypted data. The bucket sort algorithm involves three steps: distributing input array values into buckets, sorting each bucket individually, and concatenating all sorted buckets.

$$index \leftarrow \lfloor \frac{(arr[i] - min\_value) \times bucket\_count}{(max\_value - min\_value + 1)} \rfloor$$

, for input array arr. From the above line it becomes apparent that index computation is not feasible because the array elements are encrypted while other values required for index computation are unencrypted. In this scenario, computing the index to partition the input array into buckets for further sorting is impossible. Given this limitation, we are left with only one feasible sorting operation, which is bubble sort.

After Enc_distance computation, the Enc_distance subarrays $dist_k[i][j]$ are obtained for each edge node i, where i indiacte edge node and j indiacte data with in the array. Bubble sort is applied to each encrypted distance subarrays, where two consecutive elements are compared and swapped if the first element is greater. To compare encrypted data, we use FHE_Subtraction and the sign bit ($sbit$) ($Enc(0)$ or $Enc(1)$) of the result is used to check which data is greater, then FHE_Mux circuit is used to store data in a sorted manner, using the $sbit$ as a selection line, as shown in sorting Algorithm 2 in lines$[2-16]$. This process runs concurrently in n

edge devices, reducing significant timing overhead. To decide K nearest neighbors in the entire encrypted dataset, K top vlaues from each n edge nodes are collected in and stored in $Dist[i][j]$ array at Node1 and sorted again with bubble sort to obtain the final K nearest neighbors (K top values) as shown from line 24, in sorting Algorithm 2 (also refer Fig. 8). These final K neighbours will be used for voting. After getting K nearest neighbours, the next step is plurality voting based on class labels of the neighbours and the majority voted label is assigned to test data input. Here train data labels (L_i) are taken as +1 (positive class) and -1 (negative class) in plaintext. To assign plurality-voted label to test data input, we need to check which label count is greater than the threshold value (half of the number of neighbours($K/2$)). To perform this, we add MSBs (most significant bits) of the labels for K neighbours using FHE_Adder circuit [50], which will be *Enc(0)* for +1 label and *Enc(1)* for -1 label. If positive class labels (+1) are more than negative class labels (-1), then the FHE_Adder result will be less than $K/2$ and vice versa. This FHE_Adder result is compared with the threshold value ($K/2$) with the help of the FHE_Subtraction circuit and the MSB of FHE_Subtraction is used as the selection line of FHE_Mux to predict the label of test data.

Table 2. Operations and CKKS Limitations

Algorithm	Operations Used	Operations Not Realizable with CKKS
KNN	FHE_Addition, FHE_Subtraction, FHE_Multiplication, FHE_Comparison, FHE_Sorting, FHE_Mux	FHE_Comparison, FHE_Sorting, FHE_Mux
SVM	FHE_Multiplication, FHE_Addition, FHE_Comparison, FHE_Mux	FHE_Comparison, FHE_Mux
LR	FHE_Multiplication, FHE_Addition, FHE_Absolute, FHE_Division, FHE_Comparison, FHE_Mux, FHE_Subtraction	FHE_Comparison, FHE_Division, FHE_Absolute, FHE_Mux
DNNs	FHE_Addition, FHE_Multiplication, FHE_Exponent, FHE_Log, FHE_SquareRoot	FHE_Exponent, FHE_Log, FHE_SquareRoot

However, incorporating application-specific values of K, this costly encrypted sorting step can be replaced with a linear computation of K -max. That in turn can improve the overall timing performance.

Feasibility and Optimizations for CKKS based Implementation of KNN

In a CKKS based KNN implementation, query vectors and datasets are encoded as ciphertexts, enabling parallel distance calculations between the query and multiple database points using SIMD operations. A significant challenge arises in selecting the k nearest neighbors, as CKKS does not support exact comparison or sorting of encrypted data. Approximate techniques, proposed in [13] implemented to perform comparison operations with polynomial approximations and a series of modular arithmetic operations to perform comparisons on encrypted numerical values without decryption. These methods often lack the efficiency and precision of plaintext KNN. The final step of identifying the k nearest distances and determining the majority class is particularly susceptible to errors. When distances between the query and training points are similar, small numerical inaccuracies in CKKS computations can lead to incorrect neighbor rankings, resulting in the selection of incorrect neighbors. This can cause false positives (assigning an incorrect class due to irrelevant points being selected) or false negatives (missing the correct class by excluding true nearest neighbors). As observed in CKKS based SVM and LR, post-decryption refinement, such as decrypting distances for plaintext sorting, risks information leakage. An adversary accessing decrypted distances could launch membership inference or reconstruction attacks, compromising data or model privacy. Thus, while CKKS enables homomorphic distance computations for KNN, secure and efficient neighbor selection remains a critical challenge.

6 WHY NN AND LLM CAN BE BETTER REALIZED IN CKKS BUT NOT GENRAL ML?

The basic components of NN and LLMs typically involve matrix multiplications (e.g., linear layers, attention heads), element-wise operations (e.g., activations like ReLU, GELU, sigmoid), normalization (e.g., layer norm, batch norm), and pooling or softmax layers. When attempting to implement these components under FHE, particularly using TFHE, significant challenges arise. While

TFHE is highly effective for bit-level operations like exact comparisons and logical functions, it is poorly suited for the large-scale arithmetic and lack of SIMD support that neural networks and LLMs depend on. Essential tasks such as matrix multiplications, dot products, and non-linear activations require high-precision computations that become inefficient with TFHE and substantial computational overhead, making these operations impractical in real-world scenarios. In contrast, schemes like CKKS, designed for efficient approximate arithmetic, are far better suited to these workloads.

CKKS natively supports matrix-vector and matrix-matrix operations, dot products, and scalar multiplications in an encrypted domain using SIMD packing of complex numbers, making it highly effective for homomorphically evaluating the linear algebraic core of these models [17]. The vectorized processing of CKKS enables parallel evaluation of multiple slots, further improving efficiency for such operations. However, operations involving exact comparisons (e.g., argmax in softmax, top-k selection (in case of probability token selection, ranking, recommendation systems, and text generation) , or hard thresholding in pruning), discrete branching, and sorting (in case of top-k recommendations, SoftRank approximate sorting by predicting smooth probabilistic scores, and retrieval-augmented generation) are not directly supported in CKKS. These operations require either switching to an exact scheme (such as TFHE) or designing polynomial or piecewise polynomial approximations [5]. While certain activations (e.g., ReLU, sigmoid) can be reasonably approximated using low-degree polynomials (e.g., Chebyshev or minimax approximations), not all operators lend themselves to accurate polynomial approximation. Functions involving discontinuities or sharp decision boundaries, like the argmax in softmax or sorting-based operations, are especially challenging to approximate accurately without significant error accumulation. The accuracy drop due to such approximations depends on the complexity of the function and the degree of the polynomial used. For example, approximate softmax using low-degree polynomials can lead to non-trivial degradation in model confidence and calibration, as reported in [8] and [5]. Studies have shown that for image classifiers and small transformers, accuracy drops of 1–5% are typical when activation and normalization functions are approximated [15].

Comparison operations are particularly challenging in FHE systems because, unlike basic operations such as addition and multiplication, they are not natively supported by CKKS FHE schemes. We have implemented the method proposed in [13] that uses polynomial composition to approximate comparisons. The idea is to approximate the sign function (which determines whether one number is greater than, equal to, or less than another) through repeated compositions of specially designed polynomials that converge to +1 or -1 as needed. This method involves selecting a base polynomial that satisfies specific properties (odd symmetry, convergence at +1 or -1, and particular curvature) and composing it multiple times to approximate the sign function. The more compositions, the better the approximation, but also the higher the computational cost. An improved variant combines two different polynomials in sequence to reduce the required number of compositions, making the comparison faster while maintaining accuracy.

A major challenge in applying CKKS to edge environments is memory consumption. CKKS ciphertexts are substantially larger than plaintext equivalents due to the packing of complex vectors, multiple ciphertext components, and large modulus sizes required for maintaining precision and security. For instance, a single encrypted vector under CKKS can require memory in the order of tens to hundreds of megabytes depending on the ring dimension, scale, and multiplicative depth [5]. This makes deployment on resource-constrained edge devices difficult. As an example, TFHE can represent a single encrypted bit using a few kilobytes (e.g., 8.9 KB), whereas CKKS ciphertexts for approximate encrypted vectors can demand over 100 MB per vector at moderate depth levels, as shown in OpenFHE benchmarks. This memory overhead significantly limits the feasibility of CKKS in edge scenarios without specialized hardware acceleration or hybrid schemes that offload intensive computations to more capable nodes. Designing memory-efficient CKKS implementations for edge environments remains an open challenge, with ongoing research exploring compression, ciphertext pruning, and streaming techniques to address this gap.

7 SECURITY ANALYSIS AND DISCUSSION

Securing ML models on resource-constrained edge devices is challenging. While encryption protects data at rest and in transit, it often renders data unusable for ML, exposing vulnerabilities. Edge models face adversarial attacks that manipulate input data, model extraction attacks targeting intellectual property, and privacy breaches exploiting training data. HE mitigates these risks by enabling encrypted computations, preventing adversarial manipulation, limiting model extraction, and preserving training data privacy.

In adversarial attacks like a CPA, an adversary $\mathcal{A}$ modifies input x to mislead an ML model $\mathcal{M}$. Without encryption, $\mathcal{A}$ finds x' such that $\mathcal{M}(x') \neq \mathcal{M}(x)$. With HE, $\mathcal{A}$ only sees encrypted inputs $E(x)$ and $E(x')$, preventing such manipulation. In model extraction, $\mathcal{A}$ queries $\mathcal{M}$ to approximate it. Without encryption, observed outputs $\mathcal{M}(x_1), \mathcal{M}(x_2), \dots$ reveal model details. With

HE, only encrypted outputs $E(\mathcal{M}(x_1)), E(\mathcal{M}(x_2)), \ldots$ are visible, obstructing model reconstruction. Privacy attacks analyze $\mathcal{M}$ to extract training data D_{train}, but HE ensures computations occur on encrypted data, preventing direct access to D_{train}.

HE supports encrypted operations where $E(x_1) \odot E(x_2) = E(x_1 \circ x_2)$. Given an ML model $\mathcal{M}$, the encrypted input $E(x)$ produces $E(\mathcal{M}(x))$ after homomorphic computation, maintaining confidentiality. This secures against adversarial attacks, as $\mathcal{A}$ only sees $E(x)$; prevents model extraction, as $\mathcal{A}$ accesses only $E(\mathcal{M}(x))$; and protects training data, keeping D_{train} encrypted as $E(D_{\text{train}})$. HE mitigates various threats on edge devices. Data breaches are prevented by keeping data encrypted during computation, ensuring unauthorized access is infeasible. MitM attacks are countered by encrypted data transmission, represented as $E(T)$. Command injection is mitigated by encrypting queries, $E(C)$, preventing tampering. Data integrity is preserved as encrypted computations maintain the original data's structure, $E(D_1) \cdot E(D_2) = E(D_1 \cdot D_2)$. Credential theft is thwarted by storing credentials in encrypted form, $E(C)$, and data leakage is prevented by ensuring computations do not expose plaintext information.

Economic Viability via TCO Analysis

The framework is economically viable, as demonstrated by a Total Cost of Ownership (TCO) analysis comparing its low-cost edge cluster to traditional cloud-based FHE solutions.

The document highlights that a Raspberry Pi 4 cluster (4–8 nodes, 8GB RAM each, interconnected via Gigabit Ethernet) costs approximately \$500–\$1000 [21], [45]. In contrast, a cloud server instance like AWS c6i.16xlarge, which processes FHE operations 10–20 times faster, incurs significantly higher costs. A typical AWS c6i.16xlarge instance costs approximately \$3.06 per hour (based on public AWS pricing as of 2023), translating to \$26,827 annually for continuous operation (8760 hours) [3]. For a 5-year TCO, the cloud solution costs approximately \$134,135, excluding data transfer and storage fees. The Raspberry Pi cluster's TCO includes initial hardware costs (\$1000), power consumption (approximately 5W per node, 40W total for 8 nodes, at \$0.15/kWh, yielding \$52.56/year or \$262.80 over 5 years), and minimal maintenance (estimated at \$100/year or \$500 over 5 years), totaling approximately \$1762.80. While cloud servers offer superior performance, the FHEMaLe framework targets applications like medical predictions where inference times of minutes to hours are acceptable. The TCO ratio (cloud:edge $\approx 76:1$) demonstrates that the edge cluster is economically viable for privacy-sensitive, non-real-time applications, leveraging low-cost hardware to achieve comparable functionality at a fraction of the cost.

Scalability for Practical Workloads

The framework demonstrates scalability for practical workloads by distributing FHE-based ML computations across multiple edge nodes, reducing inference time as node count increases.

Scalability is achieved by parallelizing FHE tasks across a cluster of edge nodes, as described in the document's star topology, where Node$_1$ distributes encrypted data to worker nodes for independent processing. Table 4 shows that distributing six NAND gate operations across three nodes reduces computation time from 65.7 seconds (one node) to 22.14 seconds (NuFHE), demonstrating near-linear speedup ($65.7/22.14 \approx 2.97$ for $n = 3$). For practical ML workloads, the framework supports KNN (37 minutes, 11 nodes), SVM (4.15 minutes, 11 nodes), LR (7.82 minutes, 11 nodes), and DNN (33.8 minutes, 8 nodes) on the Wisconsin Breast Cancer dataset (568 instances, 11 features). The increasing node count enhances efficiency and fault tolerance, handling larger workloads. The Fabric API facilitates task distribution and data sharing, minimizing communication overhead. For larger datasets or more complex models, adding nodes further reduces inference time, as the star topology avoids inter-node communication bottlenecks. This scalability is practical for non-real-time applications like medical diagnostics, where the dataset size and model complexity align with the tested workloads.

8 RESULTS

Our encrypted machine learning framework operates in both cloud and edge environments. In the cloud environment, encrypted computations are performed on an HP Z240 Tower Workstation equipped with an Intel Xeon E3-1240 v5 3.50 GHz processor and 64 GB of RAM, running Ubuntu 20.04 LTS. In the edge environment, encrypted operations are distributed across a cluster of Raspberry Pi 4 Model B devices (each with a Broadcom BCM2711, quad-core Cortex-A72 @ 1.5 GHz processor and 8 GB of RAM), connected in a star topology to a central master node. A single client system, configured with an Intel Core™ i7-8700 CPU running Ubuntu 22.04.3 LTS, is responsible for encrypting data before transmission and decrypting results received from either the cloud server or the edge cluster. We leverage the Fabric API for encrypted data sharing and parallel execution, distributing tasks

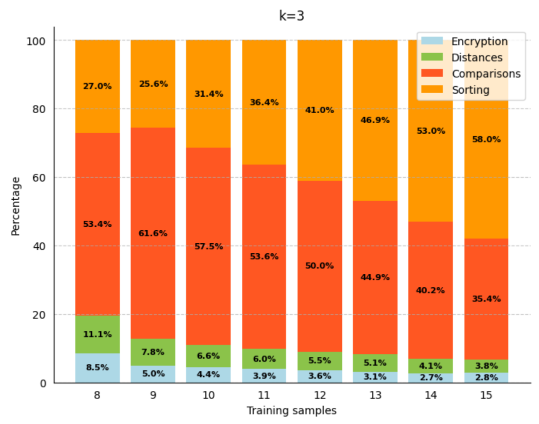


Fig. 10. Encrypted KNN using CKKS on Cloud

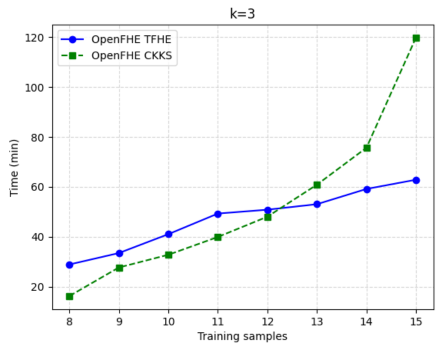


Fig. 11. CKKS vs TFHE KNN computaion time on Cloud

Fig. 12. Choice of library in the encrypted domain on Cloud.

Table 3. CKKS vs TFHE Time (single prediction) and Memory per element (one element 16 bit) on Cloud.

ML model	**OpenFHE CKKS time (sec)**	**OpenFHE TFHE time (sec)**	**OpenFHE TFHE Memory (MB)**	**OpenFHE CKKS Memory (MB)**
KNN (Training data=4, Test data=1)	248	780	0.064	107
KNN(Training data=15, Test data=1)	7182	3768	0.064	107
SVM	18	946	0.064	107
LR	17	650	0.064	107
NN	57	3982	0.064	107

across multiple Raspberry Pi nodes to optimize computation. In this work, we use the standard Wisconsin dataset (568 instances, 11 features) [19] from the UCI ML repository for binary classification. Table 3 presents the comparison of CKKS and TFHE schemes (using OpenFHE) for various machine learning models on the server-side, highlighting both execution time for a single encrypted prediction and memory consumption per 16-bit encrypted element. The results indicate that CKKS significantly outperforms TFHE in computation time for models relying heavily on linear algebraic operations, such as SVM (18 sec CKKS vs. 946 sec TFHE) and LR (17 sec CKKS vs. 650 sec TFHE). This advantage stems from CKKS's ability to perform SIMD-style packed approximate arithmetic efficiently, making it a better choice for models that do not require bitwise precision. In contrast, for models with more complex comparison or selection logic, such as KNN with 15 training samples, TFHE exhibits lower execution time (3768 sec) compared to CKKS (7182 sec). This is because TFHE's native support for exact comparison operations provides better scalability for KNN's distance-based selection, where CKKS struggles due to the need for polynomial approximations of such operations. The comparison of CKKS and TFHE encrypted KNN implementations for cloud environments, focusing on execution times as shown in Figure 10 and Figure 11. For training samples from 8 to 15 (k=3), CKKS is faster up to 12 samples (16.2–48 min) compared to TFHE (28.85–50.8 min), but TFHE outperforms beyond 12 samples, with CKKS reaching 119.7 min at 15 samples versus TFHE's 62.8 min. CKKS's execution time breakdown shows comparisons (569–2613 sec) and sorting (288–4283 sec) dominate at higher samples, unlike encryption (91–210 sec) and distances (118–283 sec) and the memory vs computation trade-off, as shown in Table 3.

Memory consumption analysis reveals another critical distinction. CKKS requires about 107 MB per encrypted 16-bit element, irrespective of model size, reflecting the overhead from large ciphertext components and scale management in approximate arithmetic. In contrast, TFHE's memory usage is minimal (0.064 MB per element), making it far more suitable for edge computing environments, where memory and resource constraints are critical. Therefore, while CKKS offers superior performance for linear models and neural networks on high-resource servers, its heavy memory footprint limits its practicality for edge deployments. On the other hand, TFHE, despite its higher computational cost in many cases, aligns better with the constraints of edge platforms (such as Raspberry Pi clusters) due to its compact ciphertexts and native support for exact logical operations. Hence, in the following discussion we only include edge results implemented using different variants of TFHE libraries like NuFHE [41], OpenFHE python [43]. To enhance timing performance, we preload training data onto edge devices, reducing latency and enabling faster query responses. This improves real-time processing by minimizing computational overhead and ensuring better load balancing and fault tolerance.

Table 4. Six Encrypted NAND Gate Operations Distribution on Edge Nodes.

$Node_1$	$Node_2$	$Node_3$	**NuFHE (Time)**	**OpenFHE Python (Time)**
6	0	0	65.7 sec	44.68 sec
4	2	0	42.84 sec	29.82 sec
3	3	0	34.59 sec	22.38 sec
2	2	2	22.14 sec	14.91 sec

Table 5. Encrypted Arithmetic Operations on Edge Devices (16- bit data).

Operation	**NuFHE (Time)**	**OpenFHE Python (Time)**	**OpenFHE OpenMP C++ (Time)**
Addition	56.79 sec	32.31 sec	11 sec
Subtraction	43.40 sec	39.31 sec	21 sec
Multiplication	32.46 min	23.89 min	170 sec

Results on Edge

Before implementing encrypted ML modules on edge, first, we distributed basic gate logic operations among edge nodes to observe the timing using FHE primitive gates provided by NuFHE and OpenFHE library. We have distributed 6 NAND gate operations, and the timing overhead is reduced significantly after distributing among edge nodes as shown in Table 4. Certain mathematical operations, including addition (16-bit), subtraction (16-bit), and multiplication (16-bit), were also analyzed after being translated in their encrypted form on a single pi board (Table 5).

Following the successful evaluation of the basic gate-level performance, the subsequent step involved implementing the encrypted KNN algorithm on a distributed edge network. The experiment was carried out using varying numbers of edge nodes and different standard neighbor values K, specifically 3, 5 and 7. Along with NuFHE, in this work we have also explored the OpenFHE [43] library for fully homomorphic operations which is way faster. The comparison results for NuFHE and OpenFHE framework for distributed encrypted KNN computation are presented in Table 6. However, in our Raspberry Pi board only 4 cores are present and OpenFHE with the improved bootstrapping works better in this scenario. After the successful implementation of encrypted KNN in distributed environment, the next encrypted SVM prediction steps were performed on varying numbers of edge nodes. The prediction time results are tabulated in the Table 7 a, and the library performances are compared as shown in Fig. 13.

Table 6. Distributed Encrypted KNN prediction time for K = 3, 5, 7 on Edge.

No. of Edge Nodes (n)	**NuFHE (Time)**	**OpenFHE (Time)**	**NuFHE (Time)**	**OpenFHE (Time)**	**NuFHE (Time)**	**OpenFHE (Time)**	**OpenFHE OpenMP C++ (Time)**
	K=3		K=5		K=7		K=3
1	11.06 hr	7.31 hr	11.12 hr	7.38 hr	11.16 hr	7.54 hr	3.47 hr
2	5.54 hr	3.66 hr	5.55 hr	3.71 hr	5.57 hr	3.77 hr	1.98 hr
3	3.67 hr	2.43 hr	3.70 hr	2.48 hr	3.72 hr	2.50 hr	1.17 hr
8	114 min	74 min	116 min	78 min	118 min	80 min	37.06 min

Table 7. Encrypted prediction time on Edge.

a. Distributed Encrypted SVM prediction time

No. of Edge Nodes (n)	**NuFHE (Time)**	**OpenFHE Python (Time)**	**OpenFHE OpenMP C++ (Time)**
1	7.18 hr	4.70 hr	35.1 min
2	3.15 hr	2.06 hr	19.01 min
3	2.30 hr	1.50 hr	12.55 min
11	34 min	18 min	4.15 min

b. Distributed Encrypted LR prediction time.

No. of Edge Nodes (n)	**NuFHE (Time)**	**OpenFHE Python (Time)**	**OpenFHE OpenMP C++ (Time)**
1	10.41 hr	5.39 hr	47.41 min
2	4.71 hr	2.43 hr	26.8 min
3	3.14 hr	1.62 hr	19.55 min
11	56 min	28.23 min	7.82 min

Table 8. Encrypted NN on Edge Device.

Model	Nodes	No. of layers	NuFHE (Time)	OpenFHE OpenMP C++ (Time)
NN	8	3	310 min	33.83 min

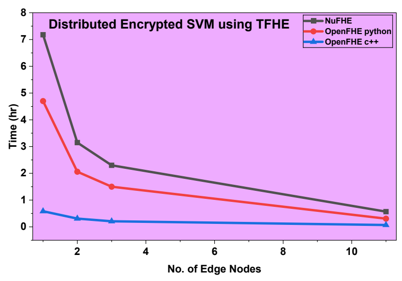


Fig. 13. Encrypted SVM prediction time on Edge

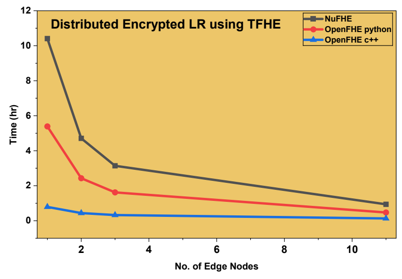


Fig. 14. Encrypted LR prediction time on Edge

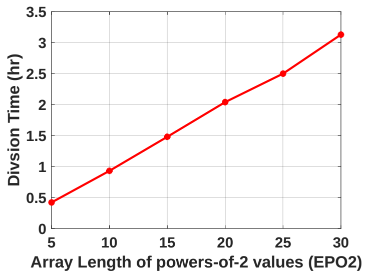


Fig. 15. Encrypted Division time w.r.t. length of *EPO2*

Table 9. Comparison of Popular FHE Libraries Supporting Encrypted Machine Learning

Framework / Library	Encryption Scheme(s)	Encrypted ML Support Type	Key Features	Operation Limitations
Microsoft SEAL	BFV, CKKS	Linear models, simple NN	High-performance CKKS; large-scale support; widely adopted	CKKS limitations: No native support for FHE_Comparison, FHE_Mux, FHE_Division; approximation needed for FHE_Log, FHE_Exp
HElib	BGV	Linear regression, LR	Smart noise management; modular arithmetic	No support for approximate arithmetic; not suitable for real-valued DNNs
PALISADE	BFV, BGV, CKKS	LR, CNN, SVM	Multi-scheme APIs; good documentation	CKKS: no exact comparison or Mux; lacks TFHE-style bootstrapping
Lattigo	BFV, CKKS	LR, deep learning (via APIs)	Go implementation; cloud-native focus	CKKS-only; lacks logic ops; no FHE_Comparison or Mux support
Concrete	TFHE, FHEW	Binarized Neural Networks (BNNs)	Efficient bootstrapping; optimized for real-time	No support for real-valued ops; not suitable for CKKS-based DNNs
TFHE	TFHE	Logic gates, binary ops	Bit-level accuracy; efficient bootstrapping	Cannot process real numbers; unsuitable for CKKS-style operations
TenSEAL	CKKS, BFV	PyTorch encrypted inference	Python interface; tensor support	CKKS lacks FHE_Comparison, FHE_Mux, FHE_Division; no training support
HE Transformer (Intel)	CKKS	ONNX-compatible DNNs	nGraph and OpenVINO integration	No support for bit-level logic; CKKS limits on comparison, exponentials, roots

Framework / Library	**Encryption Scheme(s)**	**ML Support Type**	**Key Features**	**Operation Limitations**
HEAAN	CKKS (approximate)	Matrix ops, DNNs	Original CKKS; widely used in research	CKKS: approximation error in log, exp, root; no logic support
CUFHE	TFHE	Boolean functions	GPU-accelerated binary FHE ops (CUDA)	Only for logic circuits; not usable for DNNs or real-number ops
This work (Cloud)	CKKS, TFHE	LR, SVM, KNN	Supports both CKKS and TFHE	Library dynamically selects either CKKS/TFHE based on user accuracy requirement
This work (Edge)	TFHE	LR, SVM, KNN	Supports both CKKS and TFHE	Memory-efficient library selection for exact computation using TFHE

After the successful implementation of encrypted KNN and encrypted SVM in distributed environment, the next encrypted LR prediction steps were performed on the varying number of edge nodes. The prediction time results are tabulated in the Table 7 b, and the library performances are compared as shown in Fig. 14. In LR implementation, an approximate division algorithm was proposed in section 4.5.5, where we use $EPO2=\{2^0, 2^1, 2^2, 2^3 \dots 2^n\}$ and we have taken 11 elements in this array according to the application. In different datasets, the size of the $EPO2$ array will vary according to the application, which will affect the time taken to compute division result. We have showcased encrypted approximate division time w.r.t. varying length of $EPO2$ array in Fig 15. The ensemble learning approach enhances accuracy and robustness by combining predictions from multiple models. Each model and input data are encrypted using HE before distribution across the Raspberry Pi cluster. Nodes independently process their encrypted data subsets, and the encrypted results are aggregated to generate the final prediction, ensuring data privacy throughout the process. Our designed operators are sufficient to implement a DNN. As noted in the Section 1, the implementation of deep neural networks is not required for this data set. However to show the completeness of our operators, we implemented a simple DNN using the same dataset, with results presented in Table 8. The results are specific to the Breast Cancer Wisconsin dataset. The number of nodes are hyperparameters that depend on the dataset's features and the model architecture. These choices influence performance, and results may vary across different datasets and ML/DL models. Our analysis shows our proposed operators are also sufficient to implement LLM attention layer. However, straight forward LLM and DNN implementations incur huge computation overhead. In future work, we plan to propose DNN and LLM specific optimizations to improve performance.

Further, we highlight that Raspberry Pi cluster for FHE computations presents an interesting low-cost alternative to cloud servers. Using multiple Raspberry Pi 4B devices (with 8GB RAM each) interconnected via Gigabit Ethernet, we can create a budget-friendly edge computing cluster for around \$500-1000 (4-8 nodes). The main tradeoff comes in computational power, while a cloud server instance (like AWS c6i.16xlarge) can process FHE operations 10-20x faster due to higher CPU, GPU frequencies and optimized memory hierarchies, it costs roughly \$2-3 per hour. The Pi cluster excels for non-time-critical FHE workloads where data privacy and long-term cost savings are priorities.

Moreover, results in Table 6, 7 shows the implementation of our encrypted ML algorithms distributed in nature and improves with the increase of nodes in cluster. Here, we have tested with cluster of 11 nodes, but that can be improved with larger cluster. Further, our results in Figure 13, 14 show that final timing performance also dependent on the choice of libraries and supported levels of homomorphic depth. Finally, Table 9 shows the limitations of existing frameworks and advantage of FHEMaLe framework. Our framework can perform machine learning prediction in just 17 seconds on the cloud and 4.15 minutes on the edge, offering flexibility based on performance and deployment needs.

Our experimental results demonstrate the trade-offs between CKKS and TFHE schemes in encrypted machine learning inference. As shown in Table 4.10, CKKS achieves significantly faster single prediction times compared to TFHE but at a much higher memory cost per element (107 MB vs 0.064 MB).

Table 10. Encrypted ML Prediction Times: Cloud vs. Edge Deployment

	OpenFHE CKKS (cloud)	OpenFHE c++ TFHE (edge)
KNN	248 sec	37.06 min
SVM	18 sec	4.15 min
LR	57 sec	7.82 min

9 CONCLUSION AND FUTURE WORK

In future work, we plan to investigate further improvements through hybrid scheme switching, leveraging libraries such as OpenFHE for dynamic switching between CKKS and TFHE [43]. We will also explore scalable key management strategies and adaptive load balancing for enhanced performance in large-scale edge networks. While encrypted ML processing is slower than plaintext prediction, our end-to-end encrypted framework is well-suited for applications where real-time results are not critical, such as medical diagnosis, weather forecasting, financial analysis, and energy grid optimization, where data security is prioritized.

REFERENCES

[1] Ahmad Al Badawi et al. “Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus”. In: *IEEE Transactions on Emerging Topics in Computing* 9.3 (2020), pp. 1330–1343.

[2] Abdulmalik Alwarafy et al. “A Survey on Security and Privacy Issues in Edge-Computing-Assisted Internet of Things”. In: *IEEE Internet of Things Journal* 8.6 (2021), pp. 4004–4022. DOI: 10.1109/JIOT.2020.3015432.

[3] *Amazon EC2 Pricing*. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 2023-12-01. 2023.

[4] Zhongyuan Bian, Ping Li, and Sheng Ma. “When Homomorphic Encryption Marries Federated Learning: Secure Distributed Learning Without Sharing Raw Data”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 1302–1317.

[5] Fabian Boemer et al. “mpcHE: Secure evaluation of deep neural networks using mixed-protocol multi-party computation”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1471–1487.

[6] Keith Bonawitz et al. “Practical secure aggregation for privacy-preserving machine learning”. In: *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1175–1191.

[7] Raphael Bost et al. “Machine learning classification over encrypted data”. In: *Cryptology ePrint Archive* (2014).

[8] Florian Bourse et al. “Fast homomorphic evaluation of deep discretized neural networks”. In: *Annual International Cryptology Conference*. Springer, 2018, pp. 483–512.

[9] Ayantika Chatterjee and Indranil Sengupta. “Sorting of Fully Homomorphic Encrypted Cloud Data: Can Partitioning be Effective?” In: *IEEE Transactions on Services Computing* 13.3 (2020), pp. 545–558. DOI: 10.1109/TSC.2017.2711018.

[10] Ayantika Chatterjee and Indranil Sengupta. “Translating Algorithms to Handle Fully Homomorphic Encrypted Data on the Cloud”. In: *IEEE Transactions on Cloud Computing* 6.1 (2018), pp. 287–300. DOI: 10.1109/TCC.2015.2481416.

[11] Hao Chen et al. “Logistic regression over encrypted data from fully homomorphic encryption”. In: *BMC medical genomics* 11 (2018), pp. 3–12.

[12] Qi Chen et al. “Privacy-preserving searchable encryption in the intelligent edge computing”. In: *Computer Communications* 164 (2020), pp. 31–41.

[13] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. “Efficient homomorphic comparison methods with optimal complexity”. In: *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part II 26*. Springer. 2020, pp. 221–256.

[14] Jung Hee Cheon, Hyojin Lee, and Yongsoo Song. “A practical bootstrapping for approximate homomorphic encryption with non-sparse key-switching”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2018, pp. 360–384.

[15] Jung Hee Cheon et al. “Encrypted image classification using scalable and efficient logistic regression and neural network”. In: *International Conference on Information Security and Cryptology*. Springer, 2019, pp. 100–117.

[16] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Cham: Springer International Publishing, 2017, pp. 409–437. ISBN: 978-3-319-70694-8.

[17] Jung Hee Cheon et al. “Homomorphic encryption for arithmetic of approximate numbers”. In: *Advances in cryptology–ASIACRYPT 2017: 23rd international conference on the theory and applications of cryptology and information security, Hong kong, China, December 3-7, 2017, proceedings, part i 23*. Springer. 2017, pp. 409–437.

[18] Ilaria Chillotti et al. *TFHE: Fast Fully Homomorphic Encryption over the Torus*. Cryptology ePrint Archive, Paper 2018/421. https://eprint.iacr.org/2018/421. 2018. URL: https://eprint.iacr.org/2018/421.

[19] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[20] Haokun Fang and Quan Qian. "Privacy Preserving Machine Learning with Homomorphic Encryption and Federated Learning". In: *Future Internet* 13.4 (2021). ISSN: 1999-5903. DOI: 10.3390/fi13040094. URL: https://www.mdpi.com/1999-5903/13/4/94.

[21] Jeff Geerling. *Building a Raspberry Pi Cluster*. https://www.jeffgeerling.com/blog/2020/building-raspberry-pi-cluster. 2020.

[22] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178.

[23] Micah Goldblum et al. "Dataset security for machine learning: Data poisoning, backdoor attacks, and defenses". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.2 (2022), pp. 1563–1580.

[24] Rob Hall, Stephen E Fienberg, and Yuval Nardi. "Secure multiple linear regression based on homomorphic encryption". In: *Journal of Official Statistics* 27.4 (2011), pp. 669–691.

[25] Awni Y Hannun et al. "Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network". In: *Nature medicine* 25.1 (2019), pp. 65–69.

[26] Shengshan Hu et al. "Securing fast learning! ridge regression over encrypted big data". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE. 2016, pp. 19–26.

[27] Mihailo Isakov et al. "Survey of attacks and defenses on edge-deployed neural networks". In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2019, pp. 1–8.

[28] Abdul Rehman Javed et al. "A collaborative healthcare framework for shared healthcare plan with ambient intelligence". In: *Human-centric Computing and Information Sciences* 10.1 (2020), p. 40.

[29] Ronaldo Jerang et al. "Privacy-Preserving of Edge Intelligence using Homomorphic Encryption". In: *2023 3rd International Conference on Intelligent Technologies (CONIT)*. 2023, pp. 1–6. DOI: 10.1109/CONIT59222.2023.10205745.

[30] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "Gazelle: A Low Latency Framework for Secure Neural Network Inference". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1651–1669.

[31] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. "Secure multi-party differential privacy". In: *Advances in neural information processing systems* 28 (2015).

[32] Murat Kantarcioglu. "A survey of privacy-preserving methods across horizontally partitioned data". In: *Privacy-Preserving Data Mining: Models and Algorithms* (2008), pp. 313–335.

[33] Joon-Woo Lee et al. "Privacy-preserving machine learning with fully homomorphic encryption for deep neural network". In: *iEEE Access* 10 (2022), pp. 30039–30054.

[34] Jing Lin et al. "ML Attack Models: Adversarial Attacks and Data Poisoning Attacks". In: *arXiv preprint arXiv:2112.02797* (2021).

[35] Fang Liu, Wee Keong Ng, and Wei Zhang. "Encrypted SVM for outsourced data mining". In: *2015 IEEE 8th International Conference on Cloud Computing*. IEEE. 2015, pp. 1085–1092.

[36] Guanghua Liu. "A Q-Learning-based distributed routing protocol for frequency-switchable magnetic induction-based wireless underground sensor networks". In: *Future Generation Computer Systems* 139 (2023), pp. 253–266.

[37] Zhuoran Ma et al. "Lightweight privacy-preserving medical diagnosis in edge computing". In: *IEEE Transactions on Services Computing* 15.3 (2020), pp. 1606–1618.

[38] Chiara Marcolla et al. "Survey on Fully Homomorphic Encryption, Theory, and Applications". In: *Proceedings of the IEEE* 110.10 (2022), pp. 1572–1609. DOI: 10.1109/JPROC.2022.3205665.

[39] Daniele Micciancio and Yuriy Polyakov. "Bootstrapping in FHEW-like Cryptosystems". In: *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 17–28. ISBN: 9781450386562. DOI: 10.1145/3474366.3486924. URL: https://doi.org/10.1145/3474366.3486924.

[40] MG Sarwar Murshed et al. "Machine learning at the network edge: A survey". In: *ACM Computing Surveys (CSUR)* 54.8 (2021), pp. 1–37.

[41] *NuFHE*. [Online] https://github.com/nucypher/nufhe. URL: https://github.com/nucypher/nufhe.

[42] *OpendCL*. [Online] https://github.com/KhronosGroup/OpenCL-Guide.git. URL: https://github.com/KhronosGroup/OpenCL-Guide.git.

[43] *OpenFHE*. [Online] https://www.openfhe.org/. URL: https://www.openfhe.org/.

[44] Heejin Park et al. "Efficient machine learning over encrypted data with non-interactive communication". In: *Computer Standards & Interfaces* 58 (2018), pp. 87–108.

[45] *PiCluster: Raspberry Pi Cluster Project*. https://picluster.org. 2023.

[46] B Pradeep Kumar Reddy and Ayantika Chatterjee. "SMLaaS: Secure Machine Learning as a Service Ensuring Data and Model Parameter Privacy". In: *Security and Privacy* 8.4 (2025), e70047.

[47] Feifei Qiao, Zhong Li, and Yubo Kong. "A privacy-aware and incremental defense method against GAN-based poisoning attack". In: *IEEE Transactions on Computational Social Systems* (2023).

[48] Mohammad Saidur Rahman et al. "Towards privacy preserving AI based composition framework in edge networks using fully homomorphic encryption". In: *Engineering Applications of Artificial Intelligence* 94 (2020), p. 103737. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2020.103737. URL: https://www.sciencedirect.com/science/article/pii/S0952197620301512.

[49] B Pradeep Kumar Reddy, Ruchika Meel, and Ayantika Chatterjee. *Encrypted KNN Implementation on Distributed Edge Device Network*. Cryptology ePrint Archive, Paper 2024/648. https://eprint.iacr.org/2024/648. 2024. URL: https://eprint.iacr.org/2024/648.

[50] B. Reddy and Ayantika Chatterjee. "Encrypted Classification Using Secure K-Nearest Neighbour Computation". In: Nov. 2019, pp. 176–194. ISBN: 978-3-030-35868-6. DOI: 10.1007/978-3-030-35869-3_13.

[51] B. Reddy., Ruchika Meel., and Ayantika Chatterjee. "Encrypted KNN Implementation on Distributed Edge Device Network". In: *Proceedings of the 21st International Conference on Security and Cryptography - SECRYPT*. INSTICC. SciTePress, 2024, pp. 680–685. ISBN: 978-989-758-709-2. DOI: 10.5220/0012761100003767.

[52] Fatemeh Rezaeibagha et al. "Authenticable Additive Homomorphic Scheme and its Application for MEC-Based IoT". In: *IEEE Transactions on Services Computing* 16.3 (2023), pp. 1664–1672. DOI: 10.1109/TSC.2022.3211349.

[53] Rashmi R Salavi, Mallikarjun M Math, and UP Kulkarni. "A survey of various cryptographic techniques: From traditional cryptography to fully homomorphic encryption". In: *Innovations in Computer Science and Engineering: Proceedings of the Sixth ICICSE 2018*. Springer. 2019, pp. 295–305.

[54] Chen Song and Xinghua Shi. "ReActHE: A homomorphic encryption friendly deep neural network for privacy-preserving biomedical prediction". In: *Smart Health* 32 (2024), p. 100469.

[55] *TFHE*. [Online] https://tfhe.github.io/tfhe/. URL: https://tfhe.github.io/tfhe/.

[56] Huy Trinh et al. "Energy-aware mobile edge computing and routing for low-latency visual data processing". In: *IEEE Transactions on Multimedia* 20.10 (2018), pp. 2562–2577.

[57] B Venkata Kranthi and Borra Surekha. "Real-time facial recognition using deep learning and local binary patterns". In: *Proceedings of International Ethical Hacking Conference 2018: eHaCON 2018, Kolkata, India*. Springer. 2019, pp. 331–347.

[58] Xiaoyan Yan, Qilin Wu, and You Peng Sun. "A Homomorphic Encryption and Privacy Protection Method Based on Blockchain and Edge Computing". In: *Wirel. Commun. Mob. Comput.* 2020 (2020), 8832341:1–8832341:9. URL: https://api.semanticscholar.org/CorpusID:221380986.

[59] Bo Yang et al. "Mobile-edge-computing-based hierarchical machine learning tasks distribution for IIoT". In: *IEEE Internet of Things Journal* 7.3 (2019), pp. 2169–2180.

[60] Yu Yao et al. "Secure transmission scheme based on joint radar and communication in mobile vehicular networks". In: *IEEE transactions on intelligent transportation systems* (2023).

[61] Mahmut Taha Yazici, Shadi Basurra, and Mohamed Medhat Gaber. "Edge machine learning: Enabling smart internet of things applications". In: *Big data and cognitive computing* 2.3 (2018), p. 26.

[62] Tjalling J Ypma. "Historical development of the Newton–Raphson method". In: *SIAM review* 37.4 (1995), pp. 531–551.