Leia: A Lightweight Cryptographic Neural Network Inference System at the Edge

Xiaoning Liu[1], Bang Wu[2], Xingliang Yuan[2], Xun Yi[1]

[1] RMIT University, Australia {maggie.liu, xun.yi}@rmit.edu.au
[2] Monash University, Australia {bang.wu, xingliang.yuan}@monash.edu

Abstract. The advances in machine learning have revealed its great potential for emerging mobile applications such as face recognition and voice assistant. Models trained via a Neural Network (NN) can offer accurate and efficient inference services for mobile users. Unfortunately, the current deployment of such service encounters privacy concerns. Directly offloading the model to the mobile device violates model privacy of the model owner, while feeding user input to the service compromises user privacy. To address this issue, we propose, tailor, and evaluate Leia, a lightweight cryptographic NN inference system at the edge. Unlike prior cryptographic NN inference systems, Leia is designed with two mobile-friendly perspectives. First, Leia leverages the paradigm of edge computing wherein the inference procedure keeps the model closer to the mobile user to foster low latency service. Specifically, Leia's architecture consists of two non-colluding edge services to obliviously perform NN inference on the encoded user data and model. Second, Leia's realization makes the judicious use of potentially constrained computational and communication resources in edge devices. In particular, Leia adapts the Binarized Neural Network (BNN), a trending flavor of NN model with low memory footprint and computational cost, and purely chooses the lightweight secret sharing techniques to develop secure blocks of BNN. Empirical validation executed on Raspberry Pi confirms the practicality of Leia, showing that Leia can produce a prediction result with 97% accuracy by 4 seconds in the edge environment.

1 Introduction

Recent flourishment of Machine Learning has promoted the Neural Network (NN) powered mobile applications such as face detection cameras and speech recognition assistants. A NN prediction service is typically deployed through two scenarios. One scenario is the on-device NN prediction service, where the application downloads the pre-trained model from a company who owns the model, and performs an inference task over user's data on mobile device [5]. Another scenario relies on the cloud service provider (e.g., Google Cloud AI [3]), where user data and model are delivered to cloud who runs the NN inference task and sends back the prediction result to the mobile device [6]. Unfortunately, both scenarios are troublesome due to the increasingly raised privacy issues. User's data contains sensitive information about their daily activities. Uploading such data to cloud in plaintext can put individual's privacy in danger [9,19]. On the other hand, from the aspect of model owners, their models are valuable and often trained on proprietary data [8,26,30]. The unauthorized exposure of the proprietary model and underlying confidential data inflects severe commercial damages.

To alleviate the privacy issues, one approach seems plausible is to delegate the inference tasks with encrypted model and user data to a centralized cloud server. However, this approach relies on heavy cryptographic techniques like (fully) homomorphic encryption (HE). An efficient alternative tends to use secure multi-party computation (MPC) techniques with specialized designs that can execute NN inference over encrypted user input data and/or encrypted models. For example, Delphi [8], MiniONN [30] proceed the inference tasks between the mobile device and the model owner, while continuous interaction is involved between them during secure computation. Namely, both parties have to be online and connected throughout the entire inference process. It is noteworthy that the above rigid operational confinement might not be always feasible in cellular networks. Other systems like XONN [35], Quotient [10] employ constant-round secure 2-party computation protocols that result in less engagement of mobile device and model owner. These systems still consume large bandwidth costs due to the inherent drawbacks of the MPC techniques relied on, i.e., Garbled Circuits. Later in Section 6, we demonstrate the large bandwidth saving by our design compared to GC based realization.

Our Contributions: In this paper, we propose and enable Leia, a cryptographic NN inference system executing at the edge. Leia takes the edge based architecture as a starting point, harnessing the novelty from system, cryptographic and machine learning domains. The combination endows Leia privacy assurance and seamless embracement of NN powered mobile applications.

Our first insight is to leverage edge computing, wherein the processing keeps the model closer to the mobile user so as to foster low latency service of NN inference [1, 45]. However, devising edge based architecture is non-trivial for Leia's scenario. Both models and user's data should fully be protected against edge nodes during inference. Moreover, relaxing the constraint of the model provider and mobile device being online is expedient because of the dynamic network effects in cellular networks. To this end, we resort to the edge nodes as two non-colluding computational services to fulfill the above privacy objective and operational requirement simultaneously. The model provider encodes the model and sends the encoded one to edge nodes only once. After that, the user mobile device can submit the encoded input and obtain the encoded inference result. Within the process, edge nodes obliviously perform inference without further interacting with either the model provider or user mobile device.

The edge based paradigm entails judicious usage of computational and communication resources. We thus choose the relatively lightweight secret sharing techniques to offer Leia security guarantee. Yet, merely transforming the NN inference procedure into cryptographic operations does not necessarily achieve satisfied efficiency in communication and computation for mobile and edge devices. Instead, our second insight is the adoption of the Binarized Neural Network (BNN), a special flavor of NN model with weights and activations are all confined to ± 1. Because the small BNN model can drastically reduce the memory demand, and the beneath operations over binary values are more compatible with cryptographic primitives, it thus becomes our natural choice. Thereby, we subtly build Leia from ground up with secure layer functions, including the secure linear layers (secure convolutional layer SCONV and secure fully connected layer SFC), the secure batch normalization function (SecBN), the secure binary activation function (SecBA), and the secure max pooling layer (SMP). They are highly customized for the BNN and securely realized based on secret sharing as the building blocks of Leia. We implement and deploy Leia to Raspberry Pi, making Leia produce a 97% accurate prediction result by 4 seconds in the edge environment.

Organization: The rest of the paper is organized as follows. Section 2 investigates related work. Section 3 introduces the preliminaries used in this paper. Section 4 overviews the system architecture and threat model. Section 5 expatiates the protocol designs. Empirical evaluation is given in Section 6.

2 Related Works

Privacy-Preserving Neural Network Training and Inference: CryptoNet [21] adapts the leveled HE to perform privacy-preserving NN inference in an outsourced environment. Despite some optimizations have been employed, CryptoNet still suffers from intensive computational overheads due to heavy weight HE. Some other systems [8,26,30] consider a different setting, in which the client directly communicates with the model provider for inference, but does not want to reveal her input. Gazelle [26] devises a 2PC-based secure NN inference framework combining the lattice-based packed additive HE and Yao's Garbled Circuits (GC). After that, Delphi [8] and MiniONN [30] are proposed with careful optimization and achieve higher efficiency. Difference from our work, the above systems require interactions between the client and model provider during secure computation. Our work leverages edge computing relaxing both parties from being actively online throughout the inference.

SecureML [32] proposes privacy-preserving NN training and inference based on the generic MPC framework. The proposed protocol uses GC to conduct nonlinear activation function. Quotient [10] and XONN [35] rely on GC to design privacy-preserving NN inference over quantized NN models with weights which are restricted within $\{-1, +1, 0\}$ and $\{-1, +1\}$, respectively. Such quantization allows for the conversion from arithmetic operations to Boolean operations, and thus are more compatible with the protocols realized with GC free-XOR optimizations. However, the GC-based protocols introduce prohibitive bandwidth consumption which can be the bottleneck at the edge. This escalated consumption may also incur considerable charges by cellular network service providers. Our work subtly builds a privacy-assured collaborative inference protocol from ground up, harnessing the novelty from both cryptographic and machine learning literature. Our system originates a design where an all-binarized neural network

inference procedure is securely carried out with customized layer functions under lightweight MPC primitives, and thus is particularly suitable for the application deployment for mobile devices. Empirical evidence shown in Section 6 confirms that, for the equivalent functionalities, Leia's realizations introduce 30-2500$\times$ less bandwidth costs than the GC-based realizations. Very recently, Helen [43] proposes a linear model learning protocol in a fully decentralized setting to thwart the malicious parties.
Secure Multi-Party Computation Framework: Privacy-preserving machine learning protocols can be carried out via generic MPC techniques, such as Garbled Circuits [41], Secret Sharing [12,22], , and the frameworks mixed with multiple primitives [18,31,33]. Among these, a line falls into devising machine learning specified MPC frameworks, such as the two-party framework with a trusted party to generate correlated randomness proposed in Chameleon [36], the three-party frameworks proposed in ABY3 [31], SecureNN [39], and the four-party framework in Trident [33]. For performance consideration, recent privacy-preserving machine learning systems opt for specialized and optimized designs instead of direct adoption of generic MPC frameworks [8, 10, 35, 43].
Binarized Neural Networks: Binarized neural networks (BNN) [16, 17], i.e., neural networks with weights and possible activations restricted to ± 1, have been applied in various domains, such as image classification [28, 34], speech recognition [27]. It is also considered as a line of model quantization, which is a training network with low precision weights and activations like fixed-point integers [23] and ternary [24]. Note that normalization techniques [25, 37] are widely adopted in modern neural network training to avoid the activations growing too large to unstablize the model. Our design employs batch normalization [25] to align with the realization of BNN [17].

3 Background

3.1 Binarized Neural Networks

Binarized Neural Network [17] (BNN) comprises a sequence of *linear* and *non-linear* layers, where the number of layers L indicates the depth of network. The BNN inference takes as input a matrix (i.e., $X \in \mathbb{R}^{n_{in} \times m_{in}}$) or a tensor (i.e., $\mathbf{X} \in \mathbb{R}^{c \times n_{in} \times m_{in}}$) representing the task-specific raw data, and produces a prediction result based on a trained model (i.e., $W \in \{-1, +1\}^{n \times m}$ or $\mathbf{W} \in \{-1, +1\}^{c \times n \times m}$). The parameters c, n_{in} (and n), m_{in} (and m) represent the number of *channel*, the *width* and *height* of input (and model), respectively. In BNN, all weights are binarized as $\{-1, +1\}$, which makes the model significantly smaller than the one of an equivalent network with high precision weights [34].
Linear Layers: The linear layers typically can be of two types: the fully connected layers (FC) and the convolutional layers (CONV). Both types can be formulated as the variances of the vector dot product ($\mathsf{VDP}(\cdot, \cdot)$) between two vectors $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{w} \in \mathbb{R}^n$ as: $\mathsf{VDP}(\mathbf{x}, \mathbf{w}) = \sum_{k=1}^{n} w_k \cdot x_k$, where w_k, x_k indicate the k-th element of vector $\mathbf{x}, \mathbf{w}$, respectively.

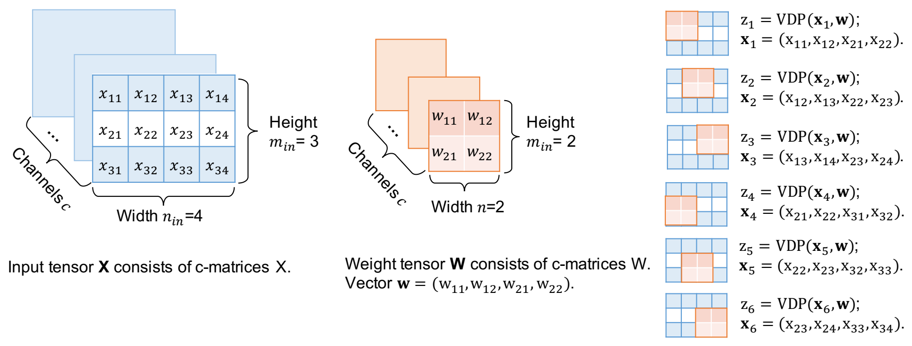
x_{11} x_{12} x_{13} x_{14}
x_{21} x_{22} x_{23} x_{24}
x_{31} x_{32} x_{33} x_{34}
Height m_{in}= 3
Channels c
Width n_{in}=4
Input tensor **X** consists of c-matrices X.
w_{11} w_{12}
w_{21} w_{22}
Height m_{in}= 2
Channels c
Width n=2
Weight tensor **W** consists of c-matrices W.
Vector **w** = (w_{11},w_{12},w_{21},w_{22}).
z_1 = VDP($\mathbf{x}_1$,$\mathbf{w}$); $\mathbf{x}_1$ = (x_{11},x_{12},x_{21},x_{22}).
z_2 = VDP($\mathbf{x}_2$,$\mathbf{w}$); $\mathbf{x}_2$ = (x_{12},x_{13},x_{22},x_{23}).
z_3 = VDP($\mathbf{x}_3$,$\mathbf{w}$); $\mathbf{x}_3$ = (x_{13},x_{14},x_{23},x_{24}).
z_4 = VDP($\mathbf{x}_4$,$\mathbf{w}$); $\mathbf{x}_4$ = (x_{21},x_{22},x_{31},x_{32}).
z_5 = VDP($\mathbf{x}_5$,$\mathbf{w}$); $\mathbf{x}_5$ = (x_{22},x_{23},x_{32},x_{33}).
z_6 = VDP($\mathbf{x}_6$,$\mathbf{w}$); $\mathbf{x}_6$ = (x_{23},x_{24},x_{33},x_{34}).

Fig. 1. An illustration of the CONV layer.

The FC layer takes as input a matrix X (either representing the input raw data or the output from previous layer), flattens the matrix into a single vector $\mathbf{x} \in \mathbb{R}^n$, applies the weight matrix $W \in \{-1,+1\}^{n\times m}$ and bias vector $\mathbf{bias} \in \mathbb{R}^m$, and outputs a vector $\mathbf{z} \in \mathbb{R}^m$. For $k \in [1,m]$, it repeatedly proceeds the $z_k = \mathsf{VDP}(\mathbf{x},\mathbf{w}_k) + bias_k$ on the input vector $\mathbf{x}$ and each vector $\mathbf{w}_k$ of the W, and the output z_k is the k-th element of the resulting vector $\mathbf{z}$. The layer output $\mathbf{z}$ can be submitted to the non-linear operations.

The CONV layer is normally applied for image classification. It takes as input an image tensor $\mathbf{X} \in \mathbb{R}^{c_{in}\times n_{in}\times m_{in}}$, applies the weight (aka filter) tensor $\mathbf{W} \in \{-1,+1\}^{c_o\times c_{in}\times n\times m}$ and bias vector $\mathbf{bias} \in \mathbb{R}^{c_o}$, and outputs a feature map tensor $\mathbf{Z} \in \mathbb{R}^{c_o\times n_o\times m_o}$. For each input matrix $X \in \mathbb{R}^{n_{in}\times m_{in}}$, the CONV layer repeatedly moves the weight matrix $W \in \{-1,+1\}^{n\times m}$ as a sliding window, from left to right and top-down with given stride, until passing through the entire image matrix. And inside each sliding window s, it flattens X and W as vectors $\mathbf{x}$ and $\mathbf{w}$, and proceeds $z_s = \mathsf{VDP}(\mathbf{x},\mathbf{w}) + bias_s$ to obtain z_s as one element in resulting matrix $Z \in \mathbb{R}^{n_o\times m_o}$. Fig. 1 provides a concrete illustration of how the CONV layers can be reformulated as the $\mathsf{VDP}(\cdot,\cdot)$. For simplicity purpose, it takes as input the toy-value 4×3 input matrix X, 2×2 weight matrix W, and shows the procedure to calculate each element in resulting matrix Z based on the VDP operations.

Binary Activation: The non-linear binary activation function (BA) is attached on each neuron, and can be viewed as a portion of the linear layers. On each neuron, the BA function takes as input the real-valued activation $a \in \mathbb{R}$ outputted from previous operation, performs element-wise $\mathsf{Sign}(\cdot)$ function, and outputs the sign bit as the binarized activation $a \in \{-1,+1\}$.

Batch Normalization: Batch Normalization [25] (BN) is used to rectify the distribution of each layer's input so as to speed up the NN training procedure and regularize the model. The procedure of the BN function during inference performs as follows: (1) it performs element-wise normalization on each neuron's feature a via $\hat{a} = (a-\mu)/\sigma$ where μ is the *running mean* and the non-zero σ is *running variance* the of training dataset; and (2) it applies the *scale* parameter $\gamma \in \mathbb{R}$ and the *shift* parameter $\beta \in \mathbb{R}$ to get the output $z = \gamma \cdot \hat{a} + \beta$. Note, the parameters $\mu, \delta, \gamma, \beta$ are highly dependent on the trained data and can be used

to deduce the data distribution, and they thus have to be carefully protected during the secure computation.
Max Pooling Layer: Max pooling layer (MP) is normally applied straight after the CONV layer to downsample the image. It takes the matrices outputted from the CONV layer, slides a certain window on each matrix, and obtains the maximum value within the window as an element of the output matrix.

3.2 Cryptographic Primitives

Correlated Oblivious Transfer: Oblivious Transfer (OT) is a fundamental cryptographic primitive that can obliviously carry out two-party secure computation. Given engaged two parties, a *sender* P_0 holding its input a pair of binary strings $m_0, m_1 \in \{0,1\}^\ell$, and a *receiver* P_1 holding its input a choice bit $\mathfrak{b} \in \{0,1\}$, a 1-out-of-2 OT functionality for the ℓ-bit strings allows the receiver P_1 to obtain $m_\mathfrak{b}$, while ensuring that P_0 learns nothing about $\mathfrak{b}$, and P_1 learns nothing about $m_{1-\mathfrak{b}}$.

Correlated OT (COT) [11] is one particular OT flavor running between two parties in a similar way with improved practicality. The main difference in a COT is that, instead of inputting two messages directly, the sender constructs and inputs a correlation function $f_\Delta(\cdot)$ to link its two input messages m_0 and m_1 in a way that m_0 is a random value and $m_1 = f_\Delta(m_0)$. The m_0 and $m_\mathfrak{b} \in \{m_0, m_1\}$ are the output of COT under $f_\Delta(\cdot)$. Note that the function $f_\Delta(\cdot)$ is a correlation robust random oracle $H : \{0,1\}^\ell \to \{0,1\}^\ell$ and can be instantiated using a one-way hash function. We denote the above described COT functionality as $COT(m_0, f_\Delta(\cdot); \mathfrak{b})$. The n-times COT$_\ell$ (i.e., $n\times$COT$_\ell$) can be run in parallel, where each COT$_\ell$ is used for transferring the ℓ-bit strings.
Arithmetic Sharing and Multiplication Triple: Arithmetic sharing [12] produces two Arithmetic shares $\langle x\rangle_0^A$ and $\langle x\rangle_1^A$ of an ℓ-bit secret value x in the ring $\mathbb{Z}_{2^\ell}$, where the reconstruction is performed via $x = \langle x\rangle_0^A + \langle x\rangle_1^A \pmod{2^\ell}$. A party P_0 who wants to share x can generate the randomness $r \in_R \mathbb{Z}_{2^\ell}$ and sends $x - r \pmod{2^\ell}$ to the counterparty P_1. At the end, P_0 sets $\langle x\rangle_0^A = r$ and P_1 produces $\langle x\rangle_1^A = x - r \pmod{2^\ell}$. In this paper, all operations over Arithmetic shares are performed under $\pmod{2^\ell}$ unless explicitly mentioned.

Addition/subtraction over shares ($\langle z\rangle_i^A = \langle x\rangle_i^A \pm \langle y\rangle_i^A$), multiplication by a public value ($\langle z\rangle_i^A = \eta \cdot \langle x\rangle_i^A$) can be efficiently calculated by each party P_i at local without any interaction. Multiplication over two shares ($\langle z\rangle^A = \langle x\rangle^A \cdot \rangle y\rangle^A$) requires assistance with pre-computed arithmetic-shared Multiplication Triple, i.e., a type of Beaver's Triple [13] in the format $\langle c\rangle^A = \langle a\rangle^A \cdot \langle b\rangle^A$. Each party P_i sets $\langle e\rangle_i^A = \langle x\rangle_i^A - \langle a\rangle_i^A$ and $\langle f\rangle_i^A = \langle y\rangle_i^A - \langle b\rangle_i^A$. Both parties interact to reconstruct e and f. Party P_i then sets $\langle z\rangle_i^A = i \cdot e \cdot f + f \cdot \langle a\rangle_i^A + e \cdot \langle b\rangle_i^A + \langle c\rangle_i^A$. Note that multiplication triples are data-independent and can be generated through cryptographic approaches like additive HE and COT [11,18,30], or by a trusted third party like in [39,44].
Boolean Sharing and Boolean AND Triple: Boolean sharing [18,22] can be viewed as the Arithmetic sharing over $\mathbb{Z}_2$. In this paper, we use GMW protocol [22] to evaluate the function represented as Boolean circuit. It produces two

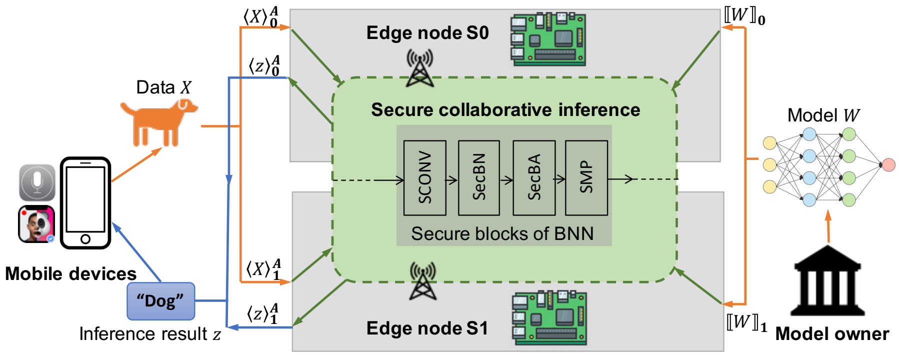

Fig. 2. System architecture.

Boolean shares $[\![x]\!]_0$ and $[\![x]\!]_1$ of a secret bit x between two parties P_0 and P_1, respectively. Reconstruction of x is performed via $x = [\![x]\!]_0 \oplus [\![x]\!]_1$. The addition over Arithmetic shares in $\mathbb{Z}_2$ is reformulated as XOR operation ($\oplus$), and is computed locally by each party P_i ($[\![z]\!]_i = [\![x]\!]_i \oplus [\![y]\!]_i$). Meanwhile, the multiplication is replaced by the bitwise AND operation ($\wedge$) over Boolean shares, and is calculated with the assistance of pre-computed Boolean AND Triple $[\![c]\!] = [\![a]\!] \wedge [\![b]\!]$. Similar to multiplication triple, the AND triple can be generated via Random OT (ROT) [11,31] or by a trusted third party.

4 System Overview

4.1 Architecture

Fig. 2 illustrates Leia's system architecture, which involves three entities: the mobile device, the model owner, and the two distinct edge nodes S_0 and S_1. In practice, the model owner can be a ML-powered mobile application developing company, who obtains a customized NN model based on proprietary data. It resorts to Leia to provide secure inference service for its users without revealing the model in cleartext. The mobile device of the user collects user's data input and asks the mobile application for inference tasks without revealing the private user input. The two edge nodes can be deployed from separate edge computing service providers like Azure IoT Edge [2] and AWS Lambda@Edge [4].

From a high level point of view, Leia's cryptographic NN inference service is operated as follows. The model owner holds a BNN model and deploys the encoded model matrix W to the two edge nodes S_0 and S_1, where W is secret-shared (i.e., Boolean shares) into $[\![W]\!]_0$ and $[\![W]\!]_1$. Once the mobile device invokes a NN inference request, the raw input will be protected as a secret-shared (i.e., Arithmetic shares) matrix $\langle X\rangle_0^A$ and $\langle X\rangle_1^A$, and be submitted to corresponding edge nodes. After that, the two nodes run secure collaborative inference procedure (green box in Fig. 2), and send the encoded inference result $\langle z\rangle_0^A$ and $\langle z\rangle_1^A$ to the mobile device which can reconstruct the result (e.g., a label to classify the user's input). Leia's secure inference procedure consists of a series of computational blocks in BNN. A typical block in Leia assembles several layer functions, including the secure convolutional layer (SCONV) or secure fully connected layer (SFC), the secure batch normalization function (SecBN), the secure binary activation function (SecBA) and the secure max pooling layer (SMP). Note that, for efficient realization, we craft a secure normalized binary activation function (SecNBA) combing the SecBN and SecBA functions (see details in Section 5.2).

4.2 Threat Model and Privacy Goal

Leia considers the following threat model: all entities are the *semi-honest* parties, and the two edge nodes are *non-colluding* computing devices. More specifically, every party will faithfully follow the prescribed secure NN inference protocol yet trying to deduce the information from the transcripts exchanged during the protocol execution individually. When corruption happens, a semi-honest adversary can compromise at most one of the edge nodes and either the mobile user or the model owner, while the other parties remain honest. This two-server mode has been used in a group of previous works on privacy-preserving NN inference [10,15,32]. In our practice, we can regard them as from two distinct and well-established edge service providers (e.g., Microsoft Azure IoT Edge service and Amazon Lambda@Edge service) who are not willing to risk reputation and business model to cheat.

Leia guarantees both the privacy of mobile user's data and the model privacy. It hides both the user's data and the model values (e.g., the trained weights and the batch normalization parameters) from being known by the edge nodes. Meanwhile, Leia is consistent with the security guarantees in prior work [8,10,30,32]. That is, the parameters of network architecture is considered as hyper-parameters already known by the edge nodes, including the number of layers, the sizes of weight matrices, and types of operations used in each layer. Such hyper-parameters are data independent and not proprietary since they are usually described in scientific and white papers. We are aware that a malicious user can exploit the inference service as a blackbox oracle to perform attacks [20, 38] to extract auxiliary information from prediction results. Like prior cryptographic inference systems [8,30,35,43], we emphasize that protecting against such attacks is a complementary problem beyond Leia's security scope [29]. Mitigation strategies can consider the adoption of differentially private training algorithms [42].

5 Our Proposed Design

5.1 Secure Linear Layers

We present in this section the secure realizations of linear layers, i.e., the secure convolutional layer (SCONV) and the secure fully connected layer (SFC). As mentioned above, they can be expressed as some variants of $\mathsf{VDP}(\mathbf{x},\mathbf{w}) + bias$ over n-dimensional layer input vector $\mathbf{x}$, weight vector $\mathbf{w}$, and the $bias \in \mathbb{R}$ attached on each neuron. Note that the hidden layer's input is the activation $\mathbf{a}$.

Due to the inherit binarization property of BNN, all weights $\mathbf{w}$ and the hidden layer input $\mathbf{a}$ are restricted as in $\{-1, +1\}$. Only except for the first layer, the input is some real-valued vectors $\mathbf{x} \in \mathbb{R}^n$. Meanwhile, we encode the "-1" as "0" and denote them as $\mathbf{a}^b, \mathbf{w}^b \in \{0, 1\}^n$. Here, we make an important observation from the machine learning literature [25] that the bias can be removed if applying batch normalization, because the scale β in BN achieves the same effect as the bias. Empirically open source learning framework follows this treatment like PyTorch [7]. Thereby, in this paper, we keep consistent with the setting of recent work in the machine learning domain [7,25] to set the bias as 0, so as to avoid the involvement of real-valued bias and thus fitting better with the MPC techniques.

To realize the linear transformations in a privacy-preserving manner, our design carefully protects both mobile user's data (i.e., the input) and the BNN model (i.e., the weights) with lightweight secret sharing techniques (Arithmetic sharing and Boolean sharing). In particular, we introduce the secure Boolean-VDP function, and the secure Boolean-Arithmetic-VDP function which performs VDP over mixed share representations. They are the main building blocks to realize the linear layers.

Input: Boolean shares of binary activation vector $\mathbf{a}^b \in \{0, 1\}^n$, and binary weight vector $\mathbf{w}^b \in \{0, 1\}^n$.
Output: Arithmetic shares of Binary-VDP result $z = \mathsf{VDP}(\mathbf{a}^b, \mathbf{w}^b)$.

1. For each $k \in [1, n]$, S_i computes $[\![t_k^b]\!]_i = [\![a_k^b]\!]_i \oplus [\![w_k^b]\!]_i$; $[\![p_k^b]\!]_i = [\![t_k^b]\!]_i \oplus i$, and sets its share of the XNOR value as $[\![p_k^b]\!]_i$.
2. To convert the XNOR value from over $\mathbb{Z}_2$ to $\mathbb{Z}_{2^\ell}$, S_i proceeds **the** $\mathsf{B2A}(\cdot)$ **gadget** as follows:
   (a) S_0 sets $\langle d_k \rangle_0^A = [\![p_k]\!]_0$, $\langle e_k \rangle_0^A = 0$, and S_1 sets $\langle d_k \rangle_1 = 0$, $\langle e_k \rangle_1^A = [\![p_k]\!]_1$.
   (b) Thereafter, S_0 and S_1 compute $\langle p_k \rangle_i^A = \langle d_k \rangle_i^A + \langle e_k \rangle_i^A - 2 \cdot \langle d_k \rangle^A \cdot \langle e_k \rangle^A$.
3. To compute the PopCount value, S_i proceeds the following steps:
   (a) S_i counts the number of "1" in the XNOR value via $\langle p \rangle_i^A = \sum_{k=1}^n \langle p_k \rangle_i$;
   (b) At the end, S_i sets $\langle z \rangle_i^A = 2\langle p \rangle_0^A - i * n$.

Fig. 3. The secure Boolean-VDP function $\mathsf{SecBVDP}(\cdot, \cdot)$ based on XNOR-PopCount.

Secure Hidden Layer VDP. The secure Boolean-VDP function ($\mathsf{SecBVDP}$) is used to compute VDP for the hidden layers. It takes as input a set of Boolean shares of binary activation vector as $[\![\mathbf{a}^b]\!]_i$ and weight vector as $[\![\mathbf{w}^b]\!]_i$, respectively, and outputs the Arithmetic shares $\langle z \rangle_i^A$ of their Boolean-VDP result, where $i \in \{0, 1\}$ is the identifier of each edge node server. We emphasis that the activation vector is the output of the binary activation function in BNN, which is thus naturally binarized. We note that the VDP operation on two plaintext binary vectors can be converted to a simpler XNOR-PopCount operations [17,34]. That is, for $\mathbf{a}, \mathbf{w} \in \{-1, +1\}$, the element-wise multiplication $p_k = a_k \cdot w_k$ is switched to bitwise-XNOR via $p_k^b = \mathsf{XNOR}(a_k^b, w_k^b) = \neg(a_k^b \oplus w_k^b)$ when $\mathbf{a}^b, \mathbf{w}^b \in \{0, 1\}$, where $k \in [1, n]$. Meanwhile, the accumulation over all multiplication results $\sum_{i=1}^n p_k$ for $p_k \in \{-1, +1\}$ is converted to PopCount, that is, counting the number of “1”s in the resulting binary vector $\mathbf{p}^b = (p_1^b, p_2^b, ..., p_n^b) \in \{0,1\}$ as p. Ultimately, the result is $2p - n$.

Following this convention, Fig. 3 expatiates the $\mathsf{SecBVDP}$ function on secret-shared data. For every element of the binary input vector and weight vector, each edge node S_i computes at local $[\![p_k^b]\!]_i = [\![a_k^b]\!]_i \oplus [\![w_k^b]\!]_i \oplus i$ to perform the element-wise XNOR over its shares. To perform the PopCount operation, a plausible approach is letting each node S_i summarize over its shares directly, i.e., $\langle z\rangle_i = \sum_{k=1}^{n} [\![p_k^b]\!]_i$. However, as the modular addition (i.e., XOR) on Boolean shares is performed as $[\![p_k^b]\!]_0 + [\![p_k^b]\!]_1 \pmod 2$, direct summarization cannot correctly perform the modular addition over $\mathbb{Z}_{2^\ell}$. Given an obvious example that $[\![p_k^b]\!]_0 = 1$ and $[\![p_k^b]\!]_1 = 1$ when $[\![p_k^b]\!] = 0$, yet the reconstructed $\langle z\rangle$ is 2 instead of 0. Thereby, the conversion from over $\mathbb{Z}_2$ to over $\mathbb{Z}_{2^\ell}$ (i.e., the $\mathsf{B2A}$ gadget) should be applied to each Boolean-shared $[\![p_k^b]\!]$. Let $d_k = [\![p_k^b]\!]_0$ and $e_k = [\![p_k^b]\!]_1$. The conversion follows the expression $\langle p_k^b\rangle^A = \langle d_k + e_k - 2 * d_k * e_k\rangle^A$ over $\mathbb{Z}_{2^\ell}$. With this logic, S_0 and S_1 conduct the conversion and obtain their shares $\langle p_k^b\rangle_i^A$ respectively. S_i then performs the summarization to obtain $\langle p\rangle_i^A$ as the result of PopCount. Finally, S_i computes $2\langle p\rangle_i^A - i * n$, given that the length of vector n is the system parameter.

Input: Arithmetic shares of integer input vector $\mathbf{x} \in \mathbb{Z}^n$, Boolean shares of binary weight vector $\mathbf{w}^b \in \{0,1\}^n$.
Output: Arithmetic shares of Binary-Integer-VDP result $z = \mathsf{VDP}(\mathbf{x}, \mathbf{w}^b)$.

1. S_0 and S_1 engage in a $n \times \mathrm{COT}_\ell$. For the k-th COT_ℓ for $k \in [1, n]$, S_i proceeds the following steps to compute $\langle x_k\rangle^A \cdot [\![w_k^b]\!]$:
2. To securely compute $\langle u_k\rangle^A = ([\![w_k^b]\!]_0 \oplus [\![w_k^b]\!]_1) \cdot \langle x_k\rangle_0^A$, such that S_i gets $\langle u_k\rangle_i^A$:
   (a) S_0 acts as a sender, and S_1 is the receiver;
   (b) S_0 constructs a correlation function $f_\Delta(s) = s + (\neg[\![w_k^b]\!]_0 \cdot \langle x_k\rangle_0^A - [\![w_k^b]\!]_0 \cdot \langle x_k\rangle_0^A)$, generates $r_u \in_R \mathbb{Z}_{2^\ell}$, and sets the $m_0 = r_u$;
   (c) S_0 and S_1 run $COT(m_0, f_\Delta(m_0); \mathfrak{b}_u)$;
   (d) As the output, S_1 obtains $m_{\mathfrak{b}_u} = m_0 + \mathfrak{b}_u \cdot (\neg[\![w_k^b]\!]_0 \cdot \langle x_k\rangle_0^A - [\![w_k^b]\!]_0 \cdot \langle x_k\rangle_0^A)$;
   (e) S_0 sets $\langle u_k\rangle_0^A = -m_0 + [\![w_k^b]\!]_0 \cdot \langle x_k\rangle_0^A$, and S_1 sets $\langle u_k\rangle_1^A = m_{\mathfrak{b}_u}$.
3. To securely compute $\langle v_k\rangle^A = ([\![w_k^b]\!]_0 \oplus [\![w_k^b]\!]_1) \cdot \langle x_k\rangle_1^A$, such that S_i gets $\langle v\rangle_i^A$:
   (a) S_1 acts as a sender, and S_0 is the receiver;
   (b) S_1 constructs a correlation function $g_\Delta(s) = s + (\neg[\![w_k^b]\!]_1 \cdot \langle x_k\rangle_1^A - [\![w_k^b]\!]_1 \cdot \langle x_k\rangle_1^A)$, generates the randomness $r_v \in_R \mathbb{Z}_{2^\ell}$, and sets $m_0 = r_v$;
   (c) S_0 sets a choice bit $\mathfrak{b}_v = [\![w_k^b]\!]_0$;
   (d) S_0 and S_1 run $COT(m_0, g_\Delta(m_0); \mathfrak{b}_v)$;
   (e) As the output, S_0 obtains $m_{\mathfrak{b}_v} = m_0 + \mathfrak{b}_v \cdot (\neg[\![w_k^b]\!]_1 \cdot \langle x_k\rangle_1^A - [\![w_k^b]\!]_1 \cdot \langle x_k\rangle_1^A)$;
   (f) S_0 sets $\langle v_k\rangle_0^A = m_{\mathfrak{b}_v}$, and S_1 sets $\langle v_k\rangle_1^A = -m_0 + [\![w_k^b]\!]_1 \cdot \langle x_k\rangle_1^b$.
4. At the end, S_i sets $\langle z\rangle_i^A = \sum_{k=1}^{n} (\langle u_k\rangle_i^A + \langle v_k\rangle_i^A)$.

Fig. 4. The secure Boolean-Arithmetic-VDP function $\mathsf{SecBAVDP}(\cdot,\cdot)$ based on COT.

Secure First Layer VDP. As mentioned above, the two inputs of the first layer are the real-valued matrix of the user's data (e.g., image) and the binarized weight matrix, which will be then submitted to some variants of VDP. To realize the first layer in a secure fashion, a common way seems plausible is to protect both real-valued data matrix and binarized weight matrix via Arithmetic sharing, and then perform VDP on Arithmetic-shared data. However, protecting the weights as Arithmetic shares would substantially exaggerates the bandwidth to transmit the model and wastes BNN's advancement. Besides, the multiplications over Arithmetic shares require the assistant of Beaver's multiplication triples [13], and generating the disposable triples incurs intensive bandwidth costs that scale linear with the number of triples. As the bandwidth is the bottleneck at the edge, overwhelming amount of communication leads to significantly slippery slope on the overall performance and additional charges by cellar network service provider.

To minimize the overall bandwidth costs at the edge, we craft the secure Boolean-Arithmetic-VDP function ($\mathsf{SecBAVDP}$) allowing for direct multiplication on mixed share representations. It takes as input the Boolean shares $[\![\mathbf{w}^b]\!]$ of the binary weight vector $\mathbf{w} \in \{0,1\}^n$ and the Arithmetic shares $\langle \mathbf{x} \rangle^A$ of the real-valued input vector $\mathbf{x} \in \mathbb{Z}^n$ (normally integers for image classification), and outputs the Arithmetic-shared result of $\mathsf{VDP}(\mathbf{x}, \mathbf{w}^b)$ as $\langle z \rangle^A$. Our main observation is that, the element-wise multiplication $\langle z_k \rangle^A = [\![w_k]\!] \cdot \langle x_k \rangle^A$ on each Boolean-shared weight and Arithmetic-shared input can be expressed as $\langle z_k \rangle^A = ([\![w_k^b]\!]_0 \oplus [\![w_k^b]\!]_1) \cdot (\langle x_k \rangle_0^A + \langle x_k \rangle_1^A)$. Such an expression can be efficiently undertook by two executions of COT_ℓ corresponding to the correlation functions f_Δ and g_Δ, respectively. The first COT_ℓ execution carries out $\langle u_k \rangle^A = \langle ([\![w_k]\!]_0 \oplus [\![w_k]\!]_1) \cdot \langle x_k \rangle_0^A \rangle^A$, where S_0 acts as the sender and S_1 acts as the receiver. By treating $[\![w_k]\!]_1$ as the choice bit $\mathfrak{b}_u$, the above logic can be rewritten as $\mathfrak{b}_u \cdot (\neg [\![w_k^b]\!]_0 \cdot \langle x_k \rangle_0^A - [\![w_k^b]\!]_0 \cdot \langle x_k \rangle_0^A)$ performed by COT_ℓ, and the $[\![w_k^b]\!]_0 \cdot \langle x_k \rangle_0^A$ calculated by S_0 at local. Similarly, the second COT_ℓ execution carries out $\langle v_k \rangle^A = \langle ([\![w_k]\!]_0 \oplus [\![w_k]\!]_1) \cdot \langle x_k \rangle_1^A \rangle^A$, where S_1 acts as the sender and S_0 acts as the receiver, and $[\![w_k]\!]_0$ is the choice bit. The details of the proposed $\mathsf{SecBAVDP}$ function are given in Fig. 4. Proof of correctness is given in Appendix A.1.

To perform the secure first layer VDP, we encode the weight vector $\mathbf{w}$ as a tuple $({}^+\mathbf{w}^b, {}^-\mathbf{w}^b)$, where ${}^+\mathbf{w}^b, {}^-\mathbf{w}^b \in \{0,1\}^n$. That is, when an element $w = +1$, the corresponding tuple is ${}^+w \leftarrow 1$ and ${}^-w \leftarrow 0$; while when $w = -1$, it is encoded as ${}^+w \leftarrow 0$ and ${}^-w \leftarrow 1$. As specified in Fig. 5, the secure first layer VDP function ($\mathsf{Sec1VDP}$) executes two times of the $\mathsf{SecBAVDP}$ function to obtain ${}^+z = \mathsf{VDP}({}^+\mathbf{w}, \mathbf{x})$ and ${}^-z = \mathsf{VDP}({}^-\mathbf{w}, \mathbf{x})$. The final output is computed as $z = {}^+z - {}^-z$.

Input: Arithmetic shares of integer input vector $\mathbf{x} \in \mathbb{Z}^n$, Boolean shares of $(+,-)$ binary weight vectors ${}^+\mathbf{w}^b \in 1^n$ and ${}^-\mathbf{w}^b \in 0^n$.
Output: Arithmetic shares of $z = \mathsf{VDP}(\mathbf{w}, \mathbf{x})$.

1. S_0 and S_1 run to obtain $\langle {}^+z \rangle_i^A = \mathsf{SecBAVDP}([\![{}^+\mathbf{w}^b]\!], \langle \mathbf{x} \rangle^A)$.
2. S_0 and S_1 run to obtain $\langle {}^-z \rangle_i^A = \mathsf{SecBAVDP}([\![{}^-\mathbf{w}^b]\!], \langle \mathbf{x} \rangle^A)$.
3. S_i computes $\langle z \rangle_i^A = \langle {}^+z \rangle_i^A - \langle {}^-z \rangle_i^A$.

Fig. 5. The secure first layer VDP function $\mathsf{Sec1VDP}(\cdot,\cdot)$.

5.2 Secure Batch Normalization and Binary Activation

Rationale. Batch normalization and binary activation are usually applied as a combination on each linear layer, directly following the linear transformation. Only except the output layer, the prediction results are the output from the batch normalization without binary activation. At a high level, the combination of such two functions proceeds the functionality $a^b = \mathsf{sign}(\epsilon_1 \cdot a + \epsilon_2)$, where ϵ_1, ϵ_2 are derived from the running mean μ, the running variance δ of training dataset, and the trained the scale γ and the shift β as defined in [25]. During our model training procedure over plaintext, we observe that the trained ϵ_1, ϵ_2 are real-valued, where their integer parts before radix point are usually very small (i.e., 0 or 1) and the the fractional parts can last for a few digits (e.g., 10 digits) with 5 more significant figures. To handle the real-valued numbers in secure computation, a common way is to scale the ϵ_1, ϵ_2 to integers with a certain precision factor 2^q, followed by a ring conversion applied on secret-shared a. Such a conversion is normally scaling up the $\langle a \rangle^A \in \mathbb{Z}_{2^\ell}$ to $\langle a' \rangle^A \in \mathbb{Z}_{2^\kappa}$ where $\kappa > \ell + q$. After sharing the ϵ_1, ϵ_2 in $\mathbb{Z}_{2^\kappa}$ as $\langle \epsilon_1 \rangle^A, \langle \epsilon_2 \rangle^A \in \mathbb{Z}_{2^\kappa}$, the computation $\langle y \rangle^A = \langle \epsilon_1 \rangle^A \cdot \langle a' \rangle^A + \langle \epsilon_2 \rangle^A$ can be securely carried out $\in \mathbb{Z}_{2^\kappa}$. And the $\mathsf{sign}(\langle y \rangle^A)$ can be securely realized via a most significant bit (MSB) extraction.

However, the limitations of such common way are two-fold. First, an additional ring conversion operation has to be applied on each neuron leading to more computational costs. Second, the enlarged ring $\mathbb{Z}_{2^\kappa}$ leads to a more complex bitwise MSB extraction. In general, this MSB extraction operation follows the bit extraction protocol in [44]. It firstly decomposes the $\langle y \rangle^A \in \mathbb{Z}_{2^\kappa}$ to its Boolean-shared binary strings. Then it applies a bitwise full adder logic to obtain the shared carry bit string whose highest bit is exactly the MSB of y. The non-local operations require the interaction between the two edge nodes with the complexity scales linear with the length of the bit string (i.e., κ). So, a larger ring size leads to more bandwidth costs which is undesired at the edge.

To this end, we propose two secure functions: 1) the secure normalized binary activation function (SecNBA) for the first layer and hidden layers; and 2) the secure batch normalization function (SecBN) in the output layer.

Secure Normalized Binary Activation. The SecBNA function combines the secure batch normalization and the secure binary activation. We transform its functionality into a simpler problem through

$$a^b = \mathsf{sign}(\epsilon_1) * \mathsf{sign}(a + \epsilon) \tag{1}$$

, where $\epsilon = \epsilon_2/\epsilon_1$, and the multiplication between the sign bits can be carried out via XNOR operation. Through our careful examination, we observe that $\epsilon = \epsilon_2/\epsilon_1$ is real-valued number with large integer parts. We thus quantize ϵ directly as integer and share it in $\mathbb{Z}_{2^\ell}$, so as to circumvent the conversion between different rings. It is noteworthy that our trained parameter ϵ_1 in batch normalization can be a negative value, and thus its sign bit $\mathsf{sign}(\epsilon_1)$ can not be ignored. We further denote it as $\zeta = \mathsf{sign}(\epsilon_1)$ and $z = \mathsf{sign}(a + \epsilon)$, and the functionality of binary normalized activation becomes $a^b = \mathsf{XNOR}(\zeta, z)$. Since ζ and ϵ are independent of inference input, they thus can be pre-computed by the model owner.

Input: Arithmetic shares of integer feature $a \in \mathbb{Z}$, Arithmetic shares of preprocessed parameter $\epsilon \in \mathbb{Z}$, Boolean shares of preprocessed parameter $\zeta \in \{0,1\}$.
Output: Boolean shares of binarized activation $a^b \in \{0,1\}$.

1. S_i calculates $\langle y \rangle_i^A = \langle a \rangle_i^A + \langle \epsilon \rangle_i^A$.
2. S_i decomposes $\langle y \rangle_i^A \in \mathbb{Z}_{2^\ell}$ as a bit string $\langle y_k \rangle_i^A$, where $k \in [1, \ell]$.
3. To extract the MSB of the $\langle y \rangle_i^A$, S_i proceeds **the** $\mathsf{MSB}(\cdot)$ **gadget** as follows:
   (a) For each $k \in [1, \ell]$: S_0 sets $\llbracket w_k \rrbracket_0 = \langle y_k \rangle_0^A$, $\llbracket p_k \rrbracket_0 = \langle y_k \rangle_0^A$, and $\llbracket q_k \rrbracket_0 = 0$, and S_1 sets $\llbracket w_k \rrbracket_1 = \langle y_k \rangle_1^A$, $\llbracket p_k \rrbracket_1 = 0$, and $\llbracket q_k \rrbracket_1 = \langle y_k \rangle_1^A$;
   (b) S_0 and S_1 compute $\llbracket d_k \rrbracket_i = \llbracket p_k \rrbracket \cdot \llbracket q_k \rrbracket$ in a batch to reduce the number of rounds of interaction; S_i sets $\llbracket c_1 \rrbracket_i = \llbracket d_1 \rrbracket_i$.
   (c) For $k \in [2, \ell - 1]$: S_i computes $\llbracket d_k \rrbracket_i = \llbracket d_k \rrbracket_i + i$ at local;
   (d) S_0 and S_1 compute $\llbracket e_k \rrbracket_i = \llbracket w_k \rrbracket \cdot \llbracket c_{k-1} \rrbracket + i$, and $\llbracket c_k \rrbracket_i = \llbracket e_k \rrbracket \cdot \llbracket d_k \rrbracket + i$.
   (e) S_i computes $\llbracket y_\ell \rrbracket_i = \llbracket w_\ell \rrbracket_i + \llbracket c_{\ell-1} \rrbracket_i$, and sets the $\llbracket y_\ell \rrbracket_i$ as the MSB of a.
4. To convert the MSB to the sign bit of a, S_i sets $\llbracket z \rrbracket_i = \llbracket y_\ell \rrbracket_i + i$.
5. S_i calculates XNOR at local $\llbracket a^b \rrbracket = \llbracket z \rrbracket_i + \llbracket \zeta \rrbracket_i + i$.

Fig. 6. The secure binary normalized activation function $\mathsf{SecNBA}(\cdot,\cdot,\cdot)$.

Fig. 6 details the secure realization of the SecNBA function following the eq. 1. It takes as input the Arithmetic shares of the feature $\langle a \rangle^A$ that outputted from the linear transformation, and outputs the Boolean shares of binarized activation $\llbracket a^b \rrbracket$. The computation comprises three atomic operations: 1) applying the shares of the preprocessed $\langle \epsilon \rangle^A$ to the shared feature $\langle a \rangle^A$ as $\langle y \rangle^A$ in step 1; 2) the extraction of the Boolean-shared sign bit $\llbracket z \rrbracket$ based on the MSB of $\langle y \rangle^A$ in steps 2,3; and 3) the local XNOR operation over sign bit and shared $\llbracket \zeta \rrbracket$ in step 4. Here, to extract the MSB of secret shares, the edge nodes S_0 and S_1 jointly execute the $\mathsf{MSB}(\langle a \rangle^A)$ gadget to obtain their shares of MSB $\llbracket y_l \rrbracket_i$. This gadget follows the bit extraction protocol in [44], which is able to efficiently extract the MSB of the Arithmetic-shared values and produce a Boolean-shared MSB. Since the MSB is 0 of non-negative values and 1 of the negative values, which is exactly the one's complement of a given sign bit, each node S_i then performs logical negation on each $\llbracket y_l \rrbracket_i$ at local to obtain the shared sign bit $\llbracket z \rrbracket_i$.

Input: Arithmetic shares $\mathbb{Z}_{2^\kappa}$ of integer activation $a \in \mathbb{Z}$, Arithmetic shares $\mathbb{Z}_{2^\kappa}$ of the preprocessed parameters $\epsilon_1, \epsilon_2 \in \mathbb{Z}_q$.
Output: Arithmetic shares of normalized activation $z \in \mathbb{Z}$.

1. To normalize the activation, S_0 and S_1 compute $\langle z \rangle_i^A = \langle \epsilon_1 \rangle^A \cdot \langle a_q \rangle^A + \langle \epsilon_2 \rangle_i^A$.

Fig. 7. The secure batch normalization function $\mathsf{SecBN}(\cdot,\cdot,\cdot)$.

Secure Batch Normalization for Output Layer. The secure batch normalization function (SecBN) is applied right after the secure linear transformations in the output layer. As presented in Fig. 7, it takes as input the Arithmetic shares of activation $\langle a \rangle^A$ outputted from linear transformation (i.e., $\mathsf{SecBVDP}$) and shares of two parameters $\langle \epsilon_1 \rangle^A, \langle \epsilon_2 \rangle^A$, and outputs the Arithmetic shares of the normalized activation $\langle z \rangle^A$. Here, the $\mathsf{B2A}$ gadget, i.e., the step 2 in the

SecBVDP, performs the conversion from over $\mathbb{Z}_2$ to $\mathbb{Z}_{2^\kappa}$, and thus the output of SecBVDP is a feature already secret-shared in $\mathbb{Z}_{2^\kappa}$. Note that this ring conversion operation will not affect the following binary activation, as the output of SecBN is the shared inference result. The parameters ϵ_1, ϵ_2 are already enlarged during preprocessing, and secret shared in $\in \mathbb{Z}_{2^\kappa}$. S_0 and S_1 apply a multiplication over shares $\langle \epsilon_1 \rangle^A \cdot \langle a \rangle$ followed by a local addition with $\langle \epsilon_2 \rangle^A$ to obtain $\langle z \rangle$.

Input: Boolean shares of m-number of n-dimensional binary activation vectors $\mathbf{a}_1^b, ..., \mathbf{a}_m^b \in \{0,1\}^n$, where the dimension n matches the size of pooling window.
Output: Boolean shares of m-number of binary pooling result $z_1^b, ..., z_m^b \in \{0,1\}$.

1. For each $t \in [1, m]$, S_i proceeds the following steps to determine the shares of maximum element in $[\![\mathbf{a}_t^b]\!]_i = ([\![a_{t,1}^b]\!]_i, ..., [\![a_{t,n}^b]\!]_i)$:
2. For all $k \in [1, n]$: S_i computes $[\![a_k^b]\!]_i = [\![a_k^b]\!]_i \oplus i$.
3. S_i initializes $[\![c_t^b]\!]_i = [\![a_{t,1}^b]\!]_i$.
4. For each $k \in [2, n]$: S_0 and S_1 compute $[\![c_t^b]\!]_i = [\![c_t^b]\!] \wedge [\![a_{t,k}^b]\!]$.
5. Then, S_i computes $[\![z_t^b]\!]_i = [\![c_t^b]\!]_i \oplus i$.

Fig. 8. The secure max pooling function SecMP$(\cdot)$.

5.3 Secure Binary Max Pooling Layer

The secure binary max pooling layer (SMP) is used to obtain the maximum values among the secret-shared binary activations within a certain sliding window. The number of n binary activations within the window can be denoted as a n-dimensional binary activation vector $\mathbf{a}^b = (a_1^b, ..., a_n^b) \in \{0,1\}^n$. And we assume that there are overall m-number of vectors as $\mathbf{a}_1^b, ..., \mathbf{a}_m^b$. The MP layer is normally applied right after a secure linear layer. We make an observation that its functionality over plaintext $\mathbf{a}^b$ can be realized as the bitwise-OR operation on all bits of the vectors, i.e., the maximum value is $z^b = a_1^b \vee a_2^b \vee ... \vee a_n^b$, , so as to find if $\mathbf{a}^b$ constitutes with any "1" bit. However, when realizing functionality in a secure manner, the key takeaway is to achieve obliviousness, i.e., for every step in a certain computation, both edge nodes have to proceed equivalent operations. Through carefully examination, we transform the logic to $z^b = \neg(\neg a_1^b \wedge \neg a_2^b \wedge ... \wedge \neg a_n^b)$ which is more compatible with our secret sharing based realization.

With this philosophy in mind, we present in Fig. 8 the proposed secure max pooling function (i.e., SecMP) design specialized for the two edge nodes case as the main building block of the MP layer. It takes as input the Boolean shares of m-number of n-dimensional binary activation vectors $[\![\mathbf{a}_t^b]\!]$, determines the maximum element for each vector, and outputs their Boolean shares as the pooling result $[\![z_t^b]\!]$, where $t \in [1, m]$. For each shared vector $[\![\mathbf{a}_t^b]\!]$, each edge node S_i for $i \in \{0,1\}$ computes the bitwise-XOR on its identifier i and its shares $[\![a_{t,1}^b]\!]_i$ of all bits in $[\![\mathbf{a}_t^b]\!]$ at local. S_i then initializes a temporary variable $[\![c_t^b]\!]_i$ to

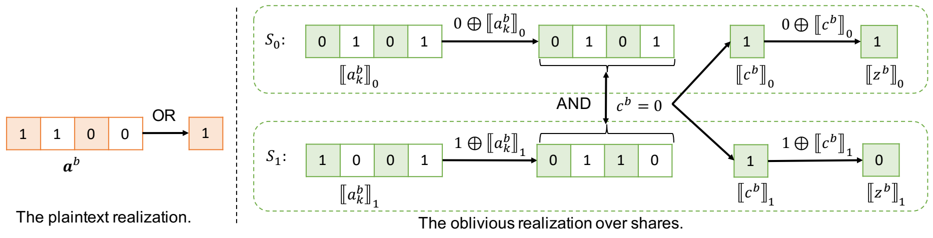

Fig. 9. The oblivious realization of the secure max pooling function.

store the shares of first bit $[\![a_{t,1}^b]\!]_i$, respectively. Afterwards, S_0 and S_1 perform AND over all of its shares, i.e., $[\![c_t^b]\!]_i = [\![a_{t,1}]\!]_i \wedge ... \wedge [\![a_{t,n}]\!]_i$. Each node sets its output share $[\![z_t^b]\!]_i$ as its identifier i XOR with $[\![c_t^b]\!]_i$. We emphasis that all operations performed by the two nodes are identical, and thus endowing our SecMP function obliviousness. Appendix A.2 provides the proof of correctness of the SecMP function.

For the purpose of easier understanding, Fig. 9 provides a concrete example of the secure max pooling function over a 4-dimensional binary activation vector $\mathbf{a}^b = (1, 1, 0, 0)$. It illustrates the equivalence between the plaintext realization based on bitwise OR and our proposed oblivious realization over secret shared data. As shown, the plaintext max pooling function can be simply realized with bitwise-OR operation over every bit of a^b, so as to obtain the maximum bit, i.e., "1" in our example. Correspondingly, its secure version is obliviously proceeded by the two edge nodes S_i for $i \in \{0, 1\}$, who locally performs bitwise-XOR on its identifier i and each bit of its share $[\![z^b]\!]_i$, They then jointly compute the AND over all the resulting bits as $[\![c^b]\!]_i$. Finally, they compute bitwise-XOR on i and $[\![c^b]\!]_i$ as its final output $[\![z^b]\!]_i$. The reconstructed output $[\![z^b]\!] = [\![z^b]\!]_0 \oplus [\![z^b]\!]_1 = 1$ is the same as the result given in clear value.

5.4 Secure Binarized Neural Network Inference Protocol

Given above layer functions, we now describe our secure binarized neural network inference protocol ϕ in the following section. It comprises two phases: the preprocessing phase performed by each entity individually, and the secure inference phase jointly carried out by the two non-colluding edge nodes.

Preprocessing Phase. During the preprocessing phase, the mobile user, the model provider, and edge nodes independently compute data that can be used during the execution of secure inference. The mobile user holding a task-specific raw input (e.g., an image for image classification) produces the real-valued (c-)matrix(-ces) $X \in \mathbb{R}^{n_{in} * m_{in}}$ representing such an input, and produces the Arithmetic shares of the matrix(-ces). He then deploys the corresponding shares to each edge nodes S_0 and S_1, respectively. S_0 and S_1 will pad these shares with 0 to be fit with the weight size, and partition them as number of s vectors $\langle \mathbf{x} \rangle^A = (\langle x_1 \rangle^A, ..., \langle x_n \rangle^A)$ based on the sliding window n (a hyper-parameter), where $s = (n_{in} * m_{in})/n$. Besides, they will also prepare multiplication triples and Boolean AND triples during vacant time.

```
Secure BNN Inference Phase of Protocol $\phi$:
 1: for each layer $j \in [1, L-1]$ do
 2:     if $j = 1$, first layer (SCONV): then
 3:         for each channel $c$ and window $s$ do $S_0$ and $S_1$ run $\langle(a)^1_{s,c}\rangle^A \leftarrow$ Sec1VDP($[\![(^+\mathbf{w}^b)^1_{s,c}]\!], [\![(^-\mathbf{w}^b)^1_{s,c}]\!], \langle(\mathbf{x})_{s,c}\rangle^A$) to get their shares of features.
 4:         end for
 5:         $S_i$ sums the features of all channels $\langle(a)^1_s\rangle^A = \sum_{\hat{c}=1}^{c}\langle(a)^1_{s,c}\rangle^A$ as the feature of $s$-th neuron.
 6:     else if $j \in [2, L]$, remaining layers (SFC): then
 7:         $S_i$ flattens its shares of the feature matrix as a vector $\langle\mathbf{a}\rangle^A_i$.
 8:         for each element $(w^b_{k'})^j, k' \in [1, n']$ of the weight vector $(\mathbf{w}^b)^j$ do
 9:             $S_0$ and $S_1$ run to get their shares of neuron $k$ feature $\langle(a_k)^j\rangle^A \leftarrow$ SecBVDP($[\![(\mathbf{a}^b)^{j-1}]\!], [\![(w^b_k)^j]\!]$), where $k \in [1, n]$.
10:         end for
11:         if $j \neq L$ then $S_0$ and $S_1$ proceed to obtain their shares of binarized activations $[\![(a^b)^j_k]\!] \leftarrow$ SecBNA($\langle(\bar{a})^j_k\rangle^A, \langle\epsilon\rangle^A, [\![\zeta]\!]$) of neuron $k$.
12:         else$S_0$ and $S_1$ proceed to obtain their shares of normalized neuron $k$ feature $\langle(\bar{a})^j_k\rangle^A \leftarrow$ SecBN($\langle(a)^j_s\rangle^A, \langle\epsilon_1\rangle^A, \langle\epsilon_2\rangle^A$).
13:         end if
14:     end if
15:     if apply the SMP layer then $S_0$ and $S_1$ run to get the shares of each neuron $k$ feature $[\![(a^b_k)^j]\!] \leftarrow$ SecMP($[\![(\mathbf{a}^b_k)^{j-1}]\!]$), where $k \in [1, m]$.
16:     end if
17: end for
```

Fig. 10. Secure BNN inference phase

For an L-layer BNN model, the model provider firstly transforms the binarized weight matrix(-ces) corresponding to each layer j to the weight vector(-s) $\mathbf{w}_j \in \{-1, +1\}^n$, where $j \in [1, L]$. He then encodes the weight vector of the first layer $\mathbf{w}_1$ to a tuple $(^+\mathbf{w}^b_1, {}^-\mathbf{w}^b_1)$ as above mentioned. The encoded weights are prepared for our proposed efficient first linear layer VDP calculation. For the weight vectors belong to the hidden layers, the model provider encodes them based on their sign, i.e., $+1 \leftarrow 1$ and $-1 \leftarrow 0$. For secure batch normalization, the model provider derives through $\epsilon_1 = \frac{\gamma}{\delta}$, $\epsilon_2 = \beta - \frac{\gamma \cdot \mu}{\delta}$, $\epsilon = \frac{\epsilon_2}{\epsilon_1} = \frac{\delta\beta}{\gamma} - \mu$, where the running mean μ, the running variance δ, the scale β, and the shift γ are defined in [25]. To this end, the model provider produces the Boolean shares of all weight vectors and the Arithmetic shares of batch normalization parameters, and then deploys the corresponding shares to the edge nodes S_0 and S_1 for coordinate processing. Note that the parameters of number of layers L, sliding window sizes n, pooling window size m, strides (normally is 1), channels c are assumed as hyper-parameters that can be deployed in clear (like in [8, 30].

Secure BNN Inference Phase. During the inference phase, the two non-colluding edge nodes S_0 and S_1 collaboratively proceed the BNN inference task on given secret-shared user input vector $\langle\mathbf{x}\rangle^A$ and model weight vectors $\langle\mathbf{w}_j\rangle^A$ for each layer $j \in [1, L]$ in a privacy assured manner. As the neural network can have distinct architectures assembled with layer functions, we here in this section takes one typical architecture as an example for demonstration purpose.

Fig. 10 depicts the secure BNN inference phase that modularly composes the above introduced layer functions performed on secret-shared data. It comprises the SCONV layer as the first layer, and $L-1$ number of the SFC layer, and can apply the SMP layer after each linear layer. For each linear layer, the SecBN and SecBA functions are applied followed by the linear transformation. For the SCONV layer, the edge nodes S_0 and S_1 slides each window s with size n on the shares of input matrix, and get an input vector $(\mathbf{x})_{s,c}$ within the window for each channel c. They then execute the Sec1VDP layer function on the shared and encoded weight vectors ${}^{+}\mathbf{w^b}_{s,c}$, ${}^{-}\mathbf{w^b}_{s,c}$ and $(\mathbf{x})_{s,c}$, to obtain the features for each channel. Such features are then summed across channels as $(a)_s$ to reduce the channels from c to 1. Suppose $j-1$ layer has n neurons and j layer has n' neurons. For each SFC layer, the edge nodes execute the SecBVDP on the activations in the $j-1$ layer and the weights in the j layer to get the feature $(a_k)^j$ for each neuron in the j layer. For every neuron's feature, they apply the SecNBA function to obtain the normalized and binarized activation $(a^b)_j$. All binarized activation in the j layer will be submitted to the $j+1$ layer as input. Besides, they can apply the SMP layer to reduce the number of neurons through execution of the SecMP function on every pooling window with size m. To this end, they obtain the shares of inference result. They then send back the shares of result to the mobile user who can reconstruct to get the prediction.

5.5 Security Guarantee

For our secure BNN inference protocol ϕ, we define security based on the *Universally Composable* (UC) security framework [14]. Under a general protocol composition operation (*universal composition*), the security of ϕ is preserved. Given a semi-honest *admissible adversary* $\mathcal{A}$ who can compromise at most one of the two non-colluding edge nodes S_0, S_1 and either the mobile user or the model owner. This reflects on the property that S_0, S_1 are non-colluding servers, i.e., if S_0 is compromised by $\mathcal{A}$, S_1 acts honestly; vice versa. Leia's protocol follows the security of the Arithmetic sharing [12], Boolean sharing [22] and OT [11]. Leia properly protects the user data, model, Beaver's Triples, and intermediate results outputted from layer functions as secret shares in $\mathbb{Z}_{2^\ell}$ and $\mathbb{Z}_2$, and the correlation functions $f_\triangle, g_\triangle$ are correlation-robust. Given above, we argue that ϕ UC-realizes an ideal functionality $\mathcal{F}$ against $\mathcal{A}$. The security captures the property that the only data learned by any compromised parties are their inputs and outputs from ϕ, but nothing about the data of the remaining honest parties.

6 Performance Evaluation

6.1 Implementation and Setup

We implement a prototype of Leia in Java. All experiments are executed on Raspberry Pi 4 Model B running Raspbian Linux 10 (buster) and equipped with Quad core Cortext-A72 (ARM v8) 64-bit SoC @ 1.5GHz processor, 4GB RAM, and gigabit ethernet. We choose FlexSC [40] as the code base of extended OTs [11] and implement our COT-based SecBAVDP function to compute the secure first linear layer. Regarding Arithmetic sharing, we set the size of the ring as $\mathbb{Z}_{2^{32}}$ for the first layer and output layer, and $\mathbb{Z}_{2^{16}}$ for the remaining hidden layers. The reported measurements make use of the MNIST and CIFAR-10 datasets, i.e., the two common-used classification benchmarks in prior work [10, 35]. We evaluate Leia on a variety of different BNN models, where the models M1 and M2 are trained on MNIST, and the models C1 and C2 are trained on CIFAR-10. (See Appendix. B for their detailed network architectures.) For training, we use PyTorch backend with standard BNN training algorithm [17]. Besides, we prepare and archive the disposal Beaver's triples in files (300 triples each file). The edge nodes can randomly select files to retrieve a set of triples upon the demand of computation, and then delete the corresponding files to ensure their randomness.

6.2 Evaluation

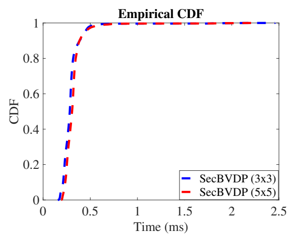

Fig. 11. Unit time of the SecBVDP function.

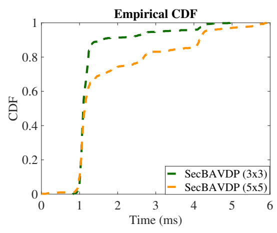

Fig. 12. Unit time of the SecBAVDP function.

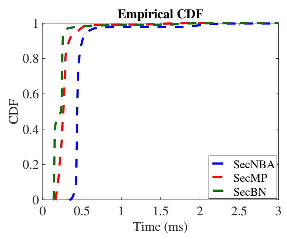

Fig. 13. Unit time of nonlinear functions.

Table 1. Computation performance of atomic layer functions (in s).

# inputs	SecBVDP	SecBVDP	SecBAVDP	SecBAVDP	SecNBA	SecBN	SecMP
	3×3 window	5×5 window	3×3 window	5×5 window			2×2 window
10^3	0.5	0.5	4.8	4.9	0.6	0.4	0.06
10^4	3.1	3.4	20.4	22.8	5.1	2.6	0.4
10^5	29.4	31.6	198.3	217.9	50.1	23.0	3.1

Microbenchmarks. We present performance benchmarks of secure layer functions as the basic building blocks used for secure BNN inference. For demonstration purpose, we choose 3×3 and 5×5 sliding windows to show the performance of the SecBVDP and the SecBAVDP functions, i.e., the secure VDP operations over 9-dimensional vectors and 25-dimensional vectors, respectively. These two window sizes are common-used and adapted to our CONV layer. Likewise, we employ the 2×2 window to demonstrate the performance of the SecMP function.

We summarize the computational cost of the proposed secure layer functions in Table 1. The time consumption of the SecBAVDP function consists of two parts: 1) the constant initialization cost of the COT protocol ($\sim$3s); and 2) the time and bandwidth to compute VDP over mixed share representations raising linearly with the number of calls. For the rest functions, their latencies ascend linearly in the growth of the number of executions yet with slight fluctuations. Besides, we grasp 10K executions of each secure layer function, and utilize the empirical cumulative distribution function (ECDF) to shed light on the distribution of their unit execution time. As depicted in Fig. 11, for overwhelming amount of executions, the unit executions of the SecBVDP function with 3×3 and 5×5 windows can be completed within 0.5ms. Fig. 12 displays the unit execution time of the SecBAVDP function without the aforementioned constant COT initialization cost. As shown, more than 90% executions take 1ms and 4.5ms for 3×3 and 5×5 windows, respectively. Fig. 13 exhibits the time costs of single executions of the non-linear functions, i.e., the SecNBA, SecBN, SecMP (with 2×2 window) functions. All three functions can be done within 1ms.

Table 2. Communication performance of atomic layer functions (in KB).

	SecBVDP 3×3 window	SecBVDP 5×5 window	SecBAVDP 3×3 window	SecBAVDP 5×5 window	SecNBA	SecBN	SecMP 2×2 window
Leia	0.3	0.8	16.0	17.3	0.2	0.03	0.01
Baseline	22.1	24.4	468.0	1257.4	25.9	78.3	20.9

The communication costs of the secure layer functions are reported and compared with the baseline in Table 2. We implement and evaluate the baseline based on GC with its free-XOR and half-AND optimizations, which realizes the equivalent functionality for each of the secure layer function. In general, Leia's realizations require $30 - 79\times$, and $150 - 2500\times$ less communication for the linear and non-linear functions than the corresponding GC-based realizations. The reported results testify that the prior constructions relying on GC [10,26,32,35] require a network environment with high bandwidth. However, they might not be fully suitable for our considered application scenario, i.e., the secure inference deployed at resources limited edge environment with cellular network.

In particular, for the COT-based SecBAVDP function, we emphasis that the adoption of such regime saves the overall bandwidth consumption at a system level. Such retrenchment includes the cost of protecting each weight element as 32-bit shares in $\mathbb{Z}_{2^{32}}$ to a tuple of 1-bit shares in $\mathbb{Z}_2$, and the cost of generation of multiplication triples in $\mathbb{Z}_{2^{32}}$. As shown by the empirical result, the GC-based realizations produce $30\times$ and $73\times$ bandwidth consumptions higher than the Leia's realizations for 9-dimensional and 25-dimensional vectors, respectively. We further report the bandwidth costs of the realizations based on multiplication triples as 270KB and 790KB, amounting to one magnitude larger than Leia's bandwidth.

Table 3. Performance of the SCONV function of first layer.

model	input $c_{in} \times n_{in} \times m_{in}$	kernel $c_o \times c_{in} \times s$	feature $c_o \times n_o \times m_o$	stride	padding	#SecBAVDP	time s	comm. MB
M2	1×28×28	16×1×5×5	16×24×24	1	-	1×18432	30	310
C1/C2	3×32×32	16×3×3×3	16×32×32	1	0	3×28800	46	490

Linear Layers. We report the performance of secure linear transformations (i.e., SCONV and SFC) below, which comprise the majority of Leia'S overall inference overhead.

Table 4. Performance of the SCONV function of hidden layers.

model	input	kernel	feature	stride	padding	#SecBVDP	time	comm.
	$c_{in} \times n_{in} \times m_{in}$	$c_o \times c_{in} \times s$	$c_o \times n_o \times m_o$				s	MB
M2	16×12×12	16×16×5×5	16×8×8	1	-	16×1024	0.9	12.5
C1	16×32×32	16×16×3×3	16×32×32	1	0	16×16384	6.2	63.3
C2	16×32×32	32×16×3×3	32×32×32	1	0	16×32768	14.5	126.6

Table 3 and Table 4 benchmark the performance of the SCONV layer function as the first layer and the hidden layer, respectively. The reported results are in line with our specified network architectures of M2, C1, and C2.[3] As C1 and C2 consist plenty of convolutional hidden layers, we choose to show the performance of their second layers (the most complicated hidden layers) for the ease of demonstration. The complexity of the SCONV layer function is determined by a set of parameters: 1) the number of input channels c_{in} and output channels c_o; 2) the dimensions of input image; 3) the kernel size (i.e., the sliding window size s), stride, and padding regime. These parameters directly reflect on the number of invocations of SecBAVDP/SecBVDP as shown. The key takeaway here is our runtime optimization of batch processing to amortize the overhead of executing SecBAVDP/SecBVDP. In detail, we flatten the input matrices across multiple channels yet within the same sliding window as a single vector, and conduct SecBAVDP/SecBVDP over it in a batch. We take as an example the complexity of C1's hidden SCONV layer reported in Table 4. It is proceeded in the batch integrating with 16-channel input matrices. As a result, the calls of SecBVDP (3×3 window) are reduced from 230400 to 16384, speeding up the time from 68s to 6.2s accordingly.

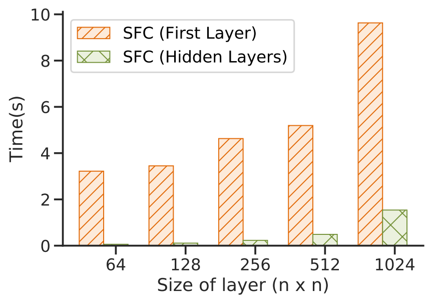

Fig. 14. Time of the $n \times n$ SFC layer.

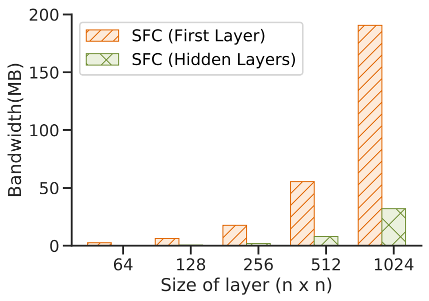

Fig. 15. Bandwidth of the $n \times n$ SFC layer.

Fig. 14 and Fig. 15 depict the computational and communication overheads of the SFC layer function as the first layer and hidden layers, respectively. They are evaluated over a series of $n \times n$ fully connected layers, i.e., both the input and weight are n-dimensional vectors. Followed by the growth of n, the time of the

[3] M1 consists of only fully connected layers.

hidden SFC layer ascends linearly attributed to our batch processing optimization, while the bandwidth ascends quadratically with the growth of dimension n. For the first SFC layer, the computational overhead is primarily dominated by the constant COT initialization time, and the bandwidth growths with the layer size.

Table 5. Performance summary of the trained networks.

network	time (s) edge	comm. (MB) edge	time (ms) [d] mobile user	time (s) [e] model owner	accuracy Leia	accuracy plaintext
M1 [a]	4.0	19.7	0.4	0.05	97%	97%
M2 [b]	37.4	328.1	0.4	0.5	99%	99%
C1 [c]	126.25	919.4	1.1	5.1	69%	65%
C2 [c]	199	1829.9	1.2	15.7	81%	88%

[a] 2 SFC+SecNBA, 1 SFC+SecBN.
[b] 2 SCONV+SecNBA, 1 SFC+SecNBA, 2 SMP, 1 SFC+SecBN.
[c] 9 SCONV+SecNBA, 3 SMP, 1 SFC+SecBN.
[d] Cost of generating shares of an image during preprocessing.
[e] One-time cost of generating shares of the model during preprocessing.

Leia's protocol. We evaluate the overall performance of Leia's cryptographic inference protocol on MNIST and CIFAR-10 datasets, and summarize the results in Table 5. The online phase of Leia is executed at the edge. The networks M1 and M2 for MNIST dataset are relatively simple, and Leia can produce high-quality prediction results within 4s and 37.4s, respectively. The more complex C1 and C2 for CIFAR-10 dataset involve 10 layers of linear transformation, and their executions require about 2min and 3.3min respectively. The workload of the mobile user is light, which confirms that Leia is amiable to the resource limited portable devices. The one-time overhead of the model owner is determined by the model size. Such cost dost not aggravate workload on model owner, as generating shares of the most complicated network C2 can be completed within 15.7s.

The effectiveness is demonstrated via the accuracy comparison between Leia's prediction results and the plaintext's results. For the M1 and M2 networks evaluated on MNIST dataset, Leia's prediction results are accurate as the plaintext (i.e., 97% and 99%, respectively). Besides, Leia achieves the accuracy of 69% and 81% for the C1 and C2 networks evaluated on CIFAR-10 dataset, amounting to slight accuracy impactions compared with the plaintext results. Such variations can be attributed to the quantization of batch normalization parameters.

References

1. AI at the Edge: The next frontier of the Internet of Things. Online at `https://iotbusinessnews.com/download/white-papers/AVNET-ai-at-the-edge-whitepaper.pdf`
2. Azure IoT Edge. Online at `https://azure.microsoft.com/en-au/services/iot-edge/`
3. Google Cloud AI. Online at `https://cloud.google.com/products/ai/`
4. Lambda@Edge. Online at `https://aws.amazon.com/lambda/edge/`

5. Machine learning and the future of mobile app development. Online at `https://heartbeat.fritz.ai/machine-learning-and-the-future-of-mobile-app-development-13dd2aeda533`
6. Machine learning on mobile: on the device or in the cloud? Online at `https://machinethink.net/blog/machine-learning-device-or-cloud/`
7. Pytorch for densenet. Online at `https://github.com/pytorch/vision/blob/master/torchvision/models/densenet.py`
8. Delphi: A cryptographic inference service for neural networks. In: Proc. of 29th USENIX Security (2020)
9. 104th United States Congress: Health Insurance Portability and Accountability Act of 1996 (HIPPA). online at `https://www.hhs.gov/hipaa/index.html` (1996)
10. Agrawal, N., Shahin Shamsabadi, A., Kusner, M.J., Gascón, A.: Quotient: two-party secure neural network training and prediction. In: Proc. of ACM CCS (2019)
11. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: Proc. of ACM CCS (2013)
12. Atallah, M., Bykova, M., Li, J., Frikken, K., Topkara, M.: Private collaborative forecasting and benchmarking. In: Proc. of WPES (2004)
13. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Proc. of Crypto (1991)
14. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2000)
15. Chen, W., Popa, R.A.: Metal: A metadata-hiding file sharing system. In: Proc. of NDSS (2020)
16. Courbariaux, M., Bengio, Y., David, J.P.: Binaryconnect: Training deep neural networks with binary weights during propagations. In: Proc. of NeuralIPS (2015)
17. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. arXiv preprint arXiv:1602.02830 (2016)
18. Demmler, D., Schneider, T., Zohner, M.: Aby-a framework for efficient mixed-protocol secure two-party computation. In: Proc. of NDSS (2015)
19. European Parliament and of the Council: The General Data Protection Regulation (GDPR). online at `http://data.europa.eu/eli/reg/2016/679/2016-05-04` (2016)
20. Fredrikson, M., Jha, S., Ristenpart, T.: Model inversion attacks that exploit confidence information and basic countermeasures. In: Proc. of ACM CCS (2015)
21. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: Proc. of ICML (2016)
22. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: Proc. of STOC (1987)
23. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. The Journal of Machine Learning Research **18**, 6869–6898 (2017)
24. Hwang, K., Sung, W.: Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In: 2014 IEEE Workshop on Signal Processing Systems (SiPS) (2014)
25. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)
26. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: Gazelle: A low latency framework for secure neural network inference. In: Proc. of 27th USENIX Security (2018)

27. Liang, S., Yin, S., Liu, L., Luk, W., Wei, S.: Fp-bnn: Binarized neural network on fpga. Neurocomputing **275**, 1072–1086 (2018)
28. Lin, X., Zhao, C., Pan, W.: Towards accurate binary convolutional neural network. In: Proc. of NeurIPS (2017)
29. Lindell, Y., Pinkas, B.: Privacy preserving data mining. Journal of cryptology **15**(3) (2002)
30. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via minionn transformations. In: Proc. of ACM CCS (2017)
31. Mohassel, P., Rindal, P.: Aby3: A mixed protocol framework for machine learning. In: Proc. of ACM CCS (2018)
32. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: Proc. of IEEE S&P (2017)
33. Rachuri, R., Suresh, A.: Trident: Efficient 4pc framework for privacy preserving machine learning (2020)
34. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Xnor-net: Imagenet classification using binary convolutional neural networks. In: Proc. of ECCV (2016)
35. Riazi, M.S., Samragh, M., Chen, H., Laine, K., Lauter, K., Koushanfar, F.: Xonn: Xnor-based oblivious deep neural network inference. In: Proc. of 28th USENIX Security (2019)
36. Riazi, M.S., Weinert, C., Tkachenko, O., Songhori, E.M., Schneider, T., Koushanfar, F.: Chameleon: A hybrid secure computation framework for machine learning applications. In: Proc. of AsiaCCS (2018)
37. Salimans, T., Kingma, D.P.: Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In: Proc. of NeurIPS (2016)
38. Shokri, R., Stronati, M., Song, C., Shmatikov, V.: Membership inference attacks against machine learning models. In: Proc. of IEEE S & P (2017)
39. Wagh, S., Gupta, D., Chandran, N.: Securenn: 3-party secure computation for neural network training. Proc. of PETS (2019)
40. Wang, X.: Flexsc. Online at `https://github.com/wangxiao1254/FlexSC` (2018)
41. Yao, A.C.C.: How to generate and exchange secrets. In: Proc. of IEEE FOCS (1986)
42. Yu, L., Liu, L., Pu, C., Gursoy, M.E., Truex, S.: Differentially private model publishing for deep learning. In: Proc. of S&P. IEEE (2019)
43. Zheng, W., Popa, R., Gonzalez, J.E., Stoica, I.: Helen: Maliciously secure coopetitive learning for linear models. In: Proc. of IEEE S&P (2019)
44. Zheng, Y., Duan, H., Wang, C.: Towards secure and efficient outsourcing of machine learning classification. In: Proc. of ESORICS. Springer (2019)
45. Zhou, L., Samavatian, M.H., Bacha, A., Majumdar, S., Teodorescu, R.: Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In: Proc. of ACM/IEEE Symposium on Edge Computing (2019)

A Proof of correctness

A.1 Proof of the SecBAVDP function

Lemma 1. *Let* $\mathbf{x}$ *and* $\mathbf{w}^b$ *be an integer and binary vectors, respectively. They are shared between parties* S_0 *and* S_1 *as Arithmetic shares* $\langle \mathbf{x} \rangle^A$ *and Boolean shares* $[\![\mathbf{w}^b]\!]$. *Given a secure COT protocol with the correlation-robust functions* $f_\triangle, g_\triangle$, *our two-party protocol* $\mathsf{SecBAVDP}(\langle \mathbf{x} \rangle^A,\ [\![\mathbf{w}^b]\!])$ *over mixed share representations correctly implements the Binary-Integer VDP functionality.*

Proof. The protocol $\mathsf{SecBAVDP}(\langle \mathbf{x} \rangle^A, [\![\mathbf{w}^b]\!])$ correctly implements the logic $\langle z \rangle^A = \sum_{k=1}^{n}([\![w_k^b]\!]_0 \oplus [\![w_k^b]\!]_1) \cdot (\langle x_k \rangle_0^A + \langle x_k \rangle_1^A)$ with two executions of COT_ℓ corresponding to the correlation functions f_Δ and g_Δ, respectively.

For the COT_ℓ with f_Δ, S_0 is the sender, and S_1 is the receiver. We show that the output of the k-th COT correctly carries out $\langle u_k \rangle^A = \langle ([\![w_k]\!]_0 \oplus [\![w_k]\!]_1) \cdot \langle x_k \rangle_0^A \rangle^A$. We have two cases:

1. If the choice bit $[\![w_k^b]\!]_1 = 0$:
   Note, $w_k^b = [\![w_k^b]\!]_0 \oplus 0 = [\![w_k^b]\!]_0$. Given the inputs $r_u \in_R \mathbb{Z}_{2^\ell}$, $f_\Delta(r_u)$ from S_0, the choice bit $\mathfrak{b}_u = [\![w_k^b]\!]_1 = 0$ from S_1. After execution of the COT, S_0 obtains $\langle u_k \rangle_0^A = -r_u$. Meanwhile, S_1 obtains $\langle u_k \rangle_1^A = m_{\mathfrak{b}_u} = m_0 = r_u + [\![w_k^b]\!]_0 \cdot \langle x_k \rangle_0 = r_u + w_k^b \cdot \langle x_k \rangle_0$ obliviously. Upon reconstruction, $u_k = w_k^b \cdot \langle x_k \rangle_0$.
2. If the choice bit $[\![w_k^b]\!]_1 = 1$:
   Note, $w_k^b = [\![w_k^b]\!]_0 \oplus 1 = \neg[\![w_k^b]\!]_0$. Given the inputs $r_u \in_R \mathbb{Z}_{2^\ell}$, $f_\Delta(r_u)$ from S_0, the choice bit $\mathfrak{b}_u = [\![w_k^b]\!]_1 = 1$ from S_1. After execution of the COT, S_0 obtains $\langle u_k \rangle_0^A = -r_u$. Meanwhile, S_1 obtains $\langle u_k \rangle_1^A = m_1 = r_u + \neg[\![w_k^b]\!]_0 \cdot \langle x_k \rangle_0 = r_u + w_k^b \cdot \langle x_k \rangle_0$ obliviously. Upon reconstruction, $u_k = w_k^b \cdot \langle x_k \rangle_0$.

The execution of the COT_ℓ with g_Δ performs in a similar way, where S_1 is the sender, and S_0 is the receiver. For the k-th COT, it takes as input $r_v \in_R \mathbb{Z}_{2^\ell}$, $g_\Delta(r_v)$ from S_1, and the choice bit $\mathfrak{b}_v = [\![w_k^b]\!]_0$ from S_0. Upon execution, S_1 always obtains $\langle v_k \rangle_1^A = -r_v$, and S_0 obtains $\langle v_k \rangle_0^A = m_0 = r_v + [\![w_k^b]\!]_1 \cdot \langle x \rangle_1^A$ or $\langle v_k \rangle_0^A = m_1 = r_v + \neg[\![w_k^b]\!]_1 \cdot \langle x \rangle_1^A$ obliviously. Upon reconstruction, $v_k = w_k^b \cdot \langle x \rangle_1^A$. Ultimately, $z_k = w_k^b \cdot (\langle x \rangle_0^A + \langle x \rangle_1^A) = w_k^b \cdot x_k$.

A.2 Proof of the SecMP function

Lemma 2. *Let $\mathbf{a}_1^b, ..., \mathbf{a}_m^b$ be m-number of n-dimensional binary activation vectors, shared between parties S_0, S_1 as Boolean shares $[\![\mathbf{a}_1^b]\!], ..., [\![\mathbf{a}_m^b]\!]$. Given the ring $\mathbb{Z}_2$, our two-party protocol* $\mathsf{SecMP}(\cdot)$ *correctly implements the max pooling functionality.*

Proof. For the ease of demonstration and without loss of generality, the following proof of correctness takes one binary activation vector $\mathbf{a}^b \in \{0,1\}^n$ as an example. The protocol implements the logic $z^b = a_1^b \vee a_2^b \vee ... \vee a_n^b$ to proceed the max pooling over binarized values, i.e., finding if the binary vector has "1" bit as mentioned above. Taken $z^b = x^b \vee y^b$ as an example, the bitwise-OR logic can be similarly reformulated as $z^b = \neg(\neg x^b \wedge \neg y^b)$. For the two-party protocol, each party S_i (for $i \in \{0,1\}$) holds its shares $[\![x^b]\!]_i, [\![y^b]\!]_i$, and attempts to obtain $[\![z^b]\!]_i$ as the result. The above logic can be correctly expressed as $[\![z^b]\!]_i = i \oplus ((i \oplus [\![x^b]\!]_i) \wedge (i \oplus [\![y^b]\!]_i))$, given the correctness of $\neg x^b = [\![x^b]\!]_0 \oplus \neg[\![x^b]\!]_1 = (0 \oplus [\![x^b]\!]_0) \oplus (1 \oplus [\![x^b]\!]_1)$. Note, all operations above are performed by each party without interaction.

B Model Architectures.

Table 6. Model architecture of M1.

layers	# SecBVDP/SecBAVDP	padding
FC (input: 784, output: 128) + BN + BA	256	-
FC (input: 128, output: 128) + BN + BA	128	-
FC (input: 128, output: 10) + BN	10	-

Table 7. Model architecture of M2.

layers	# SecBVDP/SecBAVDP	padding
CONV (input: 1 × 28 × 28, kernel: 1 × 16 × 5 × 5 feature: 16 × 24 × 24) + BN + BA	1×18432	-
MP (input: 16 × 24 × 24, window: 16 × 2 × 2 output: 16 × 12 × 12)	-	-
CONV (input: 16 × 12 × 12, kernel: 16 × 16 × 5 × 5 feature: 16 × 8 × 8) + BN + BA	16×1024	-
MP (input: 16 × 8 × 8, window: 16 × 2 × 2 output: 16 × 4 × 4) + BN + BA	-	-
FC (input: 256, output: 100) + BN + BA	100	-
FC (input: 100, output: 10) + BN	10	-

Table 8. Model architecture of C1.

layers	# SecBVDP/SecBAVDP	padding
CONV (input: 3 × 32 × 32, kernel: 3 × 16 × 3 × 3 feature: 16 × 32 × 32) + BN + BA	3×32768	0
CONV (input: 16 × 32 × 32, kernel: 16 × 16 × 3 × 3 feature: 16 × 32 × 32) + BN + BA	16×16384	0
CONV (input: 16 × 32 × 32, kernel: 16 × 16 × 3 × 3 feature: 16 × 32 × 32) + BN + BA	16×16384	0
MP (input: 16 × 32 × 32, window: 16 × 2 × 2 output: 16 × 16 × 16)	-	-
CONV (input: 16 × 16 × 16, kernel: 16 × 32 × 3 × 3 feature: 32 × 16 × 16) + BN + BA	16×8192	0
CONV (input: 16 × 16 × 16, kernel: 16 × 32 × 3 × 3 feature: 32 × 16 × 16) + BN + BA	16×8192	0
CONV (input: 16 × 16 × 16, kernel: 16 × 32 × 3 × 3 feature: 32 × 16 × 16) + BN + BA	16×8192	0
MP (input: 32 × 16 × 16, window: 32 × 2 × 2 output: 32 × 8 × 8)	-	-
CONV (input: 32 × 8 × 8, kernel: 32 × 48 × 3 × 3 feature: 48 × 6 × 6) + BN + BA	32×1728	-
CONV (input: 48 × 6 × 6, kernel: 48 × 48 × 3 × 3 feature: 48 × 4 × 4) + BN + BA	48×1728	-
CONV (input: 48 × 4 × 4, kernel: 48 × 64 × 3 × 3 feature: 64 × 2 × 2) + BN + BA	48×2304	-
MP (input: 64 × 2 × 2, window: 64 × 2 × 2 output: 64 × 1 × 1)	-	-
FC (input: 64, output: 10) + BN	10	-

Table 9. Model architecture of C2.

layers	# SecBVDP/SecBAVDP	padding
CONV (input: 3 × 32 × 32, kernel: 3 × 16 × 3 × 3 feature: 16 × 32 × 32) + BN + BA	3×32768	0
CONV (input: 16 × 32 × 32, kernel: 16 × 32 × 3 × 3 feature: 32 × 32 × 32) + BN + BA	16×32768	0
CONV (input: 32 × 32 × 32, kernel: 32 × 32 × 3 × 3 feature: 32 × 32 × 32) + BN + BA	32×32768	0
MP (input: 32 × 32 × 32, window: 32 × 2 × 2 output: 32 × 16 × 16)	-	-
CONV (input: 32 × 16 × 16, kernel: 32 × 48 × 3 × 3 feature: 48 × 16 × 16) + BN + BA	32×12288	0
CONV (input: 48 × 16 × 16, kernel: 48 × 64 × 3 × 3 feature: 64 × 16 × 16) + BN + BA	48×16384	0
CONV (input: 64 × 16 × 16, kernel: 64 × 80 × 3 × 3 feature: 80 × 16 × 16) + BN + BA	64×20480	0
MP (input: 80 × 16 × 16, window: 80 × 2 × 2 output: 80 × 8 × 8)	-	-
CONV (input: 80 × 8 × 8, kernel: 80 × 96 × 3 × 3 feature: 96 × 6 × 6) + BN + BA	80×3456	-
CONV (input: 96 × 6 × 6, kernel: 96 × 96 × 3 × 3 feature: 96 × 4 × 4) + BN + BA	96×1536	-
CONV (input: 96 × 4 × 4, kernel: 96 × 128 × 3 × 3 feature: 128 × 2 × 2) + BN + BA	96×512	-
MP (input: 128 × 2 × 2, window: 128 × 2 × 2 output: 128 × 1 × 1)	-	-
FC (input: 128, output: 10) + BN	10	-