FssNN: Communication-Efficient Secure Neural Network Training via Function Secret Sharing (Full Version)

Peng Yang[1], Zoe L. Jiang[1,2], Shiqi Gao[1], Jiehang Zhuang[1], Hongxiao Wang[3], Junbin Fang[4], Siuming Yiu[3], and Yulin Wu[1,2]

[1] Harbin Institute of Technology, ShenZhen, ShenZhen, China
{stuyangpeng,200111514,22S051007}@stu.hit.edu.cn
[2] Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies
{zoeljiang,wuyulin}@hit.edu.cn
[3] The University of Hong Kong, HKSAR, China
{hxwang,smyiu}@cs.hku.hk
[4] Jinan University, Guangzhou, China
tjunbinfang@jnu.edu.cn

Abstract. This Paper proposes FssNN, a communication-efficient secure two-party computation framework for evaluating privacy-preserving neural network via function secret sharing (FSS) in semi-honest adversary setting. In FssNN, two parties with input data in secret sharing form perform secure linear computations using additive secret haring and non-linear computations using FSS, and obtain secret shares of model parameters without disclosing their input data. To decrease communication cost, we split the protocol into online and offline phases where input-independent correlated randomness is generated in offline phase while only lightweight "non-cryptographic" computations are executed in online phase. Specifically, we propose BitXA to reduce online communication in linear computation, DCF to reduce key size of the FSS scheme used in offline phase for nonlinear computation. To further support neural network training, we enlarge the input size of neural network to 2^{32} via "MPC-friendly" PRG.
We implement the framework in Python and evaluate the end-to-end system for private training between two parties on standard neural networks. FssNN achieves on MNIST dataset an accuracy of 98.0%, with communication cost of 27.52GB and runtime of 0.23h per epoch in the LAN settings. That shows our work advances the state-of-the-art secure computation protocol for neural networks.

Keywords: Privacy-preserving neural network · Secure multi-party computation · Additive secret sharing · Function secret sharing.

1 Introduction

Machine learning techniques are widely used in practice to produce predictive models that are widely used in applications such as health-care prediction, financial services and policy making. Neural network provides a powerful method for machine learning training and inference. To improve accuracy and generalization ability of neural network model, it is desirable for multiple parties, each possessing a part of dataset, to combine their data to jointly train a model. However, these data are sensitive and cannot be revealed in the clear due to privacy concerns and regulations.

Privacy-preserving neural network (PPNN) is committed to addressing these concerns by leveraging cryptographic primitives, such as homomorphic encryption (HE) and secure multiparty computation (MPC). MPC-based PPNN provides a more concretely efficient solution by allowing different entities to train various models on their joint data without revealing any information beyond the output. Current works of this area are mainly based on two popular routes: i) MPC over Boolean circuits and ii) MPC over arithmetic circuits. The protocols evaluating Boolean circuit are built using Yao's garbled circuits (GC) [1] and result in constant-round solutions. Secret-sharing (SS)-based solutions [2] have been used for evaluating arithmetic circuits. Since PPNN involves many arithmetic operations and applies computations that cannot be succinctly represented by Boolean circuits, SS has got more attention although it requires a number of communication rounds linear in the multiplicative depth of the circuit.

Therefore, SS-based solutions still come at a steep performance overhead that may not be amiable for the real-world scenarios. During training, heavy cryptographic computations are required to conduct, imposing intensive computational and communication overheads. It is highly desirable to reduce the performance gap between secure neural network training and plaintext training for wider deployment of privacy-preserving technologies.

1.1 Related Works

In the past few years, PPNN raised much more attention, and has emerged as a flourishing research area. Many prior works [3,4,5,6,7,8,9] use a secure computation protocol based on secret sharing to compute the linear function (e.g. addition and multiplication) and garbled circuits, mix-protocol or homomorphic encryption to compute the non-linear function (e.g. comparison and activation function). They adopt offline-online computation model (aka preprocessing model [10]) to move a majority of the overheads to the offline phase, but still require a large number of communication rounds in the online phase.

The main source of inefficiency in prior implementations is that the bulk of computation for non-linear function (e.g. ReLU), which requires a large number of rounds of communication. Recently, Boyle et al. [11,12,13,14] proposed function secret sharing and applied it in secure computation protocol in the preprocessing model, which is useful for evaluating non-linear activation function with optimal online communication and round complexity. This motivated several works on PPNN based on function secret sharing, such as [15,16,17].

The state-of-the-art work [17], called AriaNN, proposes a low-interaction privacy-preserving framework for private neural network training on sensitive data. AiraNN designs semi-honest two-party computation protocol based on function secret sharing to implement an efficient online phase, and proposes optimized primitives for the building blocks of neural networks. However, this work still demands intensive workloads on the parties to conduct heavy cryptographic computations in online phase, and relies on trusted dealer to generate correlated randomness in offline phase. Therefore, it is still necessary to further reduce the communication complexity in the online phase and emulate the dealer via MPC, to enable practical and communication-efficient secure neural network training.

1.2 Our Contributions

We design and implement FssNN, a new two-party computation (2PC) framework for evaluating privacy-preserving neural network. In real-world applications, it can be used to enable two parties (data owner) to train a neural network models on their joint data by first secretly sharing their inputs, and can also be used in server-aided setting where the parties outsource the model training to two non-colluding servers. We adopt offline-online computation model where the time consuming computation of the correlated randomness is moved to offline phase. To further achieve optimal online communication complexity, we propose the hybrid method (combining additive secret sharing with FSS) to train linear layers of neural network via additive secret sharing and non-linear layers via FSS.

Our contributions can be summarized in the following three aspects:

- BitXA **protocol**. To multiply a bit and an n-bit integer for linear computation in online phase, we propose the BitXA protocol to perform the multiplication directly instead of converting the bit to integer first. As a result, the communication cost is reduced from $2n$ to $n+1$.
- **Distributed comparison function (DCF)**. To reduce key size of DCF for non-linear computation in offline phase, we propose an efficient DCF by designing a simpler and more compact key generation algorithm, and the key size is slightly smaller than [14]. The new DCF can be used to compute ReLU and its derivatives DReLU, which are the building blocks for neural networks and can be run with only one round of communication.
- **Distributed DCF key generation**. To enlarge input size of neural network training to 2^{32} and larger, we present the first MPC-friendly PRG-based distributed DCF key generation scheme, where previous constructions [18] are only for small input size (e.g., 2^{16} or smaller). Additionally, we presents a generic 2PC protocol to jointly emulate the trusted dealer in offline phase.

1.3 Organization

Following basic notations and some background on neural network and secure computation in Section 2, Section 3 presents a high-level overview of FssNN. Section 4 and 5 present efficient and secure computation protocol for linear layer (i.e., matrix multiplication and hadamard product) and non-linear layer (i.e., ReLU and its derivative DReLU). Specifically, Section 5.1 presents secure ReLU and DReLU protocol based on DCF, which is demonstrated in detail in Section 5.2, and Section 5.3 provides a construction for generating DCF key based on MPC-friendly PRG. Section 6.2 presents experimental results and analysis of our protocol. Finally, we conclude our paper in section 7.

2 Preliminaries

Notations We use arithmetic operations in the ring $\mathbb{Z}_{2^n}$, and we naturally identify elements of $\mathbb{Z}_{2^n}$ with their n-bit binary representation. Unless otherwise specified, we parse $x \in \{0,1\}^n$ as $x_{n-1}, \cdots, x_0$, where x_{n-1} is the most significant bit (MSB) and x_0 is the least significant bit (LSB). In this paper, we consider computations over finite bit unsigned and signed integers, denoted by $\mathbb{U}_{2^n}$ and $\mathbb{S}_{2^n}$ respectively, over n-bits. We note that $\mathbb{U}_{2^n} = \{0, \cdots, 2^n - 1\}$ is isomorphic to $\mathbb{Z}_{2^n}$. Moreover, $\mathbb{S}_{2^n} = \{-2^{n-1}, \cdots, 2^{n-1} - 1\}$ can be encoded into $\mathbb{Z}_{2^n}$ or $\mathbb{U}_{2^n}$ using 2's complement notation or mod 2^n operation. In addition, we define the conversion function between signed and unsigned numbers $\mathsf{S2U} : \mathbb{S}_{2^n} \to \mathbb{U}_{2^n}$ and $\mathsf{U2S} : \mathbb{U}_{2^n} \to \mathbb{S}_{2^n}$. Specifically, given $x \in \mathbb{S}_{2^n}$, there is $\mathsf{S2U}(x) = x \bmod N$, and given $x \in \mathbb{U}_{2^n}$, there is $\mathsf{U2S}(x) = x - \mathsf{MSB}(x) \cdot 2^n$, where $\mathsf{MSB}()$ means to find the most significant bit.

2.1 Neural Network

Given m training data samples $\mathbf{x}_i$ each containing d features and the corresponding output labels y_i, neural network is a computational process to learn a function g such that $g(\mathbf{x}_i) \approx y_i$. In neural network, g can be represented as a function of the coefficient matrix $\mathbf{W}$ and the input data $\mathbf{x}_i$, thus predicted output is represented as $\hat{y}_i = g(\mathbf{W}, \mathbf{x}_i)$. The phase to calculate the predicted output is called *forward propagation*, where comprises of a linear operation (e.g., matrix operations), followed by a (non-linear) activation function f. One of the most popular activation functions is the Rectified Linear Unit (ReLU).

To learn the weight $\mathbf{W}$, a cost function $C(\mathbf{W})$ that quantifies the error between predicted values $\hat{y}_i$ and actual values y_i is defined, and $\mathbf{W}$ is calculated and updated by the optimization $\mathsf{argmin}_{\mathbf{W}} C(\mathbf{W})$. The solution for this optimization problem can be computed by using stochastic gradient descent (SGD), which is an effective approximation algorithm for approaching a local minimum of a function step by step. The SGD algorithm works as follows: $\mathbf{W}$ is initialized as a vector of random values or all 0s. In each iteration, a sample $(\mathbf{x}_i, y_i)$ is selected randomly and a coefficient $\mathbf{W}$ is updated as: $\mathbf{W} := \mathbf{W} - \alpha \nabla C(\mathbf{W})$, where α is learning rate and $\nabla C(\mathbf{W})$ is the partial derivatives of the cost with respect to changes in weights. The phase to calculate the change $\alpha \nabla C(\mathbf{W})$ is called *backward propagation*, where error rates are feed back through a neural network to update weight.

In practice, instead of selecting one sample of data per iteration, a small batch of samples are selected randomly and $\mathbf{W}$ is updated by averaging the partial derivatives of all samples on the current $\mathbf{W}$. This is called a *mini-batch*

SGD, and its advantage is that it allows for the use of vectorization libraries to accelerate computation, resulting in a faster computation.

2.2 Additive Secret Sharing

In this paper, we consider additive secret sharing scheme under two-party setting. To additively share ($\mathsf{Shr}(\cdot)$) an n-bit value x, the first party P_0 sends a randomly sampled integer $\langle x\rangle_1^A = r \in \mathbb{Z}_{2^n}$ to the second party P_1, and keeps $\langle x\rangle_0^A = x - r \bmod 2^n$ as the its own share. To reconstruct ($\mathsf{Rec}(\cdot,\cdot)$) an additively shared value $\langle x\rangle^A$, P_i sends $\langle x\rangle_i^A$ to P_{1-i} who computes $\langle x\rangle_0^A + \langle x\rangle_1^A$, where $i = 0, 1$. In the following of this paper, denote values used in additive sharing by $\langle\cdot\rangle$ for short, as we mostly use additive sharing.

Additionally, consider Boolean secret sharing, which can be seen as additive secret sharing in $\mathbb{Z}_2$ and hence all the protocols discussed in additive secret sharing can carry over. In particular, the addition operation is replaced by XOR operation ($\oplus$) and multiplication by AND operation ($\mathsf{AND}(\cdot,\cdot)$). Denote party P_i's share in Boolean sharing by $\langle\cdot\rangle_i^B$.

Addition and Multiplication Assume P_i holds shared values $\langle x\rangle_i, \langle y\rangle_i$, it is easy to non-interactively add the shares by having P_i compute $\langle z\rangle_i = \langle x\rangle_i + \langle y\rangle_i \bmod 2^n$. In the following text, we omit the modular operation for simplicity. To perform multiplication, take the advantage of Beaver's precomputed multiplication triplet [19]. Assume that P_b already has shares $\langle a\rangle_i, \langle b\rangle_b, \langle c\rangle_b$ where a, b are uniformly random values in $\mathbb{Z}_{2^n}$ and $c = a \cdot b$. Then P_i locally computes $\langle e\rangle_i = \langle x\rangle_b - \langle a\rangle_i$ and $\langle f\rangle_b = \langle y\rangle_i - \langle b\rangle_i$. Both parties run $\mathsf{Rec}(\langle e\rangle_0, \langle e\rangle_1)$ and $\mathsf{Rec}(\langle f\rangle_0, \langle f\rangle_1)$ to get e, f, and P_i lets $\langle z\rangle_i = i \cdot e \cdot f + f \cdot \langle a\rangle_i + e \cdot \langle b\rangle_i + \langle c\rangle_i$.

Fixed-Point Arithmetic To support arithmetic operation over numbers with decimal points, use fixed-point representation to map rational numbers to $\mathbb{U}_{2^n}$ or $\mathbb{S}_{2^n}$. The fixed-point representation allows one to store decimal values with some approximation using n-bit integers. Consider the fixed-point addition and multiplication of two shared decimal numbers x and y. Fixed-point addition is a local operation where the corresponding shares of the operands are added separately by each party. Fixed-point multiplication requires "scale adjustment" to maintain the scale of the output instead of getting doubled for every multiplication performed. We implement this scale adjustment (or truncation) through a non-interactive "local truncation" procedure [3] where truncation is directly applied on the shares.

2.3 Function Secret Sharing

This subsection follows the definition of FSS from [14]. Intuitively, a two-party FSS scheme splits a function $f \in \mathcal{F}$ into two additive shares f_0, f_1, such that: (1) each f_b hides f; (2) for every input x, $f_0(x) + f_1(x) = f(x)$.

Definition 1. *(FSS: Syntax). A (2-party) function secret sharing (FSS) scheme is a pair of algorithms* $(\mathsf{Gen}, \mathsf{Eval})$ *such that:*

- $\mathsf{Gen}(1^\lambda, \hat{f})$ *is a PPT key generation algorithm that given* 1^λ *and* $\hat{f} \in \{0,1\}^*$ *(description of a function* $f : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$*) outputs a pair of keys* (k_0, k_1)*. We assume that* $\hat{f}$ *explicitly contains descriptions of input and output groups* $\mathbb{G}^{\mathsf{in}}, \mathbb{G}^{\mathsf{out}}$*.*
- $\mathsf{Eval}(b, k_b, x)$ *is a polynomial-time evaluation algorithm that given* $b \in \{0,1\}$ *(party index),* k_b *(key defining* $f_b : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$*) and* $x \in \mathbb{G}^{\mathsf{in}}$ *(input for* f_b*) outputs a group element* $y_b \in \mathbb{G}^{\mathsf{out}}$ *(the value of* $f_b(x)$*).*

Definition 2. *(FSS: Security). Let* $\mathcal{F} = \{f\}$ *be a function family. We say that* $(\mathsf{Gen}, \mathsf{Eval})$ *as in* **Definition 1** is an FSS scheme for $\mathcal{F}$ if it satisfies the following requirements:

- **Correctness**: For all $f : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}} \in \mathcal{F}$, and every $x \in \mathbb{G}^{\mathsf{in}}$, if $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f})$, then $\Pr[\mathsf{Eval}(0, k_0, x) + \mathsf{Eval}(1, k_1, x) = f(x)] = 1$.
- **Privacy**: For each $b \in \{0,1\}$ there is a PPT algorithm Sim_b (simulator), such that for every sequence $\{\hat{f}_i\}_{i \in \mathbb{N}}$ of polynomial-size function descriptions from $\mathcal{F}$ and polynomial-size input sequence x_b for f_b, the outputs of the following experiments Real and Ideal are computationally indistinguishable:
  - $\mathsf{Real}(1^\lambda) : (k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}_i)$; Output k_b.
  - $\mathsf{Ideal}(1^\lambda)$: Output $\mathsf{Sim}_b(1^\lambda)$.

Secure Computation via FSS Recent work of Boyle et al.[13,14] showed that FSS paradigm can be used to efficiently evaluate some function families in 2PC in the preprocessing model, where Gen and Eval correspond to the offline and online phase, respectively. Note that unlike in secret-sharing MPC, inputs and outputs in FSS are public whereas the function is secretly shared. As the parties cannot learn the values on any intermediate circuit wires, the scheme needs to mask inputs and outputs for each gate $g : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$. That is, the input is masked with r^{in} and the output with $\mathsf{r}^{\mathsf{out}}$. Then, to guarantee the correctness of computation, each gate g is replaced by an *offset function* $g^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(x) := g(x - \mathsf{r}^{\mathsf{in}}) + \mathsf{r}^{\mathsf{out}}$, where $\mathsf{r}^{\mathsf{in}} \in \mathbb{G}^{\mathsf{in}}$ and $\mathsf{r}^{\mathsf{out}} \in \mathbb{G}^{\mathsf{out}}$. The output of $g^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(x)$ is exactly the output of g masked with $\mathsf{r}^{\mathsf{out}}$. This step is repeated for each gate until both parties evaluate the last circuit gate. With the masked output of the last gate, P_0 and P_1 can recover the unmasked output (i.e., the final circuit output) without learning any intermediate wire values. For more information, refer to [14].

3 FssNN Overview

3.1 The FssNN Framework

Consider the two-party setting (denoted by P_b where $b \in \{0,1\}$ is party index, similarly hereinafter.) as shown in Fig. 1. Given the training dataset $D =$

$\{(\mathbf{x}_i, y_i)|i = 1, 2, \cdots, m\}$ which can be horizontally or vertically partitioned, FssNN works as follows: P_0 and P_1 hold the shares of each training sample, denoted by $(\langle \mathbf{x}_i \rangle_0, \langle y_i \rangle_0)$ and $(\langle \mathbf{x}_i \rangle_1, \langle y_i \rangle_1)$. Weight $\mathbf{W}$ is secretly shared between P_0 and P_1 by initializing $\langle \mathbf{W} \rangle_0$ and $\langle \mathbf{W} \rangle_1$ to be random locally; P_0 and P_1 compute gradient and update $\langle \mathbf{W} \rangle_0, \langle \mathbf{W} \rangle_1$ interactively via the SGD algorithm, which involves forward propagation and backward propagation. $\langle \mathbf{W} \rangle_0$ and $\langle \mathbf{W} \rangle_1$ are updated and remain secretly shared after each iteration of SGD, until the end when it is reconstructed.

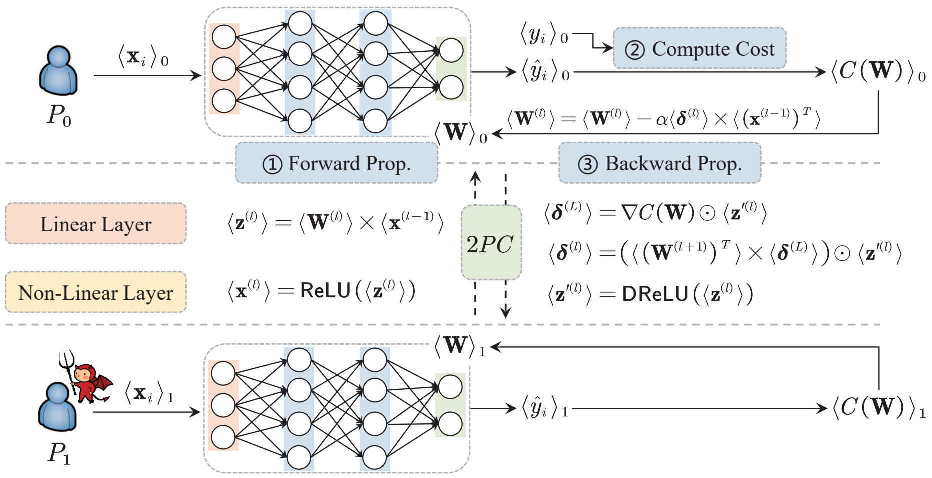
P_0
$\langle \mathbf{x}_i \rangle_0$
$\langle y_i \rangle_0$
② Compute Cost
$\langle \hat{y}_i \rangle_0$
$\langle C(\mathbf{W}) \rangle_0$
$\langle \mathbf{W}^{(l)} \rangle = \langle \mathbf{W}^{(l)} \rangle - \alpha \langle \boldsymbol{\delta}^{(l)} \rangle \times \langle (\mathbf{x}^{(l-1)})^T \rangle$
$\langle \mathbf{W} \rangle_0$
① Forward Prop.
③ Backward Prop.
Linear Layer
$\langle \mathbf{z}^{(l)} \rangle = \langle \mathbf{W}^{(l)} \rangle \times \langle \mathbf{x}^{(l-1)} \rangle$
Non-Linear Layer
$\langle \mathbf{x}^{(l)} \rangle = \mathsf{ReLU}(\langle \mathbf{z}^{(l)} \rangle)$
$2PC$
$\langle \boldsymbol{\delta}^{(L)} \rangle = \nabla C(\mathbf{W}) \odot \langle \mathbf{z}'^{(l)} \rangle$
$\langle \boldsymbol{\delta}^{(l)} \rangle = (\langle (\mathbf{W}^{(l+1)})^T \rangle \times \langle \boldsymbol{\delta}^{(L)} \rangle) \odot \langle \mathbf{z}'^{(l)} \rangle$
$\langle \mathbf{z}'^{(l)} \rangle = \mathsf{DReLU}(\langle \mathbf{z}^{(l)} \rangle)$
$\langle \mathbf{W} \rangle_1$
P_1
$\langle \mathbf{x}_i \rangle_1$
$\langle \hat{y}_i \rangle_1$
$\langle C(\mathbf{W}) \rangle_1$

Fig. 1. The FssNN Framework

In details, each layer in forward propagation comprises of a linear operation (matrix multiplication), followed by a non-linear activation function ReLU. The backpropogation updates the weights appropriately making use of the derivative of ReLU (i.e., DReLU), and linear operations (matrix multiplication and Hadamard product). In FssNN, we propose the hybrid method (combining additive secret sharing with FSS) to train linear layers of neural network via additive secret sharing and non-linear layers via FSS. In fact, linear operation in neural network training can be efficiently implemented by any SS-based 2PC. Therefore, by leveraging *additive secret sharing* in offline-online computation model, matrix addition can be implemented locally and matrix multiplication with single round of communication in online phase. The *function secret sharing* (FSS)-based approach to 2PC with preprocessing can support non-linear operations with the same online communication and round complexity as linear operation, and only make use of symmetric cryptography. Therefore, ReLU and DReLU are implemented by using the DCF scheme, comparison-oriented FSS, with the sacrifice of more correlated randomness than 2PC based on additive secret sharing.

3.2 Threat Model

We consider the security against semi-honest adversaries, i.e., the corrupted parties (e.g., P_1 in Fig. 1) running the protocol honestly while trying to learn as much information as possible about others' input or function share. In our two-party setting, secure protocols can be used to enable two parties to train a neural network model on their joint data by first secretly sharing their inputs. Furthermore, the protocol and framework can also be used in *two-server* model, where the data owners (clients) process and secretly share their data between two non-colluding parties (servers) and then the two servers can train various models on the clients' joint data by running our two-party protocol.

4 Secure Linear Layer

Neural network mainly includes linear and non-linear layers. This section describes how to securely and efficiently achieve matrix multiplication and Hadamard product in linear layer using additive secret sharing.

4.1 Secure Matrix Multiplication

Neural network makes extensive use of matrix multiplications to benefit from vectorization techniques. The secure multiplication proposed in Section 2.2 can be straightforwardly extended to matrix multiplication [3]. Note that multiplication triples are replaced by matrix multiplication triples. Observe that the matrix multiplication operation used in neural network training has a specific structure that can be optimized in communication cost: the same matrix $\mathbf{X}$ is used in many multiplications with many different matrices $\mathbf{Y}_{i=1,\cdots,k}$. While we can use independently-generated multiplication triples $\{(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)\}_{i=1,\cdots,k}$, s.t., $\mathbf{C}_i = \mathbf{A}_i \times \mathbf{B}_i$, for each multiplication, Kelkar et al. [20] show a more efficient way to reuse a part of multiplication triples (i.e., just one $\mathbf{A}$ instead of $\{\mathbf{A}_i\}_{i=1,\cdots,k}$), and call these *correlated multiplication triples* which enable multiple online multiplications with the same matrix.

Algorithm 1 illustrates the specific details of correlated matrix multiplication. It requires $\{\mathbf{A}, \{\mathbf{B}_i, \mathbf{C}_i\}_{i=1,\ldots,k}\}$ satisfying $\mathbf{C}_i = \mathbf{A} \times \mathbf{B}_i$ to be the correlated multiplication triples which can be generated using the method outlined in [20].

4.2 Secure Hadamard Product

In backpropagation algorithm outlined in Section 2.1, Hadamard product is heavily used. Observe that the Hadamard product operations used in neural network training have a specific structure that can be leveraged to further reduce the communication cost: when computing $\mathbf{X} \odot \mathbf{Y}$, the elements of $\mathbf{Y}$ are bits (i.e., 0 or 1). Since the arithmetic shares $\langle \mathbf{X} \rangle$ cannot be directly multiplied with boolean shares $\langle \mathbf{Y} \rangle^B$ as they are calculated with different moduli. Many previous works first convert all elements in $\mathbf{Y}$ to n-bit values (e.g. Bit2A in [6]) and then perform

Algorithm 1 $\mathsf{MatMul}(\langle \mathbf{X}\rangle, \{\langle \mathbf{Y}_i\rangle\}_{i=1,\cdots,k})$

```
Input: Secret shares $\{\langle \mathbf{X}\rangle_b, \langle \mathbf{Y}_1\rangle_b, \cdots, \langle \mathbf{Y}_k\rangle_b | b = \{0,1\}\}$ and correlated multiplication triples $\{\mathbf{A}, \{\mathbf{B}_i, \mathbf{C}_i\}_{i=1,\cdots,k}\}$ generated in offline phase.
Output: Secret shares $\langle \mathbf{Z}_1\rangle_b, ..., \langle \mathbf{Z}_k\rangle_b$.
1: $P_b$ computes $\langle \mathbf{X}-\mathbf{A}\rangle_b, \langle \mathbf{Y}_1-\mathbf{B}_1\rangle_b, \cdots, \langle \mathbf{Y}_k-\mathbf{B}_k\rangle_b$ locally, and sends them to $P_{1-b}$.
2: $P_0$ and $P_1$ reconstruct $\mathbf{X}-\mathbf{A}, \mathbf{Y}_1-\mathbf{B}_1, \cdots, \mathbf{Y}_k-\mathbf{B}_k$.
3: for $i = 1$ to $k$ do
4:     $P_b$ computes $\langle \mathbf{Z}_i\rangle_b = b\cdot(\mathbf{X}-\mathbf{A})\times(\mathbf{Y}_j-\mathbf{B}_j)+(\mathbf{X}-\mathbf{A})\times\langle \mathbf{B}_j\rangle_b+\langle \mathbf{A}\rangle_b\times(\mathbf{Y}_j-\mathbf{B}_j)+\langle \mathbf{C}_j\rangle_b$ locally.
5: end for
6: return $\langle \mathbf{Z}_1\rangle_b, \cdots, \langle \mathbf{Z}_k\rangle_b$.
```

secure multiplication, requiring two rounds in the online phase. To be able to perform the calculation in one round, BitXA is directly proposed to multiplies an integer by a bit.

Specifically, we begin with scalar operation (i.e., $x \cdot y$), which can be straightforwardly extended to matrix operation (i.e., $\mathbf{X} \odot \mathbf{Y}$). Formally, given $x \in \mathbb{Z}_{2^n}$ and $y \in \mathbb{Z}_2$ and assume x and y are secretly shared between P_0 and P_1. Denote the shares by $\langle x\rangle_0, \langle y\rangle_0^B$ and $\langle x\rangle_1, \langle y\rangle_1^B$, where $\langle y\rangle_0^B, \langle y\rangle_1^B$ is the boolean shares of y. To calculate $\langle z\rangle_0, \langle z\rangle_1 \in \mathbb{Z}_{2^n}$ such that $\langle z\rangle_0 + \langle z\rangle_1 = y \cdot x$, BitXA precomputes some correctness randomness for shares conversion and multiplication in offline phase, allowing BitXA to be performed in one online round. Algorithm 2 shows the details of implementing $\langle y\rangle^B \cdot \langle x\rangle$.

Algorithm 2 $\mathsf{BitXA}(\langle y\rangle^B, \langle x\rangle)$

```
Input: Secret shares $\langle x\rangle_0, \langle x\rangle_1$ and boolean shares $\langle y\rangle_0^B, \langle y\rangle_1^B$.
Output: Secret shares $\langle z\rangle_0, \langle z\rangle_1$.
1: // Offline Phase
2: Dealer samples $\langle \hat{\delta}_y\rangle_b^B \leftarrow\$ \mathbb{Z}_2$ for $b = \{0,1\}$, and subsequently converts them to arithmetic secret shares by applying the following steps:
     – Dealer sets $\langle e\rangle_0 = \langle \hat{\delta}_y\rangle_0^B, \langle f\rangle_0 = 0; \langle e\rangle_1 = 0, \langle f\rangle_1 = \langle \hat{\delta}_y\rangle_1^B$, and computes $(\langle ef\rangle_0, \langle ef\rangle_1) \leftarrow \mathsf{Mul}(\langle e\rangle, \langle f\rangle)$.
     – Dealer computes $\langle \delta_y\rangle_b = \langle e\rangle_b + \langle f\rangle_b - 2\cdot\langle ef\rangle_b$ for $b = \{0,1\}$.
3: Dealer samples $\langle \delta_x\rangle_b, \langle \delta_z\rangle_b \leftarrow\$ \mathbb{Z}_{2^n}$ for $b = \{0,1\}$, subject to $(\langle \delta_z\rangle_0 + \langle \delta_z\rangle_1) = (\langle \delta_y\rangle_0 + \langle \delta_y\rangle_1)\cdot(\langle \delta_x\rangle_0 + \langle \delta_x\rangle_1)$.
4: Dealer sends $\langle \delta_x\rangle_b, \langle \delta_y\rangle_b, \langle \delta_z\rangle_b$ and $\langle \hat{\delta}_y\rangle_b^B$ to $P_b$ respectively.
5: // Online Phase
6: $P_b$ locally computes $\langle x\rangle_b + \langle \delta_x\rangle_b, \langle y\rangle_b^B \oplus \langle \hat{\delta}_y\rangle_b^B$, and sends them to $P_{1-b}$.
7: $P_b$ reconstructs $\Delta_x = x + \delta_x, \Delta_y = y \oplus \hat{\delta}_y$, and sets $\Delta'_y = \Delta_y$ where $\Delta'_y \in \mathbb{Z}_{2^n}$.
8: $P_b$ locally calculates $\langle z\rangle_b = b\cdot\Delta'_y\cdot\Delta_x + \langle \delta_y\rangle_b\cdot\Delta_x - 2\cdot\Delta'_y\cdot\Delta_x\cdot\langle \delta_y\rangle_b - \Delta'_y\cdot\langle \delta_x\rangle_b - \langle \delta_z\rangle_b + 2\cdot\Delta'_y\cdot\langle \delta_z\rangle_b$.
9: return $\langle z\rangle_b$.
```

4.3 Security Analysis

Consider the security in the semi-honest model where two parties follow the protocol exactly, and to be proven in the framework of Universal Composition (UC) [21].

Theorem 1. *Let P_b denote the corrupted party controlled by an adversary $\mathcal{A}$ where b is either 0 or 1. Adversary $\mathcal{A}$ cannot learn any knowledge of the private data of honest party P_{1-b} in* MatMul *and* BitXA *algorithm.*

Proof. For matrix multiplication operation MatMul (resp. hadamard product operation BitXA), there is only one interaction between P_0 and P_1 which happens in the first step of online phase, where P_b send $\langle \mathbf{X}-\mathbf{A}\rangle_b, \langle \mathbf{Y}_1-\mathbf{B}_1\rangle_b, \cdots, \langle \mathbf{Y}_k-\mathbf{B}_k\rangle_b$ (resp. $\langle x\rangle_b + \langle \delta_x\rangle_b, \langle y\rangle_b^B \oplus \langle \hat{\delta}_y\rangle_b^B$) to P_{1-b}. $\mathcal{A}$ can learn nothing except masked values $\mathbf{X}-\mathbf{A}, \mathbf{Y}_1-\mathbf{B}_1, \cdots, \mathbf{Y}_k-\mathbf{B}_k$(resp. $\Delta_x = x+\delta_x, \Delta_y = y \oplus \hat{\delta}_y$). And due to the security of additive secret sharing and boolean secret sharing, the distribution of these masked values and a uniform values are identical for $\mathcal{A}$. Hence, MatMul and BitXA is a secure primitive in our protocol due to the universally composable theorem.

5 Secure Non-Linear Layer

This section describes how to securely and efficiently achieve matrix multiplication and Hadamard product in linear layer using additive secret sharing.

5.1 Secure ReLU Function

The ReLU function is a widely utilized activation function in neural network training, and ReLU and its derivative are denoted as:

$$\mathsf{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} \qquad \mathsf{DReLU}(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{1}$$

Since $\mathsf{ReLU}(x) = x \cdot \mathbf{1}\{x \geq 0\}$ and $\mathsf{DReLU}(x) = \mathbf{1}\{x \geq 0\}$, $\mathsf{ReLU}(x) = x \cdot \mathsf{DReLU}(x)$ where $\mathbf{1}\{b\}$ denotes the indicator function that outputs 1 when b is true and 0 otherwise. To securely compute $\mathsf{ReLU}(\langle x\rangle)$, $\mathsf{DReLU}(\langle x\rangle)$ is computed first, followed by computing $\mathsf{ReLU}(\langle x\rangle) = \mathsf{BitXA}(\langle \mathsf{DReLU}(x)\rangle^B, \langle x\rangle)$ via $\mathsf{BitXA}()$ algorithms in section 4.2.

Building blocks: DCF and DDCF A special piecewise function, $f^{<}_{\alpha,\beta}$, also referred to as a comparison function, outputs β if $x < \alpha$ and 0 otherwise. We refer to an FSS scheme for comparison functions as distributed comparison function (DCF). And the variant of DCF, called Dual Distributed Comparison Function (DDCF), is considered and denoted by $f^{<}_{\alpha,\beta_1,\beta_2}$ that outputs β_1 for $0 \leq x < \alpha$ and β_2 for $x \geq \alpha$. Obviously, $f^{<}_{\alpha,\beta_1,\beta_2}(x) = \beta_2 + f^{<}_{\alpha,\beta_1-\beta_2}(x)$ and thus DDCF can be constructed by DCF.

Secure DReLU Function Given $x, y \in \mathbb{U}_N$, Algorithm 3 illustrates a secure two-party computation protocol to compute $\mathbf{1}\{x < y\}$, where $(\mathsf{Gen}^{\mathsf{DDCF}}_{n-1}, \mathsf{Eval}^{\mathsf{DDCF}}_{n-1}$ is a variant of the DCF scheme, and the algorithm is shown in the appendix A.

Algorithm 3 Comp: $(\mathsf{Gen}^{\mathsf{Comp}}_n, \mathsf{Eval}^{\mathsf{Comp}}_n)$

```
$\mathsf{Gen}^{\mathsf{Comp}}_n(1^\lambda, \mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{in}}_2, \mathsf{r}^{\mathsf{out}})$
 1: Let $r = (2^n - (\mathsf{r}^{\mathsf{in}}_1 - \mathsf{r}^{\mathsf{in}}_2)) \in \mathbb{U}_N$, and $\alpha^{(n-1)} = r_{[0,n-1)}$.
 2: $(k^{(n-1)}_0, k^{(n-1)}_1) \leftarrow \mathsf{Gen}^{\mathsf{DDCF}}_{n-1}(1^\lambda, \alpha^{(n-1)}, \beta_0, \beta_1)$, where $\beta_0 = 1 \oplus r_{[n-1]}, \beta_1 = r_{[n-1]}$.
 3: Sample randoms $\mathsf{r}_0, \mathsf{r}_1 \leftarrow\$ \{0,1\}$, s.t. $\mathsf{r}_0 \oplus \mathsf{r}_1 = \mathsf{r}^{\mathsf{out}}$.
 4: For $b \in \{0,1\}$, let $k_b = k^{(n-1)}_b || \mathsf{r}_b$
 5: return $(k_0, k_1)$.
$\mathsf{Eval}^{\mathsf{Comp}}_n(b, k_b, \hat{x}, \hat{y})$
 1: Parse $k_b = k^{(n-1)}_b || \mathsf{r}_b$.
 2: Let $z = (\hat{x} - \hat{y}) \in \mathbb{U}_N$.
 3: Let $m^{(n-1)}_b \leftarrow \mathsf{Eval}^{\mathsf{DDCF}}_{n-1}(b, k^{(n-1)}_b, z^{(n-1)})$, where $z^{(n-1)} = 2^{n-1} - z_{[0,n-1)} - 1$.
 4: $v_b = (b \cdot z_{[n-1]}) \oplus m^{(n-1)}_b \oplus \mathsf{r}_b$.
 5: return $v_b$.
```

Further, utilize $(\mathsf{Gen}^{\mathsf{Comp}}_n, \mathsf{Eval}^{\mathsf{Comp}}_n)$ to compute $\mathbf{1}\{x < 0\}$, and subsequently compute $\mathsf{DReLU}\,(x) = \mathbf{1}\{x \geq 0\} = \neg(\mathbf{1}\{x < 0\})$ where $\neg$ is the negation operator. The proposed FSS scheme for DReLU is outlined in Algorithm 4. In this scheme, it is assumed that P_0 and P_1 hold secret shares $\langle x \rangle_0$ and $\langle x \rangle_1$ respectively.

Algorithm 4 $\mathsf{DReLU}(\langle x \rangle)$

```
 1: // Offline Phase
 2: Dealer computes $(k_0, k_1) \leftarrow \mathsf{Gen}^{\mathsf{Comp}}_n(1^\lambda, \mathsf{r}^{\mathsf{in}}_1, 2^{n-1}, \mathsf{r}^{\mathsf{out}})$, and sends $k_0, k_1$ to $P_0, P_1$.
 3: Dealer invokes $\mathsf{Shr}(\mathsf{r}^{\mathsf{in}}_1)$, and subsequently sends $\langle \mathsf{r}^{\mathsf{in}}_1 \rangle_0, \langle \mathsf{r}^{\mathsf{in}}_1 \rangle_1$ to $P_0, P_1$ respectively.
 4: // Online Phase
 5: For $b = 0, 1$, $P_b$ computes $\langle x + \mathsf{r}^{\mathsf{in}}_1 \rangle_b = \langle x \rangle_b + \langle \mathsf{r}^{\mathsf{in}}_1 \rangle_b$ locally, and send it to $P_{1-b}$.
 6: For $b = 0, 1$, $P_b$ reconstructs and obtains $x + \mathsf{r}^{\mathsf{in}}_1$.
 7: For $b = 0, 1$, $P_b$ computes $v_b \leftarrow b \oplus \mathsf{Eval}^{\mathsf{Comp}}_n(b, k_b, x + \mathsf{r}^{\mathsf{in}}_1, 0)$ locally.
 8: return $v_b$.
```

5.2 Communication-Optimized DCF

A central building block for our protocol is the FSS scheme for the comparison function, which is intensively used in neural network to build non-polynomial activation functions like ReLU. In this paper, we examine the case of $x, \alpha \in \mathbb{Z}_{2^n}$ and $\beta \in \mathbb{Z}_2$ and propose a communication-optimized DCF construction for the comparison function $f^{<}_{\alpha,\beta}$. The proposed construction has a smaller key size compared to state-of-the-art schemes [14] from $\lambda+(n+1)(\lambda+3)$ to $\lambda+n(\lambda+3)$ for $\mathbb{G}^{\mathsf{in}}=\mathbb{Z}_{2^n}, \mathbb{G}^{\mathsf{out}}=\mathbb{Z}_2$ and security parameter λ.

Intuition The construction draws inspiration from the distributed point function (DPF) of [12]. The Gen algorithm uses a PRG G and generates two keys (k_0,k_1) such that $\forall b\in\{0,1\}$, k_b includes a random pseudorandom generator (PRG) seed s_b and n shared correction words $(CW^{(1)},\cdots,CW^{(n)})$. A key implicitly defines a GGM-style binary tree [22] with 2^n leaves, where the leaves are labeled by inputs x. Each node in the tree is associated with a label represented by a tuple $(s,v,t)\in\{0,1\}^{\lambda}\times\{0,1\}\times\{0,1\}$, where s represents seed, v represents resulting bit, and t represents state bit. The label of each node is fully determined by the label of its parent node. The construction ensures that the "sum" $v_0\oplus v_1$ over all nodes leading to an input x that is exactly equal to $f^{<}_{\alpha,\beta}(x)$. Therefore, evaluating a key k_b on an input x (i.e., the Eval algorithm) requires traversing the tree generated by k_b from the root to the leaf representing x, computing (s_b,v_b,t_b) at each node and summing up the values v_b. Next, we will provide a comprehensive explanation of Gen and Eval algorithm by detailing *key generation phase* and *evaluation phase*.

Key generation phase Specifically, the function Gen generates the secret share of the function $f^{<}_{\alpha,\beta}(x)$ (i.e., k_0 and k_1) by generating distributed GGM-style binary trees. The two GGM-style trees generated by Gen are equivalent to the GGM-style trees representing the function $f^{<}_{\alpha,\beta}(x)$ when taken together. In this construction, the path from the root to a leaf node labeled by x is referred to as the *evaluation path* of x, and the evaluation path of the special input α is referred to as the *special evaluation path*. When $x\neq\alpha$, the prefix of x diverges from the path to α at a exact point, referred to as the *divergence node* of x relative to α. To ensure the correct creation of the two trees, we would like to maintain the *invariant*: 1) For each node on the special evaluation path, two seeds (on the two trees) are indistinguishable as random and independent, two resulting bits are identical and two control bits differ; and 2) For each node outside the special evaluation path, with the exception of the node that is the left child of divergence node, the two labels are identical. At the left child of the divergence node, two seeds and control bits are the same, but the "sum" of two resulting bits is equal to β.

Note that since the label of a node is determined by that of its parent, if the aforementioned invariant is satisfied for a node outside the special path, it will automatically be upheld by its children. Additionally, we can easily meet the invariant for the root (which is always on the special evaluation path) by simply including the labels in the keys. The challenge lies in ensuring that the invariant is also upheld when leaving the special path. In order to describe the construction, it is useful to view the two labels of a node as a Boolean secret sharing of the label, consisting of shares $\langle s\rangle^B=(s_0,s_1)$ of the λ-bit seed s, $\langle v\rangle^B=(v_0,v_1)$ of the resulting bit v and $\langle t\rangle^B=(t_0,t_1)$ of the control bit t. Suppose that the labels of

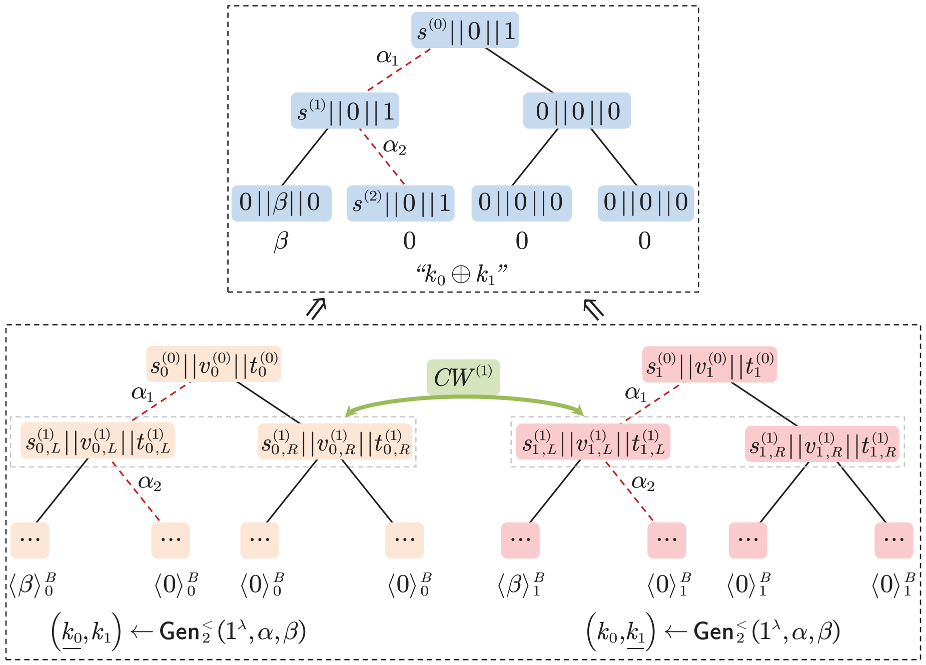
$s^{(0)}||0||1$
α_1
$s^{(1)}||0||1$
$0||0||0$
α_2
$0||\beta||0$
$s^{(2)}||0||1$
$0||0||0$
$0||0||0$
β
0
0
0
"$k_0 \oplus k_1$"
$s_0^{(0)}||v_0^{(0)}||t_0^{(0)}$
$CW^{(1)}$
$s_1^{(0)}||v_1^{(0)}||t_1^{(0)}$
$s_{0,L}^{(1)}||v_{0,L}^{(1)}||t_{0,L}^{(1)}$
$s_{0,R}^{(1)}||v_{0,R}^{(1)}||t_{0,R}^{(1)}$
$s_{1,L}^{(1)}||v_{1,L}^{(1)}||t_{1,L}^{(1)}$
$s_{1,R}^{(1)}||v_{1,R}^{(1)}||t_{1,R}^{(1)}$
$\langle\beta\rangle_0^B$
$\langle 0\rangle_0^B$
$\langle 0\rangle_0^B$
$\langle 0\rangle_0^B$
$\langle\beta\rangle_1^B$
$\langle 0\rangle_1^B$
$\langle 0\rangle_1^B$
$\langle 0\rangle_1^B$
$(\underline{k_0}, k_1) \leftarrow \mathsf{Gen}_2^{<}(1^\lambda, \alpha, \beta)$
$(k_0, \underline{k_1}) \leftarrow \mathsf{Gen}_2^{<}(1^\lambda, \alpha, \beta)$

Fig. 2. A construction example of $\mathsf{Gen}_n^{<}$ when $n = 2$

the i-th node u_i on the evaluation path are $(s_b^{(i)}, v_b^{(i)}, t_b^{(i)})$ $(b = 0, 1)$. To compute the labels of the $(i+1)$-th node, the parties start by locally computing $G(s_b^{(i)})$ for a PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2(\lambda+2)}$ and parsing $G(s_b^{(i)})$ as $(s_b^L, v_b^L, t_b^L; s_b^R, v_b^R, t_b^R)$. The first three values correspond to labels of the left child and the last three values correspond to labels of the right child. To maintain the invariant, the keys will include a correction word (CW) for each level i. As previously discussed, we only need to take into account the case where u_i is on the special evaluation path. By the invariant we have $t = 1$, in which case the correction will be applied. Without loss of generality, let us assume that $\alpha_i = 1$. This implies that the left child of u_i is not on the special evaluation path, while the right child is on the special evaluation path. To ensure that the invariant is maintained, we can include in both keys the correction $CW^{(i)} = (s^L, v^L \oplus \beta, t^L \oplus 1; s^L, v^R, t^R \oplus 1)$. Indeed, this ensures that after the correction is applied, the labels of the left (i.e., $b = 0$) and right child (i.e., $b = 1$) are $(\langle 0^\lambda\rangle_b^B, \langle\beta\rangle_b^B, \langle 0\rangle_b^B; \langle s\rangle_b^B, \langle 0\rangle_b^B, \langle 1\rangle_b^B)$ as required. The n correction values $CW^{(i)}$ are computed by Gen from the root labels by applying the above iterative computation along the special path, and are included in both keys. Figure 2 illustrates a construction example of Gen when $n = 2$, with the specific case of $\alpha = \alpha_1\alpha_2 = 01$ being depicted.

Evaluation phase In DCF, the evaluation process involves comparing a public input $x \in \mathbb{Z}_{2^n}$ to a private value α, and it goes as follows: two evaluators are each given a key which includes a distinct initial seed and n correction words. Each evaluator starts from the root, at each step i goes down one node in the tree and generate $i+1$-th labels depending on the bit x_i using a common correction word $CW^{(i)}$. At the end of the computation, each evaluator outputs resulting bit. Note that the tuple (s_b, v_b, t_b) associated with node u_i is a function of the seed associated with the parent of u_i and the correction words. Therefore, if $s_0 = s_1$ then for any descendent of u_i, k_0 and k_1 generate identical tuples. The correction words are chosen such that when a path to x departs from the path to α, the two seeds s_0 and s_1 on the first node off the path are identical, and the sum of $v_0 \oplus v_1$ along the whole path to u_i is exactly β if the departure is to the left of the path to α, i.e. $x < \alpha$, and is 0 if the departure is to the right of the path to α. Finally, along the path to α any seed s_b is computationally indistinguishable from a random string given the key k_{1-b}, which ensures the security of the construction.

Distributed Comparison Function $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$ are given by Algorithm 5 , where $G : \{0,1\}^\lambda \to \{0,1\}^{2(\lambda+2)}$ be a pseudorandom generator, and $||$ is concatenation operator.

Next, we will show the correctness and privacy of the DCF construction through Theorem 2, and its proof can be found in Appendix B.

Theorem 2. *(Proof of Correctness and Privacy) Let* $G : \{0,1\}^\lambda \leftarrow \{0,1\}^{2(\lambda+2)}$ *be a PRG. Then the scheme* $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$ *from Algorithm 5 is a DCF for the family of comparison functions* $f_{\alpha,\beta}^< : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$ *with key size* $\lambda + n(\lambda+3)$ *bits, where* $\mathbb{G}^{\mathsf{in}} = \mathbb{Z}_{2^n}, \mathbb{G}^{\mathsf{out}} = \mathbb{Z}_2$ *and security parameter* λ*. The number of PRG invocations in* Gen *is* $2n$ *and the number of PRG invocations in* Eval *is* n*.*

5.3 Distributed DCF key generation

In this section, we discuss concrete approaches for implementing offline phase via a small-scale two-party secure protocol, where two parties jointly emulate the role of dealer. Recall that this amounts to 2-party protocols for securely executing the Gen procedure for appropriate DCF gates, in turn reducing to secure generation of keys for DCF. There are two main methods for achieving this goal: 1) a variant of the secure DPF generation protocol of Doerner and Shelat [18] for small and moderate domain sizes (e.g., 2^{16} or smaller), which is black-box in the underlying PRG; and 2) generic 2PC approaches, implementing the PRG via "2PC-friendly" ciphers. To support larger domain sizes (e.g., 2^{32} in our approach), the second method is preferred.

PRG-based generation This paper generates k_0, k_1 based on secure two-party PRG, where $k_b = s_b^{(0)}||CW^{(1)}||...||CW^{(n)}$. The specific process is shown in Algorithm 6, where the algorithm $\mathsf{SecPRG}()$ is realized by the LPN-style PRG proposed by [23].

Algorithm 5 DCF: $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$

```
$\mathsf{Gen}_n^<(1^\lambda, \alpha, \beta)$
1: Let $\alpha = \alpha_1, \cdots, \alpha_n \in \{0,1\}^n$ be the bit decomposition of $\alpha$.
2: Sample randoms $s_0^{(0)} \leftarrow\$ \{0,1\}^\lambda$ and $s_1^{(0)} \leftarrow\$ \{0,1\}^\lambda$, $v_0^{(0)} = 0, v_1^{(0)} = 0$, and $t_0^{(0)} = 0, t_1^{(0)} = 1$.
3: for $i = 1$ to $n$ do
4:     $s_0^L||v_0^L||t_0^L||s_0^R||v_0^R||t_0^R \leftarrow G(s_0^{(i-1)})$, and $s_1^L||v_1^L||t_1^L||s_1^R||v_1^R||t_1^R \leftarrow G(s_1^{(i-1)})$
5:     if $\alpha_i = 0$ then
6:         $\mathsf{Keep} \leftarrow L, \mathsf{Lose} \leftarrow R$.
7:     else
8:         $\mathsf{Keep} \leftarrow R, \mathsf{Lose} \leftarrow L$.
9:     end if
10:    $s_{CW} \leftarrow s_0^{\mathsf{Lose}} \oplus s_1^{\mathsf{Lose}}$.
11:    $v_{CW}^L \leftarrow v_0^L \oplus v_1^L \oplus (\alpha_i \cdot \beta)$.
12:    $t_{CW}^L \leftarrow t_0^L \oplus t_1^L \oplus \alpha_i \oplus 1$, and $t_{CW}^R \leftarrow t_0^R \oplus t_1^R \oplus \alpha_i$.
13:    $CW^{(i)} = s_{CW}||v_{CW}^L||t_{CW}^L||t_{CW}^R$.
14:    $s_0^{(i)} = s_0^{\mathsf{Keep}} \oplus t_0^{(i-1)} \cdot s_{CW}$, and $s_1^{(i)} = s_1^{\mathsf{Keep}} \oplus t_1^{(i-1)} \cdot s_{CW}$.
15:    $t_0^{(i)} = t_0^{\mathsf{Keep}} \oplus t_0^{(i-1)} \cdot t_{CW}^{\mathsf{Keep}}$, and $t_1^{(i)} = t_1^{\mathsf{Keep}} \oplus t_1^{(i-1)} \cdot t_{CW}^{\mathsf{Keep}}$.
16: end for
17: Let $k_b = s_b^{(0)}||CW^{(1)}|| \cdots ||CW^{(n)}$.
18: return $(k_0, k_1)$.
$\mathsf{Eval}_n^<(b, k_b, x)$
1: Parse $k_b = s^{(0)}||CW^{(1)}|| \cdots ||CW^{(n)}$, and let $x = x_1, \cdots, x_n \in \{0,1\}^n$, $v = 0$, and $t^{(0)} = b$.
2: for $i = 1$ to $n$ do
3:     Parse $CW^{(i)} = s_{CW}||v_{CW}^L||t_{CW}^L||t_{CW}^R$.
4:     Parse $G(s^{(i-1)}) = \hat{s}^L||\hat{v}^L||\hat{t}^L||\hat{s}^R||\hat{v}^R||\hat{t}^R$.
5:     $\tau \leftarrow (\hat{s}^L||\hat{v}^L||\hat{t}^L||\hat{s}^R||\hat{v}^R||\hat{t}^R) \oplus (t^{(i-1)} \cdot [s_{CW}||v_{CW}^L||t_{CW}^L||s_{CW}||v_{CW}^L||t_{CW}^R])$.
6:     Parse $\tau = s^L||v^L||t^L||s^R||v^R||t^R$.
7:     if $x_i = 0$ then
8:         $v \leftarrow v \oplus v^L$
9:         $s^{(i)} \leftarrow s^L, t^{(i)} \leftarrow t^L$
10:    else
11:        $s^{(i)} \leftarrow s^R, t^{(i)} \leftarrow t^R$
12:    end if
13: end for
14: return $v$
```

Algorithm 6 $\mathsf{Gen}^{\mathsf{DCF}}(1^\lambda, b, \{\langle\alpha_i\rangle_b\}_{i=1\ldots n}, \langle\beta\rangle_b^B)$

```
1: For $b = 0, 1$, $P_b$ samples randoms $s_b^{(0)} \leftarrow\!\$ \{0,1\}^\lambda, t_b^{(0)} = b$.
2: For $b = 0, 1$, $P_b$ invokes $\mathsf{Shr}(s_b^{(0)}), \mathsf{Shr}(t_b^{(0)})$ to generate secret shares $s_b^{(0)}$ and $t_b^{(0)}$,
   then $P_b$ obtains $\langle s_0^{(0)}\rangle_b, \langle t_0^{(0)}\rangle_b^B, \langle s_1^{(0)}\rangle_b, \langle t_1^{(0)}\rangle_b^B$ respectively.
3: for $i = 1$ to $n$ do
4:     $P_0$ and $P_1$ compute $(\langle G(s_b^{(i-1)})\rangle_0, \langle G(s_b^{(i-1)})\rangle_1) \leftarrow \mathsf{SecPRG}(\langle s_b^{(i-1)}\rangle_0, \langle s_b^{(i-1)}\rangle_1)$,
       where $\langle G(s_b^{(i-1)})\rangle_0 \oplus \langle G(s_b^{(i-1)})\rangle_1 = s_b^{L,i-1}||v_b^{L,i-1}||t_b^{L,i-1}||s_b^{L,i-1}||v_b^{L,i-1}||t_b^{L,i-1}$,
       $b = 0, 1$.
5:     $P_0$ and $P_1$ run a 2PC protocol together to compute:
       $$(s_{CW}, v_{CW}^L) \leftarrow \begin{cases} (s_0^{R,i-1} \oplus s_1^{R,i-1}, v_0^{L,i-1} \oplus v_1^{L,i-1}) & \alpha_i = 0 \\ (s_0^{L,i-1} \oplus s_1^{L,i-1}, v_0^{L,i-1} \oplus v_1^{L,i-1} \oplus \beta) & \alpha_i = 1 \end{cases}$$
       $$(t_{CW}^L, t_{CW}^R) \leftarrow \begin{cases} (t_0^{L,i-1} \oplus t_1^{L,i-1} \oplus 1, t_0^{R,i-1} \oplus t_1^{R,i-1}) & \alpha_i = 0 \\ (t_0^{L,i-1} \oplus t_1^{L,i-1}, t_0^{R,i-1} \oplus t_1^{R,i-1} \oplus 1) & \alpha_i = 1 \end{cases}$$
       $P_0$ and $P_1$ obtain $s_{CW}; v_{CW}^L; t_{CW}^L, t_{CW}^R$.
6:     For $b = 0, 1$, $P_b$ computes $CW^{(i)} = s_{CW}||v_{CW}^L||t_{CW}^L||t_{CW}^R$ locally.
7:     $P_0$ and $P_1$ engage in a 2PC protocol to compute:
       $$(s_b^{(i)}, t_b^{(i)}) \leftarrow \begin{cases} (s_b^{L,i-1} \oplus t_b^{(i-1)} \cdot s_{CW}, t_b^{L,i-1} \oplus t_b^{(i-1)} \cdot t_{CW}^L) & \alpha_i = 0 \\ (s_b^{R,i-1} \oplus t_b^{(i-1)} \cdot s_{CW}, t_b^{R,i-1} \oplus t_b^{(i-1)} \cdot t_{CW}^R) & \alpha_i = 1 \end{cases}$$
       $P_0$ and $P_1$ obtain $(\langle s_b^{(i)}\rangle_0, \langle t_b^{(i)}\rangle_0), (\langle s_b^{(i)}\rangle_1, \langle t_b^{(i)}\rangle_1)$ respectively, where $b = 0, 1$.
8: end for
9: Let $k_b \leftarrow s_b^{(0)}||CW^{(1)}||...||CW^{(n)}$.
10: return $k_b$.
```

In this construction, the number of SecPRG invocations is n and the number of 2PC invocations is $2n$. The "2PC-friendly" PRG [23] can be used to instantiate SecPRG and a secret sharing-based two-party protocol [10] can be used to implement 2PC.

6 Performance Analysis and Experiment

6.1 Performance Analysis

In this section, we present the performance analysis of our protocol. Table 1 gives online rounds and communication of our protocols, ABY2.0 [6] and AriaNN [17], where input size is $n_1 \times n_2, n_2 \times n_3$ for MatMul, n for BitXA, DReLU and ReLU.

6.2 Experiment

In this section, we present the results of our implementation of FssNN using Python, and present detailed experiment results and analysis to provide evidence of the effectiveness and efficiency of FssNN. Our implementation utilized

Table 1. Online communication complexity

Protocol	Rounds			Online Comm.		
	ABY2.0	AriaNN	Ours	ABY2.0	AriaNN	Ours
MatMul	1	1	1	n_1n_3	$n_1n_2 + n_2n_3$	$n_1n_2 + n_2n_3$
BitXA	1	1	1	$2n$	$2n$	$n + 1$
DReLU	$1 + \log n$	1	1	$\sim 3n$	n	n
ReLU	$2 + \log n$	2	2	$\sim 5n$	$3n$	$2n + 1$

the PySyft library [24], which enhances machine learning frameworks such as PyTorch with a communication layer and supports fixed precision arithmetic. The experiments are run on a Macbook Air with Apple M1 processor and 8GB memory in LAN settings.

In order to simplify comparison with existing work, we follow a setup very close to AriaNN [17], which provides the most extensive experiments of private training and inference. We report experimental numbers for both the offline and the online phase of our protocols separately, however in some cases we only provide the total cost (online + offline) for the purpose of comparison.

Experiments for Secure Layer Function In this section, we compare the performance of the proposed linear protocols and non-linear protocol with the counterparts implemented in AriaNN.

Matrix Multiplication We reproduce the matrix multiplication in neural networks to test the performance of MatMul. Under different batch sizes, experiment tests offline phase and total communication cost of matrix multiplication in a neural network with three linear layers in Table 2. Experimental results show that the application of the correlated matrix multiplication leads to a reduction in communication cost in the offline phase, ranging from 18% to 35%. This reduction in the number of multiplication triples results in a significant decrease in the amount of communication required in the offline phase.

Hadamard product We test the performance of HadamProd by measuring required communication cost of Boolean matrix $\mathbf{B}$ multiplied by Arithmetic matrix $\mathbf{A}$ where matrix size is both $N \times N$. Table 3 shows the communication cost and times of HadamProd in offline and online phase. And it shows that our HadamProd is communication efficient and its communication cost is reduced by $1.6\times \sim 1.7\times$.

Table 2. Communication cost and times of MatMul in offline and online phase

Framework	Performance	Input size (N)				
		64	128	256	512	1024
AriaNN	Offline Comm.(MB)	4.663	6.576	10.402	18.054	33.359
	Online Comm.(MB)	6.172	8.723	13.824	24.027	44.434
	Total Time (ms)	104.253	137.282	200.170	289.409	535.263
FssNN (Ours)	Offline Comm.(MB)	3.257	4.662	7.473	13.094	24.336
	Online Comm.(MB)	6.172	8.723	13.824	24.027	44.433
	Total Time (ms)	104.764	135.622	190.929	278.856	497.823

Table 3. Communication cost and times of HadamProd in offline and online phase

Framework	Performance		Input size (N)				
			64	128	256	512	1024
AriaNN	Comm.(MB)	Offline	0.162	0.631	2.506	10.006	40.006
		Online	0.063	0.251	1.001	4.001	16.001
	Time.(ms)	Offline	5.070	6.640	15.560	65.100	280.530
		Online	2.160	2.620	6.070	33.490	128.930
FssNN (Ours)	Comm.(MB)	Offline	0.109	0.414	1.633	6.507	26.008
		Online	0.037	0.142	0.564	2.251	9.001
	Time.(ms)	Offline	5.050	8.000	13.500	48.940	190.890
		Online	2.210	4.100	5.450	18.960	82.690

DReLU and ReLU To evaluate the performance of DReLU and ReLU, we conducted experiments by calculating DReLU and ReLU for $N \times N$ matrix. We then measured the communication cost and time. The results of our experiments indicate that DReLU and ReLU designed by FssNN outperforms AriaNN in terms of communication cost and time. This is attributed to the reduction in key size of the DCF scheme used in FssNN.

Figure 4 shows the total time and communication of the secure layers function, where input size is 128 for MatMul, 512 for HadamProd, 256 for DReLU and ReLU in total time; input size is 1024 for MatMul and HadamProd, 256 for DReLU and ReLU in total communication.

Experiments for Neural Network We assess secure neural network training on the dataset MNIST, and consider a 3-layer fully-connected network. Experimental results demonstrate that FssNN achieved lower communication cost and training time per epoch than AriaNN.

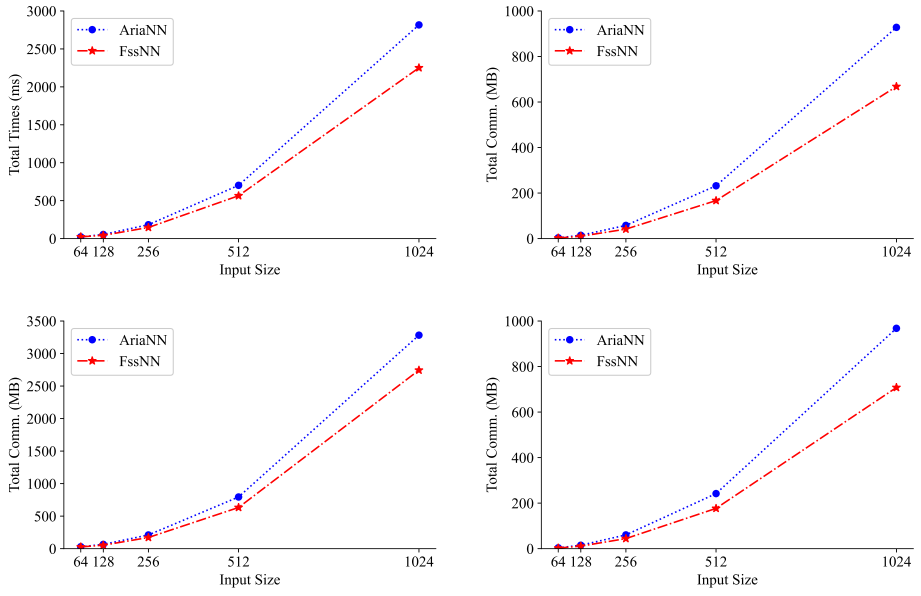

Fig. 3. Total time and communication of DReLU and ReLU

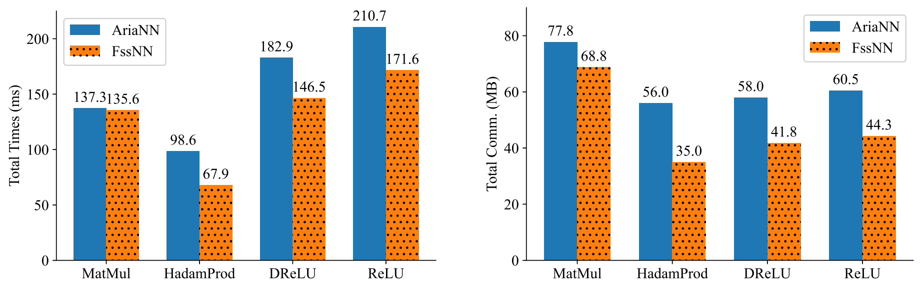

Fig. 4. Total time and communication of the secure layers function.

Table 4. Performance of secure neural network training

Framework	Model	Dataset	Epochs	Comm. (GB)	Time (h)	Accuracy
AriaNN	FCNN	MNIST	15	36.11	0.28	97.95%
FssNN (Ours)	FCNN	MNIST	15	27.52	0.23	98.00%

7 Conclusion

PPNN still come at a steep performance overhead that may not be amiable for the real-world scenarios. During training, heavy cryptographic computations are required to conduct, imposing intensive computational and communication overheads. In this paper, we proposed secure and lightweight framework FssNN for neural network training by leveraging function secret sharing. Our evaluation shows the practical performance of our design, as well as the substantial performance advantage over prior art. More attempts might be made to construct actively secure algorithms to defend against malicious adversary.

Acknowledgement

The work is supported by National Natural Science Foundation of China (62272131), National Key Research and Development Program of China (2022YFB3100087), Shenzhen Basic Research Project of China (JCYJ20200109113405927) and Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1212010005).

References

1. Yao, A.C.C.: How to generate and exchange secrets. In: 27th Annual Symposium on Foundations of Computer Science (sfcs 1986). pp. 162–167. IEEE (1986)
2. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: Proceedings of the Nineteenth ACM Symp. on Theory of Computing, STOC. pp. 218–229. ACM (1987)
3. Mohassel, P., Zhang, Y.: Secureml: A system for scalable privacy-preserving machine learning. In: 2017 IEEE symposium on security and privacy (SP). pp. 19–38. IEEE (2017)
4. Wagh, S., Gupta, D., Chandran, N.: Securenn: 3-party secure computation for neural network training. Proc. Priv. Enhancing Technol. **2019**(3), 26–49 (2019)
5. Agrawal, N., Shahin Shamsabadi, A., Kusner, M.J., Gascón, A.: Quotient: two-party secure neural network training and prediction. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1231–1247 (2019)
6. Patra, A., Schneider, T., Suresh, A., Yalame, H.: Aby2. 0: Improved mixed-protocol secure two-party computation. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2165–2182 (2021)
7. Rathee, D., Rathee, M., Goli, R.K.K., Gupta, D., Sharma, R., Chandran, N., Rastogi, A.: Sirnn: A math library for secure rnn inference. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1003–1020. IEEE (2021)
8. Liu, X., Zheng, Y., Yuan, X., Yi, X.: Medisc: Towards secure and lightweight deep learning as a medical diagnostic service. In: European Symposium on Research in Computer Security. pp. 519–541. Springer (2021)
9. Huang, Z., jie Lu, W., Hong, C., Ding, J.: Cheetah: Lean and fast secure two-party deep neural network inference. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 809–826. USENIX Association (2022)

10. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Annual Cryptology Conference. pp. 643–662. Springer (2012)

11. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 337–367. Springer (2015)

12. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing: Improvements and extensions. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1292–1303 (2016)

13. Boyle, E., Gilboa, N., Ishai, Y.: Secure computation with preprocessing via function secret sharing. In: Theory of Cryptography Conference. pp. 341–371. Springer (2019)

14. Boyle, E., Chandran, N., Gilboa, N., Gupta, D., Ishai, Y., Kumar, N., Rathee, M.: Function secret sharing for mixed-mode and fixed-point secure computation. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 871–900. Springer (2021)

15. Wagh, S.: Pika: Secure computation using function secret sharing over rings. Proceedings on Privacy Enhancing Technologies **4**, 351–377 (2022)

16. Gupta, K., Kumaraswamy, D., Chandran, N., Gupta, D.: Llama: A low latency math library for secure inference. Proceedings on Privacy Enhancing Technologies **4**, 274–294 (2022)

17. Ryffel, T., Tholoniat, P., Pointcheval, D., Bach, F.: Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. Proceedings on Privacy Enhancing Technologies **1**, 291–316 (2022)

18. Doerner, J., Shelat, A.: Scaling oram for secure computation. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 523–535 (2017)

19. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: Annual International Cryptology Conference. pp. 420–432. Springer (1992)

20. Kelkar, M., Le, P.H., Raykova, M., Seth, K.: Secure poisson regression. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 791–808 (2022)

21. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: 42nd Annual Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)

22. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. Journal of the ACM (JACM) **33**(4), 792–807 (1986)

23. Dinur, I., Goldfeder, S., Halevi, T., Ishai, Y., Kelkar, M., Sharma, V., Zaverucha, G.: Mpc-friendly symmetric cryptography from alternating moduli: Candidates, protocols, and applications. In: Annual International Cryptology Conference. pp. 517–547. Springer (2021)

24. Ryffel, T., Trask, A., Dahl, M., Wagner, B., Mancuso, J., Rueckert, D., Passerat-Palmbach, J.: A generic framework for privacy preserving deep learning. arXiv preprint arXiv:1811.04017 (2018)

A Dual Distributed Comparison Function

Dual Distributed Comparison Function (DDCF) is variant of DCF, is defined as:

$$f^{<}_{\alpha,\beta_0,\beta_1}(x) = \begin{cases} \beta_0 & x < \alpha \\ \beta_1 & \text{else} \end{cases} \tag{2}$$

where $x, \alpha \in \mathbb{U}_N, \beta_0, \beta_1 \in \{0,1\}$, and $\beta_0 \neq \beta_1$.

Note that $f^{<}_{\alpha,\beta_1,\beta_2}(x) = \beta_2 + f^{<}_{\alpha,\beta_1-\beta_2}(x)$ and thus DDCF scheme can be constructed by DCF scheme. Algorithm 7 show details of DDCF, which is slightly modified compared to [14] and the output group is guaranteed to be $\mathbb{Z}_2$.

Algorithm 7 DDCF: $(\mathsf{Gen}^{\mathsf{DDCF}}_n, \mathsf{Eval}^{\mathsf{DDCF}}_n)$

```
$\mathsf{Gen}^{\mathsf{DDCF}}_n(1^\lambda, \alpha, \beta_0, \beta_1)$
1: Let $\beta = \beta_1 - \beta_2$.
2: $(k_0^{(n)}, k_1^{(n)}) \leftarrow \mathsf{Gen}^{<}_n(1^\lambda, \alpha, \beta)$.
3: Sample random $r_0, r_1 \leftarrow\$ \{0,1\}$, s.t. $r_0 \oplus r_1 = \beta_2$.
4: For $b \in \{0,1\}$, let $k_b = k_b^{(n)} || r_b$.
5: return $(k_0, k_1)$.
$\mathsf{Eval}^{\mathsf{DDCF}}_n(b, k_b, x)$
1: Parse $k_b = k_b^{(n)} || r_b$.
2: $y_b^{(n-1)} \leftarrow \mathsf{Eval}^{<}_n(b, k_b^{(n)}, x)$.
3: $v_b = y_b^{(n-1)} \oplus r_b$.
4: return $v_b$.
```

B Proof of Theorem 2

In this section, we will demonstrate the correctness and privacy of the DCF scheme outlined in Section 5.2.

Proof. We will first address the correctness and then the privacy of the proposed construction $(\mathsf{Gen}, \mathsf{Eval})$.

Correctness: We demonstrate that the $s^{(i)}$ and $t^{(i)}$ generated by Eval match those set by Gen. This can be proven using mathematical induction: Let $x = x_1 x_2 \cdots x_n, \alpha = \alpha_1 \alpha_2 \cdots \alpha_n$, and $v_0 = \mathsf{Eval}^{<}_n(0, k_0, x), v_1 = \mathsf{Eval}^{<}_n(1, k_1, x)$.

1. When $n = 1$, As per line 1 of the algorithm $\mathsf{Eval}^{<}_n(b, k_b, x)$, the $\{s^{(0)}, t^{(0)}\}$ generated by Eval is consistent with the $\{s^{(0)}, t^{(0)}\}$ set by Gen. Since $k_b = s^{(0)} || CW^{(1)}$, it follows that $CW^{(1)} = s_{CW} || v^L_{CW} || t^L_{CW} || v^R_{CW} || t^R_{CW}$ and $G(s^{(0)}) = \hat{s}^L || \hat{v}^L || \hat{t}^L || \hat{s}^R || \hat{v}^R || \hat{t}^R$. We know that $v_b = (1 - x_1) \cdot v^L \oplus x_1 \cdot v^R$, where $v^L =$

$\hat{v}^L \oplus (t^{(0)} \cdot v^L_{CW}) = \hat{v}^L \oplus (b \cdot v^L_{CW})$ and $v^R = \hat{v}^R \oplus (t^{(0)} \cdot v^R_{CW}) = \hat{v}^R \oplus (b \cdot v^R_{CW})$ according to line 5 and line 6 of Eval. And according to the definition of $CW^{(1)}$ and $G(s^{(0)})$ in Gen, $v^L_{CW} = v^L_0 \oplus v^L_1 \oplus (\alpha_i \cdot \beta), v^R_{CW} = v^R_0 \oplus v^R_1$ and $\hat{v}^L = v^L_b, \hat{v}^R = v^R_b$, and then $v_b = ((1 - x_1) \cdot (b \oplus v^L_0)) \oplus (x_1 \cdot v^R_0)$, thus $v_0 \oplus v_1 = (1 - x_1) \cdot (\alpha_1 \cdot \beta) = \beta \cdot \mathbf{1}\{x_1 < \alpha_1\} = \beta \cdot \mathbf{1}\{x < \alpha\}$.

2. Assuming $n = i$, $\{s^{(j)}, t^{(j)}\}_{j=1,\cdots,i-1}$ generated by Eval are consistent with those set by Gen. When $n = i+1$, according to lines 7 to 12 of Eval, when $x_i = 0$ then $s^{(i)} = s^L, t^{(i)} = t^L$; when $x_i = 1$ then $s^{(i)} = s^R, t^{(i)} = t^R$. Since $s^L = \hat{s}^L \oplus (t^{(i-1)} \cdot s_{CW}), s^R = \hat{s}^R \oplus (t^{(i-1)} \cdot s_{CW}), t^L = \hat{t}^L \oplus (t^{(i-1)} \cdot t^L_{CW}), t^R = \hat{t}^R \oplus (t^{(i-1)} \cdot t^R_{CW})$ where $s_{CW}, t^L_{CW}, t^R_{CW}$ is an element of $CW^{(i)}$ and $\hat{s}^L, \hat{s}^R, \hat{t}^L, \hat{t}^R$ are an element of $G(s^{(i-1)})$, it follows that $\{s^{(j)}, t^{(j)}\}_{j=1,\cdots,i}$ are consistent with those generated by $\mathsf{Gen}^<_n(1^\lambda, \alpha)$. Finally, according to lines 15 to 16 in Gen, we can conclude that $\{s^{(j)}, t^{(j)}\}_{j=1,\cdots,i}$ generated by Eval is consistent with those set by Gen.

Therefore, it has been established that $\{s^{(j)}, t^{(j)}\}_{j=1,\cdots,n}$ generated by algorithm Eval are consistent with those set by Gen, As a result, $v_0 \oplus v_1 = \beta$ when $x < \alpha$, and 0 otherwise. Thus, $\mathsf{Eval}^<_n(0, k_0, x) \oplus \mathsf{Eval}^<_n(1, k_1, x) = f^<_{\alpha,\beta}(x)$, that is, $\Pr\Big[\mathsf{Eval}(0, k_0, x) \oplus \mathsf{Eval}(1, k_1, x) = f^<_{\alpha,\beta}(x)\Big] = 1$.

Privacy: We prove that each party's key k_b is pseudorandom. This will be done via a sequence of hybrids, where in each step we replace another correction word $CW^{(i)}$ within the key from being honestly generated to being random.

The high-level argument for privacy is as follows. Each party $b \in \{0, 1\}$ starts with a random seed $s^{(0)}_b$ that is completely unknown to the other party. In each level of key generation (for $i = 1$ to n), the parties apply a PRG to their seed $s^{(i-1)}_b$ to generate six items: namely, two seeds s^L_b, s^R_b, two resulting bits v^L_b, v^R_b and two control bits t^L_b, t^R_b. This process will always be performed on a seed that appears completely random and unknown from the view of the other party; because of this, the security of the PRG guarantees that the six items appear similarly random and unknown given the view of the other party. The ith level correction word $CW^{(i)}$ will "use up" the secret randomness of 5 of these 6 pieces: the two bits t^L_b, t^R_b, the resulting bits v^L_b, v^R_b and the seed s^{Lose}_b for $\mathsf{Lose} \in \{L, R\}$ corresponding to the direction exiting the special evaluation path α: i.e. $\mathsf{Lose} = L$ if $\alpha_i = 1$ and $\mathsf{Lose} = R$ if $\alpha_i = 0$. However, given this $CW^{(i)}$, the remaining seed s^{Keep}_b for $\mathsf{Keep} \neq \mathsf{Lose}$ still appears random to the other party. The argument then continues in similar fashion to the next level, beginning with seeds s^{Keep}_b.

For each $j \in \{1, \cdots, n\}$, we will consider a distribution Hyb_j defined roughly as follows:

1. $s^{(0)}_b \leftarrow \{0, 1\}^\lambda$ chosen at random (honestly), and $t^{(0)}_b = b$.
2. $CW^{(1)}, \cdots, CW^{(j)} \leftarrow \{0, 1\}^{\lambda+1}$ chosen at random.
3. For $i \leq j$, $s^{(i)}_b || v^{(i)}_b || t^{(i)}_b$ computed honestly, as a function of $s^{(0)}_b || v^{(0)}_b || t^{(0)}_b$ and $CW^{(1)}, \cdots, CW^{(j)}$.

4. For j, the other party's seed $s_{1-b}^{(j)} \leftarrow \{0,1\}^\lambda$ and the resulting bit $v_{1-b}^{(j)}$ are chosen at random, and $t_{1-b}^{(j)} = 1 - t_b^{(j)}$.
5. for $i > j$: the remaining values $s_b^{(i)}||v_b^{(i)}||t_b^{(i)}, s_{1-b}^{(i)}||v_{1-b}^{(i)}||t_{1-b}^{(i)}, CW^{(i)}$ are all computed honestly as a function of the previously chosen values.
6. The output of the experiment is $k_b := s_b^{(0)}||CW^{(1)}||\cdots||CW^{(n)}$.

Formally, Hyb_j is fully described in algorithm. Note that when $j = 0$, this experiment corresponds to the honest key distribution, whereas when $j = n$ this yields a completely random key k_b. We claim that each pair of adjacent hybrids $j-1$ and j will be indistinguishable based on the security of the pseudorandom generator.

Our proof follows from the following three lemmas:

Lemma 1. *For every $b \in \{0,1\}$, $\alpha \in \{0,1\}^n, \beta \in \{0,1\}$, it holds that*

$$\{k_b \leftarrow \mathsf{Hyb}_0(1^\lambda, b, \alpha, \beta)\} \equiv \{k_b : (k_0, k_1) \leftarrow \mathsf{Gen}_n^<(1^\lambda, \alpha, \beta\}$$

Lemma 2. *For every $b \in \{0,1\}$, $\alpha \in \{0,1\}^n, \beta$, it holds that*

$$\{k_b \leftarrow \mathsf{Hyb}_{n+1}(1^\lambda, b, \alpha, \beta)\} \equiv \{k_b \leftarrow U\}$$

Note that Lemma 1 and Lemma 2 follow directly by construction of Hyb_j.

Lemma 3. *There exists a polynomial p' such that for any (T, ϵ_{PRG}) -secure pseudorandom generator G, then for every $j \in \{0,1,\cdots n-1\}$, every $b \in \{0,1\}, \alpha \in \{0,1\}^n, \beta \in \{0,1\}$, and every non-uniform adversary $\mathcal{A}$ running in time $T' \leq T - p'(\lambda)$, it holds that*

$$\Big| \Pr\big[k_b \leftarrow \mathsf{Hyb}_{j-1}(1^\lambda, b, \alpha, \beta); c \leftarrow \mathcal{A}(1^\lambda, k_b) : c = 1\big]$$
$$- \Pr\big[k_b \leftarrow \mathsf{Hyb}_j(1^\lambda, b, \alpha, \beta); c \leftarrow \mathcal{A}(1^\lambda, k_b) : c = 1\big]\Big| < \epsilon_{\mathsf{PRG}}$$

Proof. Fix an arbitrary $j \in \{0,1,\cdots n-1\}, b \in \{0,1\}, \alpha \in \{0,1\}^n$ and $\beta \in \{0,1\}$. Given a Hyb-distinguishing adversary $\mathcal{A}$ with advantage ϵ for these values, we construct a corresponding PRG adversary $\mathcal{B}$. Recall that in the PRG challenge for G, the adversary $\mathcal{B}$ is given a value r that is either computed by sampling a seed $s \leftarrow \{0,1\}^\lambda$ and computing $r = G(s)$, or sampling a random $r \leftarrow \{0,1\}^{2(\lambda+2)}$.

Now, consider $\mathcal{B}$'s success in the PRG challenge as a function of $\mathcal{A}$'s success in distinguishing Hyb_{j-1} from Hyb_j. If r is computed pseudorandomly, then it is clear the generated k_b is distributed as $\mathsf{Hyb}_{j-1}(1^\lambda, b, \alpha, \beta)$.

It remains to show that if r was sampled at random then the generated k_b is distributed as $\mathsf{Hyb}_j(1^\lambda, b, \alpha, \beta)$. That is, if r is random, then the corresponding computed values of $s_{1-b}^{(j)}$ and $CW^{(j)}$ are distributed randomly conditioned on the values of $s_b^{(0)}||t_b^{(0)}||CW^{(j)}||\cdots||CW^{(j-1)}$, and the value of $t_{1-b}^{(j)}$ is given by $1 - t_b^{(j)}$. Note that all remaining values (for "level" $i > j$) are computed as a function of the values up to "level" j.

First, consider $CW^{(j)}$, computed in four parts:

```
$\mathsf{Hyb}_j(1^\lambda, b, \alpha, \beta)$
 1: Let $\alpha = \alpha_1, \cdots, \alpha_n \in \{0,1\}^n$ be the bit decomposition of $\alpha$.
 2: Sample random $s_b^{(0)} \leftarrow\$ \{0,1\}^\lambda$ and let $v_b^{(0)} = v_{1-b}^{(0)} \leftarrow 0, t_b^{(0)} = b, t_{1-b}^{(0)} = 1-b$.
 3: for $i = 1$ to $n$ do
 4:   if $i < j$ then
 5:     Sample $CW^{(j)} \leftarrow\$ \{0,1\}^\lambda \times \{0,1\} \times \{0,1\}^2$.
 6:   else
 7:     if $i = j$ then
 8:       Sample random $s_{1-b}^{(j-1)} \leftarrow\$ \{0,1\}^\lambda$ and let $t_{1-b}^{(j-1)} = 1 - t_b^{(j-1)}$.
 9:     end if
10:     $CW^{(i)} = \mathsf{CompCW}(i, \alpha_i, G(s_b^{(i-1)}), G(s_{1-b}^{(i-1)}), \beta)$.
11:     $(s_{1-b}^{(i)}, (t_{1-b}^{(i)}) = \mathsf{NextST}(1-b, i, t_{1-b}^{(i-1)}, s_{1-b}^{\mathsf{Keep}}||t_{1-b}^{\mathsf{Keep}}, CW^{(i)})$.
12:   end if
13:   $(s_b^{(i)}, t_b^{(i)}) = \mathsf{NextST}(b, i, t_b^{(1-i)}, s_b^{\mathsf{Keep}}||t_{1-b}^{\mathsf{Keep}}, CW^{(i)})$.
14: end for
15: Let $k_b = s_b^{(0)}||CW^{(1)}|| \cdots ||CW^{(n)}$
16: return $k_b$

$\mathsf{CompCW}(i, \alpha_i, S_b^{(i-1)}, S_{1-b}^{(i-1)}, \beta)$
 1: Parse $S_{1-b}^{(i-1)} = s_{1-b}^L||v_{1-b}^L||t_{1-b}^L||s_{1-b}^R||v_{1-b}^R||t_{1-b}^R$.
 2: Parse $S_b^{(i-1)} = s_b^L||v_b^L||t_b^L||s_b^R||v_b^R||t_b^R$.
 3: if $\alpha_i = 0$ then
 4:   Set $\mathsf{Keep} \leftarrow L, \mathsf{Lose} \leftarrow R$
 5: else
 6:   Set $\mathsf{Keep} \leftarrow R, \mathsf{Lose} \leftarrow L$
 7: end if
 8: $s_{CW} \leftarrow s_0^{\mathsf{Lose}} \oplus s_1^{\mathsf{Lose}}$.
 9: $v_{CW}^L \leftarrow v_0^L \oplus v_1^L \oplus (\alpha_i \cdot \beta)$
10: $t_{CW}^L \leftarrow t_0^L \oplus t_1^L \oplus \alpha_i \oplus 1$, and $t_{CW}^R \leftarrow t_0^R \oplus t_1^R \oplus \alpha_i$.
11: return $CW^{(i)} \leftarrow s_{CW}||v_{CW}^L||t_{CW}^L||t_{CW}^R$

$\mathsf{NextST}(x, i, t_x^{(i-1)}, s_x^{\mathsf{Keep}}||t_x^{\mathsf{Keep}}, CW^{(i)})$
 1: Parse $CW^{(i)} = s_{CW}||v_{CW}^L||t_{CW}^L||t_{CW}^R$.
 2: $s_x^{(i)} \leftarrow s_x^{\mathsf{Keep}} \oplus t_x^{(i-1)} \cdot s_{CW}$
 3: $t_x^{(i)} \leftarrow t_x^{\mathsf{Keep}} \oplus t_x^{(i-1)} \cdot t_{CW}^{\mathsf{Keep}}$.
 4: return $(s_x^{(i)}, t_x^{(i)})$
```

```
PRG adversary $\mathcal{B}(1^\lambda, (j, b, \alpha, \beta), r)$ :
 1: Let $\alpha = \alpha_1, \cdots, \alpha_n \in \{0,1\}^n$ be the bit decomposition of $\alpha$.
 2: Sample random $s_b^{(0)} \leftarrow\!\!\$ \{0,1\}^\lambda$ and let $v_b^{(0)} \leftarrow 0, t_b^{(0)} = b$ for $b = 0, 1$.
 3: for $i = 1$ to $(j-1)$ do
 4:     Sample $CW^{(i)} \leftarrow\!\!\$ \{0,1\}^\lambda \times \{0,1\} \times \{0,1\}^2$.
 5:     Parse $CW^{(i)} = s_{CW}||v_{CW}^L||t_{CW}^L||t_{CW}^R$.
 6:     Expand $s_b^L||v_b^L||t_b^L||s_b^R||v_b^R||t_b^R = G(s_b^{(i-1)})$.
 7:     if $\alpha_i = 0$ then
 8:         Set $\mathsf{Keep} \leftarrow L, \mathsf{Lose} \leftarrow R$
 9:     else
10:         Set $\mathsf{Keep} \leftarrow R, \mathsf{Lose} \leftarrow L$
11:     end if
12:     $(s_b^{(i)}, t_b^{(i)}) = \mathsf{NextST}(b, i, t_b^{(1-i)}, s_b^{\mathsf{Keep}}, t_{1-b}^{\mathsf{Keep}}, CW^{(i)})$.
13:     Take $t_{1-b}^{(i)} = 1 - t_b^{(i)}$
14: end for
15: Expand $s_b^L||v_b^L||t_b^L||s_b^R||v_b^R||t_b^R = G(s_b^{(j-1)})$.
16: Set $s_b^L||v_b^L||t_b^L||s_b^R||v_b^R||t_b^R = r$ (the PRG challenge).
17: $CW^{(j)} = \mathsf{CompCW}(j, \alpha_j, r, G(s_b^{(j-1)}), \beta)$.
18: if $\alpha_j = 0$ then
19:     set $\mathsf{Keep} \leftarrow L, \mathsf{Lose} \leftarrow R$
20: else
21:     Set $\mathsf{Keep} \leftarrow R, \mathsf{Lose} \leftarrow L$
22: end if
23: Compute $(s_x^{(j)}, t_x^{(j)}) = \mathsf{NextST}(x, j, t_x^{(j-1)}, s_x^{\mathsf{Keep}}||t_x^{\mathsf{Keep}}, CW^{(j)})$, for both $x \in \{0,1\}$.
24: Set $P = [s_0^L||v_0^L||t_0^L||s_0^R||v_0^R||t_0^R], [s_1^L||v_1^L||t_1^L||s_1^R||v_1^R||t_1^R]$.
25: Compute $(CW^{(j+1)}||\cdots||CW^{(n)}) = \mathsf{RemainingKey}(\alpha, j, CW^{(1)}||\cdots||CW^{(j)}, P)$.
26: return $k_b = s_b^{(0)}||CW^{(1)}||\cdots||CW^{(n)}$.
$\mathsf{RemainingKey}(\alpha, j, CW^{(1)}||\cdots||CW^{(j)}, t_0^{(j)}, t_1^{(j)}, P)$.
 1: Parse $P = [s_0^L||v_0^L||t_0^L||s_0^R||v_0^R||t_0^R], [s_1^L||v_1^L||t_1^L||s_1^R||v_1^R||t_1^R]$.
 2: for $i = (j+1)$ to $n$ do
 3:     Expand $s_x^L||v_x^L||t_x^L||s_x^R||v_x^R||t_x^R = G(s_x^{(i-1)})$ for both $x \in \{0,1\}$.
 4:     if $\alpha_i = 0$ then
 5:         set $\mathsf{Keep} \leftarrow L, \mathsf{Lose} \leftarrow R$
 6:     else
 7:         Set $\mathsf{Keep} \leftarrow R, \mathsf{Lose} \leftarrow L$
 8:     end if
 9:     $CW^{(i)} = \mathsf{CompCW}(i, \alpha_i, [s_0^L||v_0^L||t_0^L||s_0^R||v_0^R||t_0^R], [s_1^L||v_1^L||t_1^L||s_1^R||v_1^R||t_1^R], \beta)$.
10:     Compute $(s_x^{(i)}, t_x^{(i)}) = \mathsf{NextST}(x, i, t_x^{(i-1)}, s_x^{\mathsf{Keep}}||t_x^{\mathsf{Keep}}, CW^{(i)})$, for both $x \in \{0,1\}$.
11: end for
12: return $(CW^{(j)}||CW^{(j+1)}||\cdots||CW^{(n)})$
```

- $s_{CW} = s_b^{\mathsf{Lose}} \oplus s_{1-b}^{\mathsf{Lose}}$.
- $v_{CW}^L = v_b^L \oplus v_{1-b}^L \oplus (\alpha_j \cdot \beta)$.
- $t_{CW}^L = t_b^L \oplus t_{1-b}^L \oplus \alpha_j \oplus 1$.
- $t_{CW}^R = t_b^R \oplus t_{1-b}^R \oplus \alpha_j$.

In the case that r is random, then $s_{1-b}^{\mathsf{Lose}}, v_{1-b}^L, v_{1-b}^R, t_{1-b}^L$, and t_{1-b}^R (no matter the value of $\mathsf{Lose} \in \{L, R\}$) are each perfect one-time pads. So, $CW^{(j)} = s_{CW}||v_{CW}^L||t_{CW}^L||t_{CW}^R$ is indeed distributed uniformly.

Now, condition on $CW^{(j)}$ as well, and consider the value of $s_{1-b}^{(j)}$. Depending on the value of $t_{1-b}^{(j-1)}, s_{1-b}^{(j)}$ is selected either as s_{1-b}^{Keep} or $s_{1-b}^{\mathsf{Lose}} \oplus s_{CW}$. However, s_{1-b}^{Keep} is distributed uniformly conditioned on the view thus far, and so in either case the resulting value is again distributed uniformly.

Finally, consider the value of $t_{1-b}^{(j)}$. Note that both $t_b^{(j)}$ and $t_{1-b}^{(j)}$ are computed as per NextST, as a function of $t_1^{(j-1)}$ and $t_{1-b}^{(j-1)}$, respectively (and $t_{1-b}^{(j-1}$ was set to $1 - t_b^{(j-1)}$). In particular,

$$\begin{aligned}
t_b^{(j)} \oplus t_{1-b}^{(j)} &= (t_b^{\mathsf{Keep}} \oplus t_b^{(i-1)} \cdot t_{CW}^{\mathsf{Keep}}) \oplus (t_{1-b}^{\mathsf{Keep}} \oplus t_{1-b}^{(i-1)} \cdot t_{CW}^{\mathsf{Keep}}) \\
&= t_b^{\mathsf{Keep}} \oplus t_{1-b}^{\mathsf{Keep}} \oplus (t_b^{(i-1)} \oplus t_{1-b}^{(i-1)}) \cdot t_{CW}^{\mathsf{Keep}} \\
&= t_b^{\mathsf{Keep}} \oplus t_{1-b}^{\mathsf{Keep}} \oplus 1 \cdot (t_0^{\mathsf{Keep}} \oplus t_1^{\mathsf{Keep}} \oplus 1) \\
&= 1
\end{aligned}$$

Combining these pieces, we have that in the case of a random PRG challenge r, the resulting distribution of k_b as generated by $\mathcal{B}$ is precisely distributed as is $\mathsf{Hyb}_j(1^\lambda, b, \alpha, \beta)$. Thus, the advantage of $\mathcal{B}$ in the PRG challenge experiment is equivalent to the advantage ϵ of $\mathcal{A}$ in distinguishing $\mathsf{Hyb}_{j-1}(1^\lambda, b, \alpha, \beta)$ from $\mathsf{Hyb}_j(1^\lambda, b, \alpha, \beta)$. The runtime of $\mathcal{B}$ is equal to the runtime of $\mathcal{A}$ plus a fixed polynomial $p'(\lambda)$. Thus for any $T' \leq T - p'(\lambda)$, it must be that the distinguishing advantage ϵ of $\mathcal{A}$ is bounded by ϵ_{PRG}. □

This concludes the proof. □