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

Peng Yang
Harbin Institute of Technology, ShenZhen, China
stuyangpeng@stu.hit.edu.cn

Zoe Lin Jiang*
Harbin Institute of Technology, ShenZhen, China
Guangdong Provincial Key Laboratory of Novel Security Intelligence Technology
zoeljiang@hit.edu.cn

Shiqi Gao
Harbin Institute of Technology, ShenZhen, China
200111514@stu.hit.edu.cn

Jiehang Zhuang
Harbin Institute of Technology, ShenZhen, China
jiehangzhuang@stu.hit.edu.cn

Hongxiao Wang
The University of Hong Kong, HKSAR, China
hxwang@cs.hku.hk

Junbin Fang
Jinan University, GuangZhou, China
tjunbinfang@jnu.edu.cn

Siuming Yiu
The University of Hong Kong, HKSAR, China
smyiu@cs.hku.hk

Yulin Wu
Harbin Institute of Technology, ShenZhen, China
Guangdong Provincial Key Laboratory of Novel Security Intelligence Technology
wuyulin@hit.edu.cn

Xuan Wang
Harbin Institute of Technology, ShenZhen, China
Guangdong Provincial Key Laboratory of Novel Security Intelligence Technology
wangxuan@cs.hitsz.edu.cn

ABSTRACT

Privacy-preserving neural network enables multiple parties to jointly train neural network models without revealing sensitive data. However, its practicality is greatly limited due to the low efficiency caused by massive communication costs and a deep dependence on a trusted dealer. In this paper, we propose a communication-efficient secure two-party neural network framework, FssNN, to enable practical secure neural network training and inference. In FssNN, we reduce communication costs of computing neural network operators in the online and offline phases by combining additive secret sharing (SS) and function secret sharing (FSS), and eliminate the dependence on the trusted dealer based on a distributed key generation scheme. First, by integrating correction words and designing a more compact key generation algorithm, we propose a key-reduced distributed comparison function (DCF, a FSS scheme for comparison functions) with the smallest key sizes to enable efficient computation of non-linear layer functions in the offline phase. Secondly, by leveraging the proposed DCF and combining SS and FSS, we construct online-efficient computation protocols for neural network operators, such as Hadamard product, ReLU and DReLU, and reduce the online communication costs to about 1/2 of that of the state-of-the-art solution. Finally, by utilizing MPC-friendly pseudorandom generators, we propose a distributed DCF key generation scheme to replace the trusted dealer and support a larger input domain than the state-of-the-art solution.

Using FssNN, we perform extensive secure training and inference evaluations on various neural network models. Compared with the state-of-the-art solution AriaNN (PoPETs'22), we reduce the communication costs of secure training and inference by approximately 25.4% and 26.4% respectively, while keeping the accuracy of privacy-preserving training and inference close to that of plaintext training and inference.

KEYWORDS

Privacy-preserving neural network; Secure multi-party computation; Additive secret sharing; Function secret sharing.

Note: this is a full version of the paper " Communication-efficient Secure Neural Network via Key-reduced Distributed Comparison Function".

1 INTRODUCTION

Machine learning using neural networks (NN) is widely applied in many practical scenarios such as healthcare prediction, financial services, auto driving and policy making. Jointly training neural network models by collecting data from multiple parties can greatly improve the models' accuracy and generalization their capabilities, but data leakage issues and privacy protection regulations do not allow data to be shared in plain text[21]. Privacy-preserving neural network based on cryptographic methods such as homomorphic encryption[12], garbled circuit[27], and secret sharing[15] enables neural network training and inference in an encrypted state, which can effectively solve the conflicting problem of data sharing and privacy protection. Compared with these solutions based on homomorphic encryption and garbled circuit, secret sharing-based

*Zoe Lin Jiang is the corresponding author

solutions have significant advantages in computation efficiency and communication efficiency respectively, so they are regarded as the most promising solutions in practical applications.

However, secret sharing-based solutions require a large number of calculations of non-linear functions (such as activation function ReLU, etc.) during the training, which incurs huge computation overhead and communication costs. Compared with plaintext training, the running-time is several orders of magnitude slower[18, 22, 25], severely limiting secret sharing-based solutions' practicality. In order to reduce communication costs of computing non-linear function, in 2019, Boyle et al. [5] propose a secure two-party computation protocol based on function secret sharing[3, 4] in an offline-online paradigm. Their online communication rounds of computing activation function can be reduced to a constant round by using the distributed comparison function (DCF, a FSS scheme for comparison functions), but in the offline phase it still requires a huge communication overhead to precompute the DCF key and relies heavily on a trusted dealer. In 2021, Ryffel et al.[23] reduce DCF key sizes by designing a compact key generation and evaluation algorithm and present a secure ReLU protocol based on their DCF construction, thus proposing an online-efficient neural network training and inference framework, but the secure protocols designed for ReLU incorporate a 1-bit error and rely heavily on the trusted dealer in the offline phase. In 2021, Boyle et al.[2] further decrease the DCF key sizes (the key sizes are $n(\lambda+3)+\lambda+1$ bits where n is the input size and λ is the security parameter.) and construct a distributed DCF key generation scheme by extending the Doerner-Shelat protocol to replace the trusted dealer, but this scheme is only suitable for a small input domain. Therefore, the existing DCF construction faces the problems of large key sizes and poor practicality of the distributed DCF key generation scheme.

In response to above problems, we propose a communication-efficient and secure two-party neural network framework, FssNN, based on a key-reduced DCF with compact additive construction. We reduce communication costs in the online and offline phases by combining additive secret sharing and function secret sharing, and replace the trusted dealer by designing a distributed key generation scheme based on MPC-friendly pseudorandom generators, thereby greatly improving practical performance of privacy-preserving neural network. First, by integrating correction words and designing a more compact key generation algorithm, we propose a key-reduced DCF with fewer key sizes than [2] to achieve efficient computation of non-linear functions in the offline phase. Secondly, by leveraging the proposed DCF construction and combining additive secret sharing with function secret sharing, we construct online-efficient secure computation protocols for Hadamard product, ReLU and DReLU, which can reduce the communication costs of online phase to about 1/2 of that of AriaNN[23]. Finally, by introducing 2PC-friendly pseudorandom generators, we propose a distributed DCF key generation scheme based on secure two-party computation to replace the trusted dealer and support a larger input domain than the state-of-the-art solution [2]. Theoretical analysis shows that compared with he state-of-the-art DCF construction[2], our proposed DCF construction reduces the key sizes from $n(\lambda+3)+\lambda+1$ bits to $(\lceil n-\log\lambda\rceil)(\lambda+3)+2\lambda$ bits. For the typical parameter $n=32$ bits and $\lambda=127$ bits, the key reduction is 790 bits with a decrease of 17.9%. Furthermore, FssNN has fewer online rounds and lower communication complexity compared with existing framework ABY2.0[22] and AriaNN[23], while FssNN requires more computation in the offline phase.

In the experiment, we execute FssNN with Python in Ubuntu and implement an end-to-end system for secure two-party training and inference, and we conduct experimental tests on various neural network models on MNIST dataset and so on. Experimental results show that compared with the start-of-the-art work AriaNN[23], we reduce the communication costs of secure training and inference by roughly 25.4% and 26.4% respectively, while keeping the accuracy of secure training and inference close to that of plaintext counterpart.

1.1 Related Work

Privacy-preserving neural network built on MPC has emerged as a flourishing research area in the past few years. Existing works use secure computation protocols based on secret sharing to compute linear functions and protocols based on secret sharing (SS), garbled circuit (GC), or function secret sharing (FSS) to compute non-linear functions. These works also adopt the offline-online computation mode[7] to obtain an efficient online phase by moving a majority of computation and communication costs to the offline phase.

In SS-based and GC-based solutions, SecureML[21] is the first privacy-preserving neural network framework and implementation with secure two-party computation (2PC) based on SS and GC. It enables the secure training by combining Boolean secret sharing, arithmetic secret sharing and Yao's secret sharing[8], but the conversion between three types of secret shares incurs huge communication costs. ABY2.0[22] reduces online communication rounds and communication costs by designing an efficient secret shares conversion protocol, and improves the efficiency of secure two-party neural network. However, these 2PC frameworks have a number of communication rounds linear to the circuit depth, resulting in extremely high communication costs and latency. ABY3[20] and Falcon[25] are proposed to tackle secure training by leveraging three-party computation (3PC), and Trident[6] and Tetrad[18] are the secure four-party computation (4PC) framework for privacy-preserving neural network training. Compared with 2PC frameworks, these 3PC and 4PC frameworks have fewer communication rounds (still linear with circuit depth)[11], but impose a stronger security assumption (honest-majority rather than the dishonest-majority), the practicality is greatly limited.

In FSS-based solutions, non-linear functions are evaluated by using FSS-based 2PC protocols, which are optimal in terms of online communication and rounds[2]. AriaNN[23] is a low-interaction privacy-preserving neural network framework based on FSS by reducing the key sizes of distributed comparison function (DCF, a FSS scheme for comparisons) However, it still requires considerable online communication costs, and the secure protocols within AriaNN designed for ReLU incorporate a 1-bit error. BCG+21[2], LLAMA[16] and Grotto[24] provide secure computation protocols based on DCF for computing various math functions (e.g., comparison, reciprocal square root and piecewise polynomial), yet they currently can not provide support for training neural networks. Orca[17] enables secure inference and training by accelerating the computation of FSS-based 2PC protocols with GPUs, but the online phase in Orca requires additional communication rounds compared

with other FSS-based solutions, incurring the high communication latency. However, AriaNN [23], LLAMA[16], Grotto[24] and Orca[17] rely heavily on a unrealistic trusted dealer to generate DCF key in the offline phase. Although BCG+21[2] designs a distributed DCF key generation scheme by extending the Doerner-Shelat protocol[10] to replace the trusted dealer, it is only suitable to a small input domain ($\mathbb{Z}_{2^{16}}$ or smaller), which greatly limits its practicality.

The FSS-based secure neural network frameworks are summarized in Table 1.

Table 1: The FSS-based secure neural network frameworks. ○, ◐ and ● respectively represent that inference and training are not supported, only inference is supported and inference and training are both supported. "gate evaluation rounds" indicates the online communication rounds for a gate. "GPUs" indicates whether GPU implementation is provided.

Framework	inference & training	gate evaluation rounds	no trusted dealer	GPUs
BCG+21 [2]	○	0	✓	×
AriaNN [23]	●	0	×	✓
LLAMA [16]	◐	0	×	×
Grotto [24]	○	0	×	×
Orca [17]	●	$O(1)$	×	✓
FssNN	●	0	✓	×

1.2 Our Contributions

In this paper, we propose FssNN, a communication-efficient secure two-party neural network (NN) framework, to enable practical secure neural network training. FssNN has the small constant-round online communication complexity and low offline communication costs, and does not rely on the trusted dealer in the offline phase.

In details, our contributions can be summarized in the following three aspects:

- **Key-reduced distributed comparison function with compact additive construction**. By integrating correction words and designing a more compact key generation algorithm, we propose a key-reduced distributed comparison function (DCF) with fewer key sizes than the state-of-the-art DCF construction[2] from $n(\lambda+3)+\lambda+1$ bits to $(\lceil n-\log\lambda\rceil)(\lambda+3)+2\lambda$ bits where n is the input size and λ is the security parameter, thus reduce offline communication costs by 26.8% - 28.3% .
- **Online-efficient seucre neural network operators**. By leveraging the proposed DCF and combining additive secret sharing with function secret sharing, we construct online-efficient and secure computation protocols for neural network operators, such as Hadamard product, ReLU and DReLU, and reduce the online communication costs to about 1/2 of that of the state-of-the-art solution[23].
- **Distributed DCF key generation based MPC-friendly pseudorandom generators** By introducing MPC-friendly pseudorandom generators, we propose a distributed DCF key generation scheme to replace the trusted dealer and support a larger input domain ($\mathbb{Z}_{2^{32}}$ and above) than the state-of-the-art DCF key generation shceme[2].

1.3 Organization

Following basic notations and background on neural network and secure computation in § 2, § 3 presents the proposed FssNN where § 3.1 provide a high-level overview of FssNN, and § 3.2 and § 3.3 present secure computation protocols for linear layers and non-linear layers. § 4 presents theoretical analysis and experimental results, closing up with conclusion on § 5.

2 PRELIMINARIES

Notations. $\mathbb{Z}_{2^n}$ is a ring with arithmetic operations with each element identified by its n-bit binary representation. 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 $x_{[i]} \in \mathbb{Z}_2$ denotes x_i and $x_{[i,j)} \in \mathbb{Z}_{2^{j-i}}$ denotes the ring element corresponding to the bit-string $x_{j-1}||\cdots||x_i$ for $0 \le i < j \le n$. Denote scalar, vector and matrix by lowercase letter x, lowercase bold letter $\mathbf{x}$ and uppercase bold letter $\mathbf{X}$ respectively. Denote random sampling by $\in_R$ and security parameter by λ, and $\mathbf{1}\{b\}$ by the indicator function that outputs 1 when b is true and 0 otherwise. In this paper, we consider two parties and denote party b by P_b where $b \in \{0,1\}$ is party index.

2.1 Neural Network Training

Let $D = \{(\mathbf{x}_i, y_i) | i \in \{0,1,\cdots,m\}\}$ denotes training datasets where each data sample $\mathbf{x}_i$ contains d features with the corresponding output label y_i. Neural network is a computational process to learn a function g such that $g(\mathbf{x}_i) \approx y_i$ where g can be represented as a function of weight matrix $\mathbf{W}$ and input data $\mathbf{x}_i$. The neural network training procedure consists of two phases, namely forward propagation and backward propagation. The phase to calculate the predicted output $\hat{y}_i = g(\mathbf{W}, \mathbf{x}_i)$ is called *forward propagation*, which comprises of linear operations and a non-linear activation function. 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 value $\hat{y}_i$ and actual value y_i is defined, and $\mathbf{W}$ is calculated and updated by solving the optimization problem of $\text{argmin}_{\mathbf{W}} C(\mathbf{W})$. 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. 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 the coefficient matrix $\mathbf{W}$ is updated by $\mathbf{W} := \mathbf{W} - \alpha\nabla C(\mathbf{W})$, where α is the learning rate and $\nabla C(\mathbf{W})$ is the partial derivatives of the cost with respect to the changes in weight. The phase to calculate the change $\alpha\nabla C(\mathbf{W})$ is called *backward propagation*, where error rates are fed back through a neural network to update weight $\mathbf{W}$.

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 to allow for the use of vectorization techniques to accelerate computation.

2.2 Additive Secret Sharing

An additive secret sharing (SS) scheme splits a secret value into multiple shares that add up to the original secret value and none of the individual shareholders have enough information to reconstruct the secret value. In a two-party SS, P_0 with secret share $\langle x\rangle_0$ and P_1 with secret share $\langle x\rangle_1$ share the secret value $x \in \mathbb{Z}_{2^n}$, s.t. $x = (\langle x\rangle_0 + \langle x\rangle_1) \bmod 2^n$. We say that P_0 and P_1 hold together the secret share pair $\langle x\rangle$, which means that P_0 holds $\langle x\rangle_0$ and P_1 holds $\langle x\rangle_1$.

Sharing and Reconstruction. To realize the functionality $\mathcal{F}_{\mathsf{Share}}$ which additively shares a secret value $x \in \mathbb{Z}_{2^n}$, secret owner samples random $r \in \mathbb{Z}_{2^n}$, and sends $\langle x\rangle_b = (x - r) \bmod 2^n$ to P_b and sends $\langle x\rangle_{1-b} = r$ to P_{1-b}. To implement the functionality $\mathcal{F}_{\mathsf{Recon}}$ which opens an additively shared value $\langle x\rangle$, P_b sends $\langle x\rangle_b$ to P_{1-b} who computes $(\langle x\rangle_0 + \langle x\rangle_1) \bmod 2^n$ for $b = \{0, 1\}$. In the following text, we omit the modular operation for simplicity.

Addition and Multiplication. Functionalities $\mathcal{F}_{\mathsf{Add}}$ and $\mathcal{F}_{\mathsf{Mul}}$ add and multiply two shared values $\langle x\rangle$ and $\langle y\rangle$ respectively. It is easy to non-interactively add the shared values by having P_b compute $\langle z\rangle_b = \langle x\rangle_b + \langle y\rangle_b$. We overload the addition operation to denote the secure addition by $\langle x\rangle + \langle y\rangle$. To realize $\mathcal{F}_{\mathsf{Mul}}$, taking the advantage of Beaver's precomputed multiplication triples technique [1], the specific protocol Π_{Mul} works as follows: assume that P_0 and P_1 hold multiplication triples $\langle u\rangle, \langle v\rangle, \langle uv\rangle$ where $u, v \in_R \mathbb{Z}_{2^n}$, P_b locally computes $\langle e\rangle_b = \langle x\rangle_b - \langle u\rangle_b$ and $\langle f\rangle_b = \langle y\rangle_b - \langle v\rangle_b$ and then the two parties reconstruct $\langle e\rangle, \langle f\rangle$ to get e, f. Finally, P_b lets $\langle z\rangle_b = b \cdot e \cdot f + f \cdot \langle u\rangle_b + e \cdot \langle v\rangle_b + \langle uv\rangle_b$.

In the case of $n > 1$ (e.g., $n = 32$) which supports arithmetic operations (e.g., addition and multiplication), arithmetic share pair is denoted by $\langle\cdot\rangle^A$. In the case of $n = 1$ which supports Boolean operations (e.g., XOR and AND), Boolean share pair is denoted by $\langle\cdot\rangle^B$. In this paper, we mostly use the arithmetic share pair and denote it by $\langle\cdot\rangle$ for short.

Generating Multiplication Triples. Functionality $\mathcal{F}_{\mathsf{Gen}^{\mathsf{MT}}}$ generates multiplication triples $(\langle u\rangle, \langle v\rangle, \langle uv\rangle)$ consumed in Π_{Mul}. Typically, multiplication triples can be generated based on oblivious transfer (OT). In this paper, the protocol $\Pi_{\mathsf{Gen}^{\mathsf{MT}}}$ for $\mathcal{F}_{\mathsf{Gen}^{\mathsf{MT}}}$ is achieved by directly using VOLE-style OT generation scheme proposed in Ferret [26].

2.3 Function Secret Sharing

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

Definition 2.1. (FSS: Syntax). A two-party FSS scheme is a pair of algorithms (Gen, Eval) such that:

(1) $\mathsf{Gen}(1^\lambda, \hat{f})$ is a probabilistic polynomial-time (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 $\mathbb{G}^{\mathsf{in}}, \mathbb{G}^{\mathsf{out}}$.
(2) $\mathsf{Eval}(b, k_b, x)$ 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 2.2. (FSS: Correctness and Security). Let $\mathcal{F} = \{f\}$ be a function family. We say that (Gen, Eval) as in Definition 2.1 is an FSS scheme for $\mathcal{F}$ if it satisfies the following requirements:

(1) **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 $\Pr[\mathsf{Eval}(0, k_0, x) + \mathsf{Eval}(1, k_1, x) = f(x)] = 1$.
(2) **Security**: 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_i for f_i, 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 $\mathsf{Sim}_b(1^\lambda)$.

Distributed Comparison Function and its Variant. A special piecewise function, $f^{<}_{\alpha,\beta}(x)$, also referred to as a comparison function, outputs β if $x < \alpha$ and 0 otherwise. We refer to a 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 $f^{<}_{\alpha,\beta_0,\beta_1}(x)$ that outputs β_0 for $0 \le x < \alpha$ and β_1 for $x \ge \alpha$. Obviously, $f^{<}_{\alpha,\beta_0,\beta_1}(x) = \beta_1 + f^{<}_{\alpha,\beta_0-\beta_1}(x)$ and thus DDCF can be constructed by DCF.

Secure Two-party Computation via FSS. Recent work of Boyle et al. [2, 5] shows that FSS paradigm can be used to efficiently evaluate some function families in the two-party computation in the offline-online model, where Gen and Eval correspond to the offline phase and the online phase respectively. **In the offline phase**, a trusted dealer randomly samples mask r^{in} for each input wire w_{in} or $\mathsf{r}^{\mathsf{out}}$ for each output wire w_{out} in the computation circuit. For each gate g with w_{in} and w_{out}, the dealer constructs *offset function* $g^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}(x) := g(x - \mathsf{r}^{\mathsf{in}}) + \mathsf{r}^{\mathsf{out}}$, and runs Gen to generate FSS keys (k_0, k_1) corresponding to $g^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}$. Then the dealer sends k_b to P_b, and sends the corresponding mask r to P_b for circuit input and output wires w owned by P_b. **In the online phase**, P_b calculates the masked wire value $\hat{x} = x + \mathsf{r}^{\mathsf{in}}$ for each w_{in} with r^{in} owned by P_b, and sends it to P_{1-b}. Starting from the input gates, P_0 and P_1 compute gates in topological order to obtain masked output wire values. To compute a gate g with w_{in} and w_{out}, P_b uses Eval with FSS key k_b and masked input wire value $\hat{x} = x + \mathsf{r}^{\mathsf{in}}$ to obtain the masked output wire value $g(x) + \mathsf{r}^{\mathsf{out}}$. For output wires, they subtract the corresponding mask received from the dealer to obtain clear output values. In this paper, a secure two-party computation protocol is proposed to instantiate the trusted dealer.

2.4 Threat Model

We consider two-party computation secure against a *semi-honest* adversary, i.e., the corrupted party running the protocol honestly while trying to learn as much information as possible about others' input or function share. In this paper, we directly follow the definition of semi-honest security from [19].

Definition 2.3. Let $\mathcal{F} = (\mathcal{F}_0, \mathcal{F}_1)$ be a functionality. We say that Π securely realizes $\mathcal{F}$ in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time algorithm Sim_0 and Sim_1 such that:

$$\{(\mathsf{Sim}_0(1^n, x, \mathcal{F}_0(x,y)), \mathcal{F}(x,y))\}_{x,y,n} \overset{c}{\equiv} \{(\mathsf{view}_0^{\Pi}(x,y,n), \mathsf{output}^{\Pi}(x,y,n))\}_{x,y,n}$$

$$\{(\mathsf{Sim}_1(1^n, y, \mathcal{F}_1(x,y)), \mathcal{F}(x,y))\}_{x,y,n} \overset{c}{\equiv} \{(\mathsf{view}_1^{\Pi}(x,y,n), \mathsf{output}^{\Pi}(x,y,n))\}_{x,y,n}$$

where $x, y \in \{0,1\}^*$ such that $|x| = |y|$, and $n \in \mathbb{N}$.

In addition, modular sequential composition theorems [13, 19] are considered, and we prove protocols secure under the definition of semi-honest security from [19] and immediately derive their security under sequential composition. Our protocols invoke several sub-protocols and for ease of exposition we describe them using the hybrid model [19], which is the same as a real interaction except that the sub-protocol executions are replaced with calls to the corresponding trusted functionalities - protocol invoking $\mathcal{F}$ is said to be in the $\mathcal{F}$-hybrid model.

3 THE PROPOSED FSSNN

In this section, we present a high-level overview of FssNN framework in § 3.1, and provide detailed construction of secure linear layer functions and secure non-linear layer functions in § 3.2 and § 3.3 respectively.

3.1 The FssNN Overview

In this paper, we propose a secure two-party neural network framework, FssNN, to enable practical and secure training and inference. We decrease communication rounds and communication costs by utilizing additive secret sharing and key-reduced distributed comparison function (DCF, a function secret sharing scheme for comparisons), and replace the trusted dealer using a distributed DCF key generation scheme.

As shown in Figure 1, FssNN works as follows: parties P_0 and P_1 hold the secret shares of training datasets $\langle D\rangle_0$ and $\langle D\rangle_1$ respectively, and initialize $\langle \mathbf{W}\rangle_0$ and $\langle \mathbf{W}\rangle_1$ to be the all 0s locally. Then, for $b = 0, 1$, P_b randomly selects a training sample $(\langle \mathbf{x}_i\rangle_b, \langle y_i\rangle_b)$, and engages in a secure two-party SGD protocol (2PC-SGD) with P_{1-b} to update $\langle \mathbf{W}\rangle$ interactively, which involves two steps: ① forward propagation and ② backward propagation. During the forward propagation and the backward propagation, we need to securely compute linear layers (denoted by green solid line boxes) and non-linear layers (denoted by blue dashed line boxes). Finally, P_0 and P_1 select a new sample randomly and repeat the above process until the samples are used up, and finally P_b gets $\langle \mathbf{W}\rangle_b$. It can be seen from Figure 1 that the secure training is mainly divided into secure linear layers and secure non-linear layers.

To reduce the communication rounds and communication costs required to compute linear layers and non-linear layers, we propose a hybrid method combing additive secret sharing (SS) and function secret sharing (FSS) and adopt the offline-online paradigm to compute linear layers in one round of online communication and compute non-linear layers in a constant round of online communication. Figure 2 shows our secure linear layer protocols (denoted by blue dashed line boxes) and secure non-linear layer protocols (denoted by blue dashed line boxes) in the offline-online paradigm.

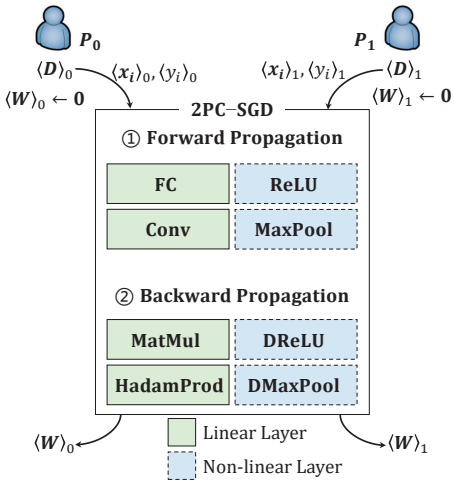

Figure 1: The Workflow of FssNN

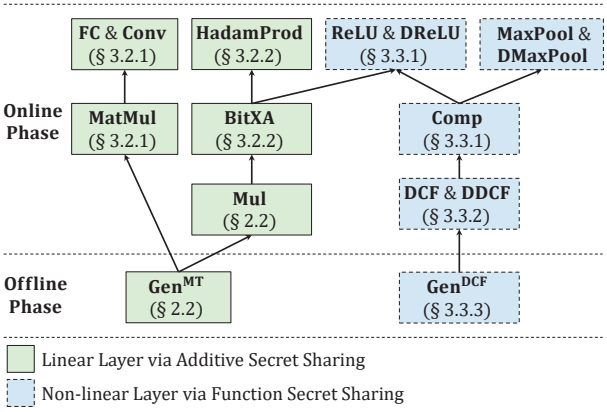

Figure 2: Secure linear layers and non-linear layers

For linear layers, by leveraging a SS-based secure two-party computation protocol, we generate multiplication triples ($\mathsf{Gen}^{\mathsf{MT}}$) in the offline phase and utilize these multiplication triples to compute linear layer functions such as MatMul, Conv, FC and HadamProd in one round of communication in the online phase. Among them, we propose a protocol, BitXA, that supports direct multiplication of a bit and an integer to reduce online communication costs, and extend it to HadamProd through vectorization techniques. **For non-linear layers**, we propose a key-reduced DCF scheme with compact additive construction and design a distributed DCF key generation scheme based on an MPC-friendly pseudorandom generator (PRG). In the offline phase, we design a distributed DCF key generation scheme ($\mathsf{Gen}^{\mathsf{DCF}}$) for the proposed key-reduced DCF to generate the DCF key rather than relying on a trusted dealer, and utilize the DCF key to compute non-linear layer functions such as ReLU, DReLU, MaxPool and DMaxPool with a constant-round online communication.

In § 3.2 and § 3.3, we will introduce the detailed construction of the secure linear layers and the secure non-linear layers respectively.

3.2 Construction of Secure Linear Layers

In this section, we present the detailed construction of linear layer functions, i.e., MatMul (§ 3.2.1, and FC as well as Conv) and HadamProd (§ 3.2.2), in the online and offline phases.

3.2.1 ***Secure Matrix Multiplication.*** By leveraging the vectorization technique in [21], secure scalar multiplication (Mul) introduced in § 2.2 can be easily extended to secure matrix multiplication (MatMul) where the multiplication triples are replaced by matrix multiplication triples. Given secret shares of matrices $\langle \mathbf{X} \rangle, \langle \mathbf{Y} \rangle$ held by P_0 and P_1 where $\mathbf{X} \in \mathbb{Z}_{2^n}^{m_1 \times m_2}, \mathbf{Y} \in \mathbb{Z}_{2^n}^{m_2 \times m_3}$, functionality $\mathcal{F}_{\mathsf{MatMul}}$ computes $\langle \mathbf{Z} \rangle$ s.t. $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$. Π_{MatMul} realizes the functionality $\mathcal{F}_{\mathsf{MatMul}}$ as follows: 1) In offline phase, P_b samples $\langle \mathbf{U} \rangle_b, \langle \mathbf{V} \rangle_b$ randomly, and then parties invoke $\mathsf{Gen}^{\mathsf{MT}}(\langle \mathbf{U} \rangle, \langle \mathbf{V} \rangle)$ to generate matrix multiplication triples $(\langle \mathbf{U} \rangle, \langle \mathbf{V} \rangle, \langle \mathbf{UV} \rangle)$. 2) In the online phase, parties open $\mathbf{X} - \mathbf{U}$ and $\mathbf{Y} - \mathbf{V}$, and then P_b computes $\langle \mathbf{Z} \rangle_b = b \cdot (\mathbf{X} - \mathbf{U}) \times (\mathbf{Y} - \mathbf{V}) + (\mathbf{X} - \mathbf{U}) \times \langle \mathbf{V} \rangle_b + \langle \mathbf{U} \rangle_b \times (\mathbf{Y} - \mathbf{V}) + \langle \mathbf{UV} \rangle_b$ locally. This requires an online communication of $(m_1 m_2 + m_2 m_3) \cdot n$ bits in in 1 round.

Secure FC ***and*** Conv. A fully-connected layer in neural network is exactly a matrix multiplication, thus secure fully-connected layer (FC) can be implemented directly using Π_{MatMul}. Likewise, convolutional layer can also be expressed as a (larger) matrix multiplication using an unrolling technique (see Figure 3. in [23]), so secure convolution layer (Conv) can also be implemented using Π_{MatMul}.

3.2.2 ***Secure Hadamard Product.*** Neural network training makes extensive use of Hadamard product in backpropagation. Observe that Hadamard product operations (denoted by $\odot$) have a specific structure that can be leveraged to reduce communication costs: when computing $\mathbf{X} \odot \mathbf{Y}$ where $\mathbf{X} \in \mathbb{Z}_{2^n}^{m_1 \times m_2}, \mathbf{Y} \in \mathbb{Z}_2^{m_2 \times m_3}$, the each element x in $\mathbf{X}$ is a n-bit integer and the each element y in $\mathbf{Y}$ is a bit. However, the arithmetic share $\langle x \rangle^A$ can not be directly multiplied by the Boolean share $\langle y \rangle^B$ since they are calculated with different moduli. Existing works first convert $\langle y \rangle^B$ to the arithmetic share $\langle y \rangle^A$ and then perform the multiplication of $\langle x \rangle^A$ and $\langle y \rangle^A$, incurring an online communication of $2m_1 m_2 n$ bits in 2 rounds.

In order to reduce online communication costs, we propose an online-efficient Hadamard product protocol to support the direct computation of $\mathbf{X} \odot \mathbf{Y}$ by moving the share conversion into the offline phase, which requires an online communication of $m_1 m_2 (n+1)$ bits in 1 round. We present a scalar protocol, Π_{BitXA}, to support the product of $\langle x \rangle^A$ and $\langle y \rangle^B$, which can be easily extended to the vector protocol $\Pi_{\mathsf{HadamProd}}$ through the vectorization technique.

Given the arithmetic share $\langle x \rangle^A$ and Boolean share $\langle y \rangle^B$, functionality $\mathcal{F}_{\mathsf{BitXA}}$ generates $\langle z \rangle^A$ with $z = x \cdot y$ and the protocol is shown in Algorithm 1. Π_{BitXA} needs an online communication of $n+1$ bits per party in 1 round.

Compare with Orca[17]. Orca[17] proposes a protocol Π_n^{select} to implement the same functionality and claims that their protocol requires no communication. However, their protocol hides the process of opening secret shares (step 1 in the online phase in Algorithm 1), so it still needs the online communication of $n+1$ bits in 1 round, and the protocol's keysize is $4n$ bits, while our protocol's "keysize" (i.e., $\langle \delta_x \rangle^A, \langle \delta_y \rangle^A, \langle \delta_z \rangle^A$ and $\langle \hat{\delta}_y \rangle^B$) is $3n+1$ bits.

Algorithm 1 BitXA: $\Pi_{\mathsf{BitXA}}(\langle y \rangle^B, \langle x \rangle^A)$

Input: P_0 and P_1 hold $\langle x \rangle^A$ and $\langle y \rangle^B$.
Output: P_0 gets $\langle z \rangle_0^A$ and P_1 gets $\langle z \rangle_1^A$ where $z = x \cdot y$.

• Offline Phase

1: P_b samples $\langle \hat{\delta}_y \rangle_b^B \in_R \mathbb{Z}_2$, and then P_0 and P_1 convert it to arithmetic share:
   (1) P_0 lets $\langle e \rangle_0^A = \langle \hat{\delta}_y \rangle_0^B, \langle f \rangle_0^A = 0$, and P_1 lets $\langle e \rangle_1^A = 0, \langle f \rangle_1^A = \langle \hat{\delta}_y \rangle_1^B$
   (2) P_0 and P_1 compute $(\langle ef \rangle_0^A, \langle ef \rangle_1^A) \leftarrow \Pi_{\mathsf{Mul}}(\langle e \rangle^A, \langle f \rangle^A)$.
   (3) P_b computes $\langle \delta_y \rangle_b^A = \langle e \rangle_b^A + \langle f \rangle_b^A - 2 \cdot \langle ef \rangle_b^A$ locally.
2: P_b samples $\langle \delta_x \rangle_b^A \in_R \mathbb{Z}_{2^n}$, and then parties compute $(\langle \delta_z \rangle_0^A, \langle \delta_z \rangle_1^A) \leftarrow \Pi_{\mathsf{Mul}}(\langle \delta_x \rangle^A, \langle \delta_y \rangle^A)$ s.t. $\delta_z = \delta_x \cdot \delta_y$.
3: P_0 and P_1 hold $\langle \delta_x \rangle^A, \langle \delta_y \rangle^A, \langle \delta_z \rangle^A$ and $\langle \hat{\delta}_y \rangle^B$.

• Online Phase

1: P_b locally computes $\langle x \rangle_b^A + \langle \delta_x \rangle_b^A, \langle y \rangle_b^B \oplus \langle \hat{\delta}_y \rangle_b^B$, and sends to P_{1-b}.
2: P_b reconstructs $\Delta_x = x + \delta_x, \Delta_y = y \oplus \hat{\delta}_y$, and sets $\Delta'_y = \Delta_y$ where $\Delta'_y \in \mathbb{Z}_{2^n}$.
3: P_b computes locally $\langle z \rangle_b^A = b \cdot \Delta'_y \cdot \Delta_x + \langle \delta_y \rangle_b^A \cdot \Delta_x - 2 \cdot \Delta'_y \cdot \Delta_x \cdot \langle \delta_y \rangle_b^A - \Delta'_y \cdot \langle \delta_x \rangle_b^A - \langle \delta_z \rangle_b^A + 2 \cdot \Delta'_y \cdot \langle \delta_z \rangle_b^A$.

Security Analysis. Theorem 3.1 captures the security of Π_{BitXA}, and the full proof is given in Appendix B.1. The security of HadamProd follows in the $\mathcal{F}_{\mathsf{BitXA}}$-hybrid model.

Theorem 3.1. *In the $\mathcal{F}_{\mathsf{Mul}}$-hybrid model, Π_{BitXA} securely computes the functionality $\mathcal{F}_{\mathsf{BitXA}}$ in the presence of semi-honest adversaries.*

3.3 Construction of Secure Non-Linear Layers

In this section, we present the construction of non-linear layer functions (i.e., DReLU and ReLU, § 3.3.1) by using a key-reduced distributed comparison function (DCF) scheme with compact additive construction (§ 3.3.2). To replace the trusted dealer in the offline phase, we propose a DCF key generation scheme based on MPC-friendly pseudorandom generators in § 3.3.3, which supports a larger input domain. In this paper, we directly use the secure maxpool algorithm proposed in [23] (see algorithm 7 in [23]) and its derivative, but utilize our proposed DCF construction.

3.3.1 ***Secure DReLU and ReLU.*** ReLU is one of the most popular activation functions in neural network training. For a signed value x, ReLU is define as $\max(0, x)$ and its derivative is defined as $\mathbf{1}\{x \geq 0\}$. Given an arithmetic share $\langle x \rangle$, the functionality $\mathcal{F}_{\mathsf{ReLU}}$ outputs the arithmetic share of $\max(0, x)$, and the functionality $\mathcal{F}_{\mathsf{DReLU}}$ outputs the Boolean share of $\mathbf{1}\{x \geq 0\}$. It can be seen that $\mathsf{ReLU}(x) = x \cdot \mathsf{DReLU}(x)$.

In this section, by leveraging the DDCF scheme constructed by using the proposed DCF (§ 3.3.2), we first design a signed integer comparison gate scheme to implement DReLU, and then compute ReLU using $\Pi_{\mathsf{ReLU}}(\langle x\rangle^A) = \Pi_{\mathsf{BitXA}}(\langle x\rangle^A, \langle \Pi_{\mathsf{DReLU}}(\langle x\rangle^A)\rangle^B)$.

Secure DReLU. To implement $\mathcal{F}_{\mathsf{DReLU}}$, we first propose signed integer comparison gate Comp in Algorithm 2 which is derived from [2]. In the Algorithm 2, $(\mathsf{Gen}_{n-1}^{\mathsf{DDCF}}, \mathsf{Eval}_{n-1}^{\mathsf{DDCF}})$ is used to evaluate $f^{<}_{\alpha^{(n-1)},\beta_0,\beta_1}(x)$ that outputs β_0 for $0 \le x < \alpha^{(n-1)}$ and β_1 for $x \ge \alpha^{(n-1)}$, and its detailed construction is shown in the Appendix A. Comp requires 1 call of DDCF and the key sizes are $(\lceil n-1-\log\lambda\rceil)(\lambda+3)+2\lambda$ bits per party where λ is the security parameter.

Base on Comp, Π_{DReLU} is proposed to compute $\mathbf{1}\{x \ge 0\}$, and this protocol is in Algorithm 3 where a trusted dealer is used to precompute FSS keys of Comp. The trusted dealer can be instantiated via using our proposed distributed DCF key generation scheme (Algorithm 5 in § 3.3.2). Π_{DReLU} requires 1 call of Comp in the offline phase, and requires requires 1 round with n bits in the online phase.

Algorithm 2 Signed Integer Comparison Gate Comp : $(\mathsf{Gen}_n^{\mathsf{Comp}}, \mathsf{Eval}_n^{\mathsf{Comp}})$

- $\mathsf{Gen}_n^{\mathsf{Comp}}(1^\lambda, \mathsf{r}_1^{\mathsf{in}}, \mathsf{r}_2^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$
  1: Let $r = (2^n - (\mathsf{r}_1^{\mathsf{in}} - \mathsf{r}_2^{\mathsf{in}})) \in \mathbb{Z}_{2^n}$, and $\alpha^{(n-1)} = r_{[0,n-1)}$.
  2: $(k_0^{(n-1)}, k_1^{(n-1)}) \leftarrow \mathsf{Gen}_{n-1}^{\mathsf{DDCF}}(1^\lambda, \alpha^{(n-1)}, \beta_0, \beta_1)$, where $\beta_0 = 1 \oplus r_{[n-1]}, \beta_1 = r_{[n-1]}$.
  3: Sample randoms $\mathsf{r}_0, \mathsf{r}_1 \in_R \mathbb{Z}_2$, s.t. $\mathsf{r}_0 \oplus \mathsf{r}_1 = \mathsf{r}^{\mathsf{out}}$.
  4: Let $k_b = k_b^{(n-1)} || \mathsf{r}_b$
  5: **return** (k_0, k_1).
- $\mathsf{Eval}_n^{\mathsf{Comp}}(b, k_b, \hat{x}, \hat{y})$
  1: P_b parses $k_b = k_b^{(n-1)} || \mathsf{r}_b$, and lets $z = (\hat{x} - \hat{y}) \in \mathbb{Z}_{2^n}$.
  2: P_b computes $m_b \leftarrow \mathsf{Eval}_{n-1}^{\mathsf{DDCF}}(b, k_b^{(n-1)}, z^{(n-1)})$, where $z^{(n-1)} = 2^{n-1} - z_{[0,n-1)} - 1$.
  3: P_b lets $v_b = (b \cdot z_{[n-1]}) \oplus m_b \oplus \mathsf{r}_b$.
  4: **return** v_b.

Algorithm 3 DReLU : $\Pi_{\mathsf{DReLU}}(\langle x\rangle^A)$

Input: P_0 and P_1 hold arithmetic share $\langle x\rangle^A$.
Output: P_0 and P_1 obtain Boolean share $\langle y\rangle^B$, where $\langle y\rangle_0^B \oplus \langle y\rangle_1^B = \mathbf{1}\{x \ge 0\}$.

- **Offline Phase**
  1: Dealer computes $(k_0, k_1) \leftarrow \mathsf{Gen}_n^{\mathsf{Comp}}(1^\lambda, \mathsf{r}_1^{\mathsf{in}}, \mathsf{r}_2^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$.
  2: Dealer sends k_b, $\langle \mathsf{r}_1^{\mathsf{in}}\rangle_b^A$,$\langle \mathsf{r}^{\mathsf{out}}\rangle_b^B$ and $\mathsf{r}_2^{\mathsf{in}}$ to P_b.
- **Online Phase**
  1: P_b computes $\langle x + \mathsf{r}_1^{\mathsf{in}}\rangle_b^A = \langle x\rangle_b^A + \langle \mathsf{r}_1^{\mathsf{in}}\rangle_b^A$ and sends $\langle x + \mathsf{r}_1^{\mathsf{in}}\rangle_b^A$ to P_{1-b}.
  2: P_b computes $x + \mathsf{r}_1^{\mathsf{in}} = \langle x + \mathsf{r}_1^{\mathsf{in}}\rangle_0^A + \langle x + \mathsf{r}_1^{\mathsf{in}}\rangle_1^A$.
  3: P_b computes $\langle y\rangle_b^B \leftarrow b \oplus \mathsf{Eval}_n^{\mathsf{Comp}}(b, k_b, x + \mathsf{r}_1^{\mathsf{in}}, \mathsf{r}_2^{\mathsf{in}}) \oplus \langle \mathsf{r}^{\mathsf{out}}\rangle_b^B$ locally.
  4: **return** $\langle y\rangle_b^B$.

Secure ReLU. $\mathcal{F}_{\mathsf{ReLU}}$ is implemented by computing $\Pi_{\mathsf{ReLU}}(\langle x\rangle^A) = \Pi_{\mathsf{BitXA}}(\langle x\rangle^A, \langle \Pi_{\mathsf{DReLU}}(\langle x\rangle^A)\rangle^B)$, which needs the same key sizes as Π_{DReLU} and requires 1 round with $2n+1$ bits in the online phase.

Security Analysis. Theorem 3.2 captures the security of Π_{DReLU}, and the full proof is given in Appendix B.2. The security of Π_{ReLU} follows in $(\mathcal{F}_{\mathsf{BitXA}}, \mathcal{F}_{\mathsf{DReLU}})$-hybrid model.

Theorem 3.2. *In the $\mathcal{F}_{\mathsf{Gen}_n^{\mathsf{Comp}}}$-hybrid model, Π_{DReLU} securely computes the functionality $\mathcal{F}_{\mathsf{DReLU}}$ in the presence of semi-honest adversaries.*

3.3.2 *Key-reduced Distributed Comparison Function with Compact Additive Construction*.

A central building block in FssNN is a distributed comparison function (DCF), which is intensively used in neural network to build activation functions like ReLU (and its derivative). We examine the case of $x, \alpha \in \mathbb{Z}_{2^n}$ and $\beta \in \mathbb{Z}_2$ and propose a key-reduced DCF scheme with compact additive construction for comparison function $f^{<}_{\alpha,\beta}(x)$. The proposed DCF construction has the following two differences compared the state-of-the-art work [2]: (1) we maintain input group $\mathbb{G}^{\mathsf{in}} = \mathbb{Z}_{2^n}$ but let the output group $\mathbb{G}^{\mathsf{out}} = \mathbb{Z}_2$ rather than $\mathbb{G}^{\mathsf{out}} = \mathbb{Z}_{2^m}$ where m, n are integers, and propose a key-reduced DCF construction by integrating correction words and designing a more compact key generation algorithm, (2) we apply the idea of early termination in [4] to reduce the number of actually required correction words, thereby further reducing the DCF key sizes. Therefore, our proposed DCF construction only supports the output group $\mathbb{G}^{\mathsf{out}} = \mathbb{Z}_2$, but it has smaller key sizes than [2] from $n(\lambda+3)+\lambda+1$ bits to $(\lceil n-\log\lambda\rceil)(\lambda+3)+2\lambda$ bits where λ is the security parameter.

Intuition. our construction draws inspiration from the distributed point function of [4], which involves the algorithm $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$. **In algorithm** $\mathsf{Gen}_n^<$, the pseudorandom generator (PRG) G is used to generate two DCF keys (k_0, k_1) such that $\forall b \in \{0,1\}$, k_b includes an initial random PRG seed $s_b^{(0)}$ and n shared correction words $(CW^{(1)}, \cdots, CW^{(n)})$. The key k_b implicitly defines a Goldreich-Goldwasser-Micali(GGM)-style binary tree [14] with 2^n leaves, where the leaves are labeled by input 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 a PRG seed, v represents a resulting bit, and t represents a state bit. The label of each node is fully determined by the label of its parent node. We let the resulting bit v record the result of $x < \alpha$ to ensure that the "sum" $v_0 \oplus v_1$ over all nodes leading to input x is exactly equal to $f^{<}_{\alpha,\beta}(x)$. In addition, we take advantage of the correlation between these tuples (s, v, t) and only store the independent PRG seed, resulting bit and state bit in correction words, thereby reducing the sizes of DCF key. **In algorithm** $\mathsf{Eval}_n^<$, P_b evaluates key k_b on an input x where P_b traverses the tree generated by k_b from the root to the leaf node representing x and computes (s_b, v_b, t_b) at each node, and finally sums up the resulting bit v_b.

Next, a comprehensive explanation of $\mathsf{Gen}_n^<$ and $\mathsf{Eval}_n^<$ is provided by detailing the *key generation phase* and the *evaluation phase.*

Key Generation Phase. Specifically, the algorithm $\mathsf{Gen}_n^<$ generates the secret share of $f^{<}_{\alpha,\beta}(x)$ (i.e., k_0 and k_1) by generating

distributed GGM-style binary trees. The two GGM-style trees generated by $\mathsf{Gen}_n^<$ 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 state 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 state bits are the same, and the "sum" of two resulting bits equals 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. In addition, we can easily meet the invariant for the root (which is always on the special evaluation path) by simply including the labels in the key. 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 share 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 state bit t. Suppose that the labels of 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 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 word 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 word $CW^{(i)} = (s^L, v^L \oplus \beta, t^L \oplus 1; s^L, v^R, t^R \oplus 1)$. Indeed, this ensures that after the correction word 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 words $CW^{(i)}$ 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 3 illustrates a construction example of $\mathsf{Gen}_n^<$ when $n = 2$, with the 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 partie are each given a key which includes a distinct initial seed $s^{(0)}$ and n correction words $(CW^{(1)}, \cdots, CW^{(n)})$. Each party 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 the 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.

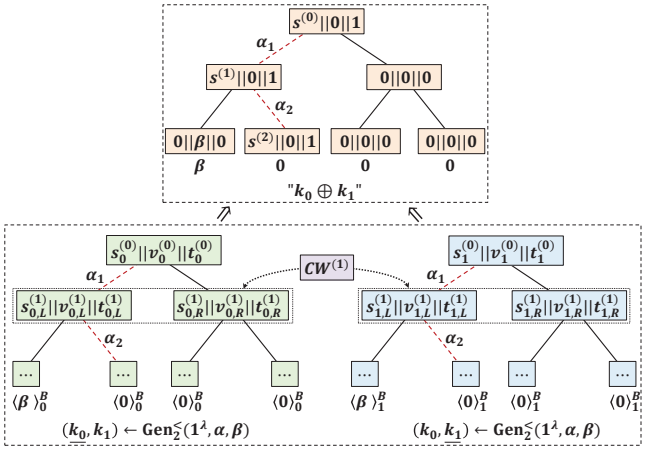

Figure 3: A construction example of $\mathsf{Gen}_n^<$ when $n = 2$

Distributed Comparison Function. $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$ are presented by Algorithm 4, where $G : \{0,1\}^\lambda \to \{0,1\}^{2(\lambda+2)}$ be a PRG, and $||$ is a concatenation operator. In Algorithm 4, the number of PRG invocations in $\mathsf{Gen}_n^<$ is $2n$ and the number of PRG invocations in $\mathsf{Eval}_n^<$ is n.

Early Termination Optimization. According to the description of the early termination technique in Boyle's distributed point function scheme [4], if the length of elements in the output group of a point function is short than the length of random string generated for each node, then several outputs can be packed into a single correction word. We can further improve the complexity of $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$ by using the "early termination" optimization, which works as follows: for any node V of depth v in the tree, there are 2^{n-v} leaf nodes in its sub-tree, or 2^{n-v} input elements with a shared prefix that ends at V. If the size of $CW^{(v+1)}$ is at least 2^{n-v} times the output length then the subsequent correction words, especially v_{CW}^L, can be computed and packed into a single $CW^{(v+1)}$ instead of involving all subsequent correction words. In this case, $CW^{(v+1)}$ will be a sequence of $v_{CW}^{L,\hat{\alpha}} = v_{CW}^{L,v+1} \oplus \cdots \oplus v_{CW}^{L,n}$ where $\hat{\alpha} \in \mathbb{Z}_{2^{n-v}}$ and $v_{CW}^{L,i}$ is the last $n-v$ values in all v_{CW}^Ls (i.e., $v_{CW}^{L,i}$ for $i = v+1, \cdots, n$) where v_{CW}^L is defined in the line 11 in Algorithm 4. The sequence will output β in the location specified by $\alpha_{[v+1,n]} = \alpha_{v+1}||\cdots||\alpha_n$, and 0 in every other location.

Algorithm 4 DCF: $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$

- $\mathsf{Gen}_n^<(1^\lambda, \alpha, \beta)$
  1: Let $\alpha_1||\cdots||\alpha_n \in \{0,1\}^n$ be the bit decomposition of α.
  2: Sample randoms $s_0^{(0)} \in_R \{0,1\}^\lambda$ and $s_1^{(0)} \in_R \{0,1\}^\lambda$, $v_0^{(0)} = 0, v_1^{(0)} = 0$, 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: $s_0^{(i)} = s_0^{\mathsf{Keep}} \oplus t_0^{(i-1)} \cdot s_{CW}, s_1^{(i)} = s_1^{\mathsf{Keep}} \oplus t_1^{(i-1)} \cdot s_{CW}$.
  15: $t_0^{(i)} = t_0^{\mathsf{Keep}} \oplus t_0^{(i-1)} \cdot t_{CW}^{\mathsf{Keep}}, t_1^{(i)} = t_1^{\mathsf{Keep}} \oplus t_1^{(i-1)} \cdot t_{CW}^{\mathsf{Keep}}$.
  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 $x = x_1||\cdots||x_n$, $v = 0$, 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: Compute $G(s^{(i-1)}) = \hat{s}^L||\hat{v}^L||\hat{t}^L||\hat{s}^R||\hat{v}^R||\hat{t}^R$.
  5: $\tau^{(i)} \leftarrow [\hat{s}^L||\hat{v}^L||\hat{t}^L||\hat{s}^R||\hat{v}^R||\hat{t}^R] \oplus (t^{(i-1)} \cdot [s_{CW}||v_{CW}^L||t_{CW}^L||s_{CW}||v_{CW}^L||t_{CW}^R])$.
  6: Parse $\tau^{(i)} = s^L||v^L||t^L||s^R||v^R||t^R$.
  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

In this paper, we let $v = \lceil n - \log\lambda \rceil$ to satisfy the above conditions. Therefore, there are only v correction words of size $\lambda + 3$, plus a $CW^{(v+1)}$ of size $2^{n-v} = \lambda$, thus the total DCF key sizes are $(\lceil n - \log\lambda \rceil)(\lambda + 3) + 2\lambda$ bits.

Security Analysis. Theorem 3.3 captures the correctness and security of the DCF construction, and its full proof can be found in Appendix B.3.

Theorem 3.3. *(Correctness and Security) The scheme* $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$ *from Algorithm 4 is a DCF for the family of comparison functions* $f_{\alpha,\beta}^<(x) : \mathbb{Z}_{2^n} \to \mathbb{Z}_2$ *with key sizes* $(\lceil n - \log\lambda \rceil)(\lambda + 3) + 2\lambda$ *bits, where* λ *is the security parameter.*

3.3.3 ***Distributed DCF Key Generation.*** A trusted dealer is required to execute the procedure $\mathsf{Gen}_n^{\mathsf{Comp}}$ for Comp in Algorithm 3, and since Comp is constructed based on the proposed DCF, the DCF key needs to be computed indeed. To instantiate the trusted dealer, the work [2] gives a secure two-party generation scheme of DCF key by extending the Doerner-Shelat [10] protocol, but the scheme is restricted to a small domain size (e.g., $\mathbb{Z}_{2^{16}}$ or smaller) since computation costs grow exponentially with the domain size. To support a larger domain size which is adopted in many practical scenarios, we propose a distributed DCF key generation scheme base on MPC-friendly pseudorandom generators (PRG) [9] where two parties jointly emulate the role of the trusted dealer via using a two-party computation protocol.

Distributed DCF Key Generation Scheme based on MPC-friendly PRG. To realize the functionality $\mathcal{F}_{\mathsf{Gen}^{\mathsf{DCF}}}$, we present $\Pi_{\mathsf{Gen}^{\mathsf{DCF}}}$ based on a secure two-party PRG to generate DCF key, and this protocol is in Algorithm 5, where $\mathcal{F}_{\mathsf{SecPRG}}$ can be realized by the MPC-friendly PRG[9] and $\mathcal{F}_{\mathsf{2PC}}$ can be instantiate via using a secure two-party protocol based on secret sharing [7, 8, 22]. Obliviously, Algorithm 5 is naturally extended to the case of using the early termination optimization.

The scheme in [2] (Fig. 9 in [2]) needs $O(2^n)$ invocations of PRG and is restricted to moderate domain sizes. By comparison, $\Pi_{\mathsf{Gen}^{\mathsf{DCF}}}$ only requires $O(n)$ invocations of PRG and can be used with larger domain sizes. Although the Appendix A.2 in [2] also mentions a distributed DCF key generation scheme via a generic 2PC with $O(n)$ evaluations of PRG, it does not give a specific construction. More importantly, the scheme mentioned in [2] is only applicable to the DCF construction proposed in [2] and cannot be used to generate the key of our proposed DCF construction, because our proposed DCF construction is essentially different from the DCF construction of [2].

4 THEORETICAL ANALYSIS AND EXPERIMENT

In the section, we present the theoretical analysis of the communication and computation complexity in § 4.1, and show the experiment results in § 4.2.

4.1 Theoretical Analysis

Online Round and Communication Complexity. The online rounds and communication costs of each neural network operation in ABY2.0 [22], AriaNN [23] and FssNN are presented in Table 2. The function $\mathsf{MatMul}_{m_1,m_2,m_3}$ denotes a matrix multiplication of dimension $m_1 \times m_2$ with $m_2 \times m_3$, and $\mathsf{HadamProd}_{m_1,m_2}$ denotes a Hadamard product of dimension $m_1 \times m_2$. ReLU_{m_1,m_2} and DReLU_{m_1,m_2} denotes ReLU and its derivative over a $m_1 \times m_2$ matrix, and $\mathsf{MaxPool}_{m,k,s}$ denotes maxpool with input the $m \times m$ where k stands for the kernel size and s stands for the stride. All communication is measured for n-bit inputs and missing entries mean that data was not available.

For round complexity, all neural network operators are computed with constant online communication rounds in AriaNN and FssNN, while linear functions (i.e., MatMul and HadamProd) computation requires 1 round and non-linear (i.e., DReLU, ReLU and

Algorithm 5 $\Pi_{\mathrm{Gen}^{\mathrm{DCF}}}(1^\lambda, b, \{\langle\alpha_i\rangle_b^B\}_{i=1,\cdots,n}, \langle\beta\rangle_b^B)$

Input: Party index b, and P_b holds $\{\langle\alpha_i\rangle_b^B\}_{i=1,\cdots,n}$ and $\langle\beta\rangle_b^B$.
Output: P_b gets DCF key k_b.

1: P_b samples randoms $s_b^{(0)} \in_R \{0,1\}^\lambda, t_b^{(0)} = b$.
2: P_b invokes $\Pi_{\mathrm{Share}}(s_b^{(0)}), \Pi_{\mathrm{Share}}(t_b^{(0)})$ to generate secret shares of $s_b^{(0)}$ and $t_b^{(0)}$, then P_b obtains $\langle s_0^{(0)}\rangle_b^B, \langle t_0^{(0)}\rangle_b^B, \langle s_1^{(0)}\rangle_b^B, \langle t_1^{(0)}\rangle_b^B$.
3: **for** $i = 1$ to n **do**
4: P_0 and P_1 engage in a secure two-party PRG to compute (for $j \in \{0,1\}$):

$$(\langle G(s_j^{(i-1)})\rangle_0^B, \langle G(s_j^{(i-1)})\rangle_1^B) \leftarrow \mathcal{F}_{\mathrm{SecPRG}}(\langle s_j^{(i-1)}\rangle_0^B, \langle s_j^{(i-1)}\rangle_1^B)$$

where $\langle G(s_j^{(i-1)})\rangle_0^B \oplus \langle G(s_j^{(i-1)})\rangle_1^B = s_j^{L,i-1}||v_j^{L,i-1}||t_j^{L,i-1}||s_j^{L,i-1}||v_j^{L,i-1}||t_j^{L,i-1}$.

5: P_0 and P_1 make access to $\mathcal{F}_{\mathrm{2PC}}$ 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: P_b computes $CW^{(i)} = s_{CW}||v_{CW}^L||t_{CW}^L||t_{CW}^R$ locally.
7: P_0 and P_1 make access to $\mathcal{F}_{\mathrm{2PC}}$ to compute:

$$s_b^{(i)} \leftarrow \begin{cases} s_b^{L,i-1} \oplus t_b^{(i-1)} \cdot s_{CW} & \alpha_i = 0 \\ s_b^{R,i-1} \oplus t_b^{(i-1)} \cdot s_{CW} & \alpha_i = 1 \end{cases}$$

$$t_b^{(i)} \leftarrow \begin{cases} t_b^{L,i-1} \oplus t_b^{(i-1)} \cdot t_{CW}^L & \alpha_i = 0 \\ t_b^{R,i-1} \oplus t_b^{(i-1)} \cdot t_{CW}^R & \alpha_i = 1 \end{cases}$$

P_0 and P_1 obtain $(\langle s_b^{(i)}\rangle_0^B, \langle t_b^{(i)}\rangle_0^B), (\langle s_b^{(i)}\rangle_1^B, \langle t_b^{(i)}\rangle_1^B)$ respectively, where $b = 0, 1$.
8: **end for**
9: P_b lets $k_b \leftarrow s_b^{(0)}||CW^{(1)}||...||CW^{(n)}$.

MaxPool) computation requires $O(\log n)$ rounds in ABY2.0. For online communication costs, FssNN achieves lower online communication costs in HadamProd and ReLU than AriaNN due to our communication efficient protocol Π_{BitXA}.

Computation Complexity. In the online phase, ABY2.0[22], AriaNN[23] and FssNN all have an order of magnitude of online computation complexity since they all adopt the offline-online paradigm. In the offline phase, FssNN uses the same multiplication triples generation scheme as ABY2.0[22] and AriaNN[23]. However, FssNN needs to generate correlated randomness (i.e., DCF key) to compute DReLU, ReLU and MaxPool, while ABY2.0[22] requires smaller correlated randomness and it can be generated more efficiently using 2PC-based offline phase (but leads to $4-5\times$ more rounds and $3-6\times$ more communication of online communication [2]) and AriaNN[23] relies on a trusted dealer to correlated randomness (but leads to stronger assumptions). Therefore, FssNN requires more computation in the offline phase, but has less online communication compared with ABY2.0[22] and does not rely on the trusted leader compared with AriaNN[23].

DCF Key Sizes. The communication criteria of DCF construction is the sizes of key k_b (i.e., the output of $\mathrm{Gen}_n^<$), so the DCF key sizes in BCG+21[2], AriaNN[23] and FssNN are shown in Table 3. It is clear that the DCF key sizes of FssNN is the smallest, which improves the offline communication efficiency of protocols Π_{DReLU}, Π_{ReLU} and Π_{MaxPool}.

Table 3: The DCF key sizes in BCG+21[2], AriaNN[23] and FssNN where $n = 32, \lambda = 127$ is typical parameters.

Para. (bits)	BCG+21	AriaNN	**FssNN**
(n, λ)	$n(\lambda+3)$ $+\lambda+1$	$n(\lambda+2n+4)$ $+\lambda+2n$	$(\lceil n - \log\lambda\rceil)(\lambda+3)$ $+2\lambda$
(32, 127)	4424	6431	3634

4.2 Experiment

In this section, we present the implement of FssNN and the detailed experiment results and analysis. We implement FssNN in Python and run the experiments on Aliyun ESC using ecs.hfr7.xlarge machines with 32 cores and 256 GB of CPU RAM in a LAN setting. In order to simplify comparison with existing works, we follow a setup very close to AriaNN [23] and use same neural network models and datasets. AriaNN [23] is the state-of-the-art seucre neural network training and inference framework based on function secret sharing, and outperform other works such as FALCON[25] and ABY2.0[22].

Evaluations for Secure Layer Functions. First, we present the offline and online communication costs of linear layer functions (i.e., MatMul and HadamProd) and non-linear layer functions (i.e., DReLU, ReLU and MaxPool) in Table 4.

Table 4: Offline and online communication of neural network operators in AriaNN[23] and FssNN where $(784, 128, 10)$**,** $(128, 128)$ **and** $(24, 2, 2)$ **are typical parameters.**

Operators (Input Sizes)	Offline Comm. (MB)		Online Comm. (MB)	
	AriaNN	**FssNN**	AriaNN	**FssNN**
$\mathrm{MatMul}_{m_1,m_2,m_3}$ $(784, 128, 10)$	0.842	0.842	0.775	0.775
$\mathrm{HadamProd}_{m_1,m_2}$ $(128, 128)$	0.381	0.272 $(\downarrow 28.6\%)$	0.251	0.142 $(\downarrow 43.4\%)$
DReLU_{m_1,m_2} $(128, 128)$	14.377	10.314 $(\downarrow 28.3\%)$	0.126	0.126
ReLU_{m_1,m_2} $(128, 128)$	14.758	10.586 $(\downarrow 28.3\%)$	0.377	0.268 $(\downarrow 28.9\%)$
$\mathrm{MaxPool}_{m,k,s}$ $(24, 2, 2)$	0.399	0.292 $(\downarrow 26.8\%)$	0.020	0.020

For linear layer functions, compared with AriaNN, the offline and online communication costs of HadamProd decreases by 28.6%

Table 2: Online round and communication complexity of ABY2.0[22], AriaNN[23] and FssNN

Operators	Rounds			Communication		
	ABY2.0	AriaNN	**FssNN**	ABY2.0	AriaNN	**FssNN**
$\mathsf{MatMul}_{m_1,m_2,m_3}$	1	1	1	m_1m_3n	$(m_1m_2+m_2m_3)n$	$(m_1m_2+m_2m_3)n$
$\mathsf{HadamProd}_{m_1,m_2}$	1	1	1	$2m_1m_2n$	$2m_1m_2n$	$m_1m_2(n+1)$
DReLU_{m_1,m_2}	$1+\log n$	1	1	$\sim 3m_1m_2n$	m_1m_2n	m_1m_2n
ReLU_{m_1,m_2}	$2+\log n$	2	2	$\sim 5m_1m_2n$	$3m_1m_2n$	$m_1m_2(2n+1)$
$\mathsf{MaxPool}_{m,k,s}$	-	3	3	-	$(\frac{m-k}{s}+1)^2(k^4+2)n$	$(\frac{m-k}{s}+1)^2(k^4+2)n$

and 43.4% respectively. For non-linear layer functions, we improve the offline communication costs by 26.8% − 28.3% over AriaNN due to the proposed key-reduced DCF in FssNN. The online communication improvement of ReLU is attributed to our communication efficient Π_{BitXA}.

Evaluations for Secure Neural Network. We benchmark secure training and inference on MNIST (60,000 training samples and 10,000 test samples) and evaluate following 3 neural networks : (1) a 3-layer fully-connected network (FCNN), (2) a 4-layer convolutional neural network (CNN) and (3) a 4-layer LeNet network (LeNet). It should be noted that FssNN, like AriaNN[23], can support to more machine learning tasks and datasets, such as models AlexNet and VGG16 in datasets CIFAR10 and Tiny Imagenet.

Time, communication and accuracy of secure training and inference in the LAN setting are presented in Table 5, where accuracy of plaintext training and inference are also reported for comparison. The time and communication for secure training are given in hours and GB per epoch, and secure inference is evaluated over pre-trained neural network models and the total time and communication are reported.

Table 5: Time, communication and accuracy of secure training and inference in FssNN (batchsize = 128). The time is reported in hours and the "comm." is reported in GB.

FssNN	Model	Epochs	Time	Comm.	Private Accuracy	Plaintext Accuracy
Training	FCNN	15	0.23	27.35	98.00%	98.04%
	CNN	10	2.24	439.78	98.60%	98.73%
	LeNet	10	3.46	648.83	98.93%	99.03%
Inference	FCNN	-	0.01	2.14	98.15%	98.17%
	CNN	-	0.30	72.84	98.95%	99.02%
	LeNet	-	0.56	107.63	99.22%	99.27%

It is observed that accuracy of secure training and inference are a little lower than their plaintext counterparts, but the gap between them isn't significant.

Compare with AriaNN[23]. The total communication and time for secure training and inference are presented in Table 6. Compared with AriaNN[23], the communication for training declined by 24.3% − 25.4% and the communication for inference decreased by 22.9% − 26.4%. This is attributed to our online-efficient protocol Π_{BitXA}, and key-reduced DCF scheme which improves the communication efficiency of protocols Π_{DReLU}, Π_{ReLU} and Π_{MaxPool}.

Table 6: The communication and time for secure training and inference in AiraNN[23] and FssNN (batchsize = 128)

	Model	Training		Inference	
		Comm. (GB)	Time (h)	Comm. (GB)	Time (h)
AriaNN	FCNN	36.11	0.28	2.84	0.02
	CNN	589.91	2.24	94.49	0.37
	LeNet	869.75	3.50	146.24	0.56
FssNN	FCNN	27.35 (↓ 24.3%)	0.23	2.14 (↓ 24.8%)	0.02
	CNN	439.78 (↓ 25.4%)	2.24	72.84 (↓ 22.9%)	0.30
	LeNet	648.83 (↓ 25.4%)	3.46	107.63 (↓ 26.4%)	0.56

5 CONCLUSION

Privacy-preserving neural network based on secure multiparty computation has emerged as a flourishing research area in the past few years, but its practicality is greatly limited due to the low efficiency caused by massive communication costs and a deep dependence on a trusted dealer. In this paper, we proposed a communication-efficient secure neural network framework, FssNN, to enable practical training and inference. By designing a key-reduced distributed comparison function with compact additive construction and leveraging additive secret sharing and function secret sharing, we reduce offline and online communication costs, and then replace the trusted dealer in the offline phase by designing a distributed key generation scheme. Experiment shows the practical performance of our proposed FssNN, as well as the substantial performance advantage over existing works. Compared with the state-of-the-art solution AriaNN, the communication costs of secure training and inference are decreased by approximately 25.4% and 26.4% respectively. More attempts might be made to construct actively secure protocols to defend against a malicious adversary.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China (62272131), Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1212010005) and CCF-Ant Group Privacy Computing Special Fund (CCF-AFSG RF20220015).

REFERENCES

[1] Donald Beaver. 1992. Efficient Multiparty Protocols using Circuit Randomization. In *Advances in Cryptology - CRYPTO 1992*. Springer, 420–432.

[2] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2021. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *Advances in Cryptology - EUROCRYPT 2021*. Springer, 871–900.

[3] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *Advances in Cryptology - EUROCRYPT 2015*. Springer, 337–367.

[4] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1292–1303.

[5] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation with Preprocessing via Function Secret Sharing. In *17th Theory of Cryptography Conference*. Springer, 341–371.

[6] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *27th Network and Distributed System Security Symposium*. The Internet Society.

[7] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology - CRYPTO 2012*. Springer, 643–662.

[8] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *22nd Network and Distributed System Security Symposium*. The Internet Society.

[9] Itai Dinur, Steven Goldfeder, Tzipora Halevi, Yuval Ishai, Mahimna Kelkar, Vivek Sharma, and Greg Zaverucha. 2021. MPC-Friendly Symmetric Cryptography from Alternating Moduli: Candidates, Protocols, and Applications. In *Advances in Cryptology - CRYPTO 2021*. Springer, 517–547.

[10] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 523–535.

[11] Dengguo Feng and Kang Yang. 2022. Concretely efficient secure multi-party computation protocols: survey and more. *Security and Safety* 1 (2022), 2021001.

[12] Craig Gentry. 2009. Fully Homomorphic Encryption using Ideal Lattices. In *Proceedings of the 41st annual ACM Symposium on Theory of Computing*. ACM, 169–178.

[13] Oded Goldreich. 2004. *Foundations of Cryptography: volume 2, Basic Applications*. Cambridge university press.

[14] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to Construct Random Functions. *J. ACM* (1986), 792–807.

[15] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game. In *Proceedings of the 19th ACM Symposium on Theory of Computing*. ACM, 218–229.

[16] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. 2022. LLAMA: A Low Latency Math Library for Secure Inference. *Proceedings on Privacy Enhancing Technologies* 4 (2022), 274–294.

[17] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2023. Orca: FSS-based Secure Training with GPUs. *Cryptology ePrint Archive* (2023).

[18] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *29th Network and Distributed System Security Symposium*. The Internet Society.

[19] Yehuda Lindell. 2017. How to Simulate it–A Tutorial on the Simulation Proof Technique. *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich* (2017), 277–346.

[20] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. ACM, 35–52.

[21] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE symposium on security and privacy*. IEEE, 19–38.

[22] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2. 0: Improved Mixed-Protocol Secure Two-Party Computation. In *30th USENIX Security Symposium*. 2165–2182.

[23] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. 2022. AriaNN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing. *Proceedings on Privacy Enhancing Technologies* 1 (2022), 291–316.

[24] Kyle Storrier, Adithya Vadapalli, Allan Lyons, and Ryan Henry. 2023. Grotto: Screaming Fast (2+1)-PC for $\mathbb{Z}_{2^n}$ via (2,2)-DPFs. *Cryptology ePrint Archive* (2023).

[25] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proceedings on Privacy Enhancing Technologies* 2021, 1 (2021), 188–208.

[26] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. 2020. Ferret: Fast extension for correlated OT with small communication. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1607–1626.

[27] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, 162–167.

A DUAL DISTRIBUTED COMPARISON FUNCTION

Dual distributed comparison function (DDCF) is a variant of DCF, is defined as:

$$f^{<}_{\alpha,\beta_0,\beta_1}(x) = \begin{cases} \beta_0 & x < \alpha \\ \beta_1 & \text{else} \end{cases} \tag{1}$$

where $x, \alpha \in \mathbb{Z}_{2^n}$, $\beta_0, \beta_1 \in \mathbb{Z}_2$, and $\beta_0 \neq \beta_1$.

We present a DDCF scheme based on the DCF scheme, its detailed construction is shown in Algorithm 6. Compared with [2], our DDCF construction is slightly modified: the output group is constrained to be $\mathbb{Z}_2$, and more importantly, our DDCF construction relies on the proposed key-reduced DCF (i.e., $(\mathsf{Gen}^{<}_n, \mathsf{Eval}^{<}_n)$, § 3.3.2). Our DDCF construction requires 1 call of DCF, and the key sizes are $(\lceil n - \log\lambda \rceil)(\lambda + 3) + 2\lambda$ bits where λ is the security parameter.

Algorithm 6 DDCF: $(\mathsf{Gen}^{\mathsf{DDCF}}_n, \mathsf{Eval}^{\mathsf{DDCF}}_n)$

- $\mathsf{Gen}^{\mathsf{DDCF}}_n(1^\lambda, \alpha, \beta_0, \beta_1)$
  1: Compute $(k^{(n)}_0, k^{(n)}_1) \leftarrow \mathsf{Gen}^{<}_n(1^\lambda, \alpha, \beta_0 \oplus \beta_1)$.
  2: Sample $r_0, r_1 \in_R \{0,1\}$, s.t. $r_0 \oplus r_1 = \beta_1$.
  3: Let $k_b = k^{(n)}_b || r_b$ for $b \in \{0,1\}$.
  4: **return** (k_0, k_1).
- $\mathsf{Eval}^{\mathsf{DDCF}}_n(b, k_b, x)$
  1: Parse $k_b = k^{(n)}_b || r_b$, and compute $y^{(n-1)}_b \leftarrow \mathsf{Eval}^{<}_n(b, k^{(n)}_b, x)$.
  2: Let $v_b = y^{(n-1)}_b \oplus r_b$.
  3: **return** v_b.

B PROOF

B.1 Proof of Theorem 3.1

Proof. For a corrupted party P_b, we show a probabilistic polynomial-time (PPT) simulator Sim_b who can generate a simulated view that is indistinguishable from P_b's view in real world. In the offline phase, Sim_b first samples $\delta_x \in_R \mathbb{Z}_{2^n}, \hat{\delta}_y \in_R \mathbb{Z}_2, \delta_y \in_R \mathbb{Z}_{2^n}$. Next, Sim_b computes $(\langle\delta_x\rangle^A_0, \langle\delta_x\rangle^A_1) \leftarrow \Pi_{\mathsf{Share}}(\delta_x)$, $(\langle\hat{\delta}_y\rangle^B_0, \langle\hat{\delta}_y\rangle^B_1) \leftarrow \Pi_{\mathsf{Share}}(\hat{\delta}_y)$ and $(\langle\delta_y\rangle^A_0, \langle\delta_y\rangle^A_1) \leftarrow \Pi_{\mathsf{Share}}(\delta_y)$, and sends $\langle\delta_x\rangle^A_b$, $\langle\hat{\delta}_y\rangle^B_b$ and $\langle\delta_y\rangle^A_b$ as P_b's values. Note that $\langle e\rangle^A_b$, $\langle f\rangle^A_b$ and $\langle ef\rangle^A_b$ isn't independent, and can be computed directly using $\langle\hat{\delta}_y\rangle^B_b, \langle\delta_y\rangle^A_b$ when we make only black-box access to $\mathcal{F}_{\mathsf{Mul}}$. Finally, Sim_b computes $\delta_z = \delta_x \cdot \delta_y$ and sends $\langle\delta_z\rangle^A_b$ to P_b by running $(\langle\delta_z\rangle^A_0, \langle\delta_z\rangle^A_1) \leftarrow \Pi_{\mathsf{Share}}(\delta_z)$. In the online phase, Sim_b follows the steps honestly using the data obtained from the offline phase. □

B.2 Proof of Theorem 3.2

Proof. For a corrupted party P_b, we show a PPT simulator Sim_b who can generate a simulated view that is indistinguishable from P_b's view in real world. In offline phase, Sim_b first samples $\mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{in}}_2, \mathsf{r}^{\mathsf{out}}$ and computes $(\langle\mathsf{r}^{\mathsf{in}}_1\rangle^A_0, \langle\mathsf{r}^{\mathsf{in}}_1\rangle^A_1) \leftarrow \Pi_{\mathsf{Share}}(\mathsf{r}^{\mathsf{in}}_1)$ and

$(\langle \mathsf{r}^{\mathsf{out}}\rangle_0^B, \langle \mathsf{r}^{\mathsf{out}}\rangle_1^B) \leftarrow \Pi_{\mathsf{Share}}(\mathsf{r}^{\mathsf{out}})$, and then send $\langle \mathsf{r}_1^{\mathsf{in}}\rangle_b, \mathsf{r}_2^{\mathsf{in}}$ and $\langle \mathsf{r}^{\mathsf{out}}\rangle_b^B$ to P_b. Afterward, Sim_b gets (k_0, k_1) via making black-box access to $\mathcal{F}_{\mathsf{Gen}_n^{\mathsf{Comp}}}$ and send k_b to P_b. In the offline phase, Sim_b follows the steps honestly using the data obtained from the offline phase. □

B.3 Proof of Theorem 3.3

Proof. The correctness and security of $(\mathsf{Gen}_n^<, \mathsf{Eval}_n^<)$ is proved as follows, and the proof of security is very similar to the proof of [2]:

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 $x = x_1x_2\cdots x_n, \alpha = \alpha_1\alpha_2\cdots\alpha_n$, 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 2 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 $CW^{(1)} = s_{CW}||v_{CW}^L||t_{CW}^L||v_{CW}^R||t_{CW}^R$ 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 $v^L = \hat{v}^L \oplus (t^{(0)}\cdot v_{CW}^L) = \hat{v}^L \oplus (b\cdot v_{CW}^L)$ and $v^R = \hat{v}^R \oplus (t^{(0)}\cdot v_{CW}^R) = \hat{v}^R \oplus (b\cdot v_{CW}^R)$ according to line 5 and line 6 of Eval. And according to the definition of $CW^{(1)}$ and $G(s^{(0)})$ in Gen, $v_{CW}^L = v_0^L \oplus v_1^L \oplus (\alpha_i\cdot\beta), v_{CW}^R = v_0^R \oplus v_1^R$ and $\hat{v}^L = v_b^L, \hat{v}^R = v_b^R$, and then $v_b = ((1-x_1)\cdot(b \oplus v_0^L)) \oplus (x_1\cdot v_0^R)$, 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 $s^L = \hat{s}^L \oplus (t^{(i-1)}\cdot s_{CW}), s^R = \hat{s}^R \oplus (t^{(i-1)}\cdot s_{CW}), t^L = \hat{t}^L \oplus (t^{(i-1)}\cdot t_{CW}^L), t^R = \hat{t}^R \oplus (t^{(i-1)}\cdot t_{CW}^R)$ where $s_{CW}, t_{CW}^L, t_{CW}^R$ 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 $\mathsf{Gen}_n^<(1^\lambda, \alpha, \beta)$. Finally, according to lines 14 to 15 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, $\Pr[\mathsf{Eval}(0, k_0, x) \oplus \mathsf{Eval}(1, k_1, x) = f_{\alpha,\beta}^<(x)] = 1$.

Security. 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 security is as follows. Each party $b \in \{0,1\}$ starts with a random seed $s_b^{(0)}$ 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 $s_b^{(i-1)}$ to generate six items: namely, two seeds s_b^L, s_b^R, two resulting bits v_b^L, v_b^R and two control bits t_b^L, t_b^R. 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 i-th level correction word $CW^{(i)}$ will "use up" the secret randomness of 5 of these 6 pieces: the two bits t_b^L, t_b^R, the resulting bits v_b^L, v_b^R and the seed s_b^{Lose} 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_b^{Keep} 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_b^{Keep}.

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

(1) $s_b^{(0)} \leftarrow \{0,1\}^\lambda$ chosen at random (honestly), and let $t_b^{(0)} = b$.
(2) $CW^{(1)}, \cdots, CW^{(j)} \leftarrow \{0,1\}^{\lambda+1}$ chosen at random.
(3) For $i \le j$, $s_b^{(i)}||v_b^{(i)}||t_b^{(i)}$ computed honestly, as a function of $s_b^{(0)}||v_b^{(0)}||t_b^{(0)}$ 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 let $t_{1-b}^{(j)} = 1 - t_b^{(j)}$.
(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 B.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 B.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 B.1 and Lemma B.2 follow directly by construction of Hyb_j.

Lemma B.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 $T' \le T - p'(\lambda)$, it holds that*

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

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

- $\mathsf{Hyb}_j(1^\lambda, b, \alpha, \beta)$
  1: Let $\alpha_1||\cdots||\alpha_n \in \{0,1\}^n$ be the bit decomposition of α.
  2: Sample random $s_b^{(0)} \in_R \{0,1\}^\lambda$ 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 $CW^{(j)} \in_R \{0,1\}^\lambda \times \{0,1\} \times \{0,1\}^2$.
  6: **else**
  7: **if** $i = j$ **then**
  8: Sample random $s_{1-b}^{(j-1)} \in_R \{0,1\}^\lambda$ 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 Keep $\leftarrow L$, Lose $\leftarrow R$
  5: **else**
  6: Set Keep $\leftarrow R$, 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 $\alpha_1||\cdots||\alpha_n \in \{0,1\}^n$ be the bit decomposition of α.
  2: Sample random $s_b^{(0)} \in_R \{0,1\}^\lambda$ 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 $CW^{(i)} \in_R \{0,1\}^\lambda \times \{0,1\} \times \{0,1\}^2$.
  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 Keep $\leftarrow L$, Lose $\leftarrow R$
  9: **else**
  10: Set Keep $\leftarrow R$, 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 Keep $\leftarrow L$, Lose $\leftarrow R$
  20: **else**
  21: Set Keep $\leftarrow R$, 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 Keep $\leftarrow L$, Lose $\leftarrow R$
  6: **else**
  7: Set Keep $\leftarrow R$, 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** $(CW^{(j)}||CW^{(j+1)}||\cdots||CW^{(n)})$

$s \leftarrow \{0,1\}^\lambda$ 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:

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

- $t^L_{CW} = t^L_b \oplus t^L_{1-b} \oplus \alpha_j \oplus 1$.
- $t^R_{CW} = t^R_b \oplus t^R_{1-b} \oplus \alpha_j$.

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

Now, condition on $CW^{(j)}$ as well, and consider the value of $s^{(j)}_{1-b}$. Depending on the value of $t^{(j-1)}_{1-b}$, $s^{(j)}_{1-b}$ is selected either as s^{Keep}_{1-b} or $s^{\mathsf{Lose}}_{1-b} \oplus s_{CW}$. However, s^{Keep}_{1-b} 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 $t^{(j)}_{1-b}$. Note that both $t^{(j)}_b$ and $t^{(j)}_{1-b}$ are computed as per NextST, as a function of $t^{(j-1)}_1$ and $t^{(j-1)}_{1-b}$, respectively (and $t^{(j-1}_{1-b}$ was set to $1 - t^{(j-1)}_b$). In particular,

$$\begin{aligned} t^{(j)}_b \oplus t^{(j)}_{1-b} &= (t^{\mathsf{Keep}}_b \oplus t^{(i-1)}_b \cdot t^{\mathsf{Keep}}_{CW}) \oplus (t^{\mathsf{Keep}}_{1-b} \oplus t^{(i-1)}_{1-b} \cdot t^{\mathsf{Keep}}_{CW}) \\ &= t^{\mathsf{Keep}}_b \oplus t^{\mathsf{Keep}}_{1-b} \oplus (t^{(i-1)}_b \oplus t^{(i-1)}_{1-b}) \cdot t^{\mathsf{Keep}}_{CW} \\ &= t^{\mathsf{Keep}}_b \oplus t^{\mathsf{Keep}}_{1-b} \oplus 1 \cdot (t^{\mathsf{Keep}}_0 \oplus t^{\mathsf{Keep}}_1 \oplus 1) \\ &= 1 \end{aligned}$$

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. □