







# Compact Key Function Secret Sharing with Non-linear Decoder

Chandan Kumar<sup>1</sup>  , Sikhar Patranabis<sup>2</sup>  and  
Debdeep Mukhopadhyay<sup>1</sup> 

<sup>1</sup> IIT Kharagpur, India

<sup>2</sup> IBM Research, India

**Abstract.** We present a variant of Function Secret Sharing (FSS) schemes tailored for point, comparison, and interval functions, featuring compact key sizes at the expense of additional comparison. While existing FSS constructions are primarily geared towards 2-party scenarios, exceptions such as the work by Boyle *et al.* (Eurocrypt 2015) and Riposte (S&P 2015) have introduced FSS schemes for  $p$ -party scenarios ( $p \geq 3$ ). This paper aims to achieve the most compact  $p$ -party FSS key size to date. We achieve a noteworthy reduction in key size, a  $2^p$ -factor decrease compared to state-of-the-art FSS constructions (including computationally efficient constructions using symmetric-key primitives) of distributed point function (DPF). Compared to the previous public-key-based FSS design for DPF, we also get a key size reduction equal to a  $2^{n/2}$ -sized row vector, where  $2^n$  is the domain size of the point function. This reduction in key size comes at the cost of a required comparison operation by the decoder (hence called a non-linear decoder), a departure from prior schemes. In  $p$ -party scenarios, our construction outperforms existing FSS constructions in key size, remaining on par with Riposte in evaluation time and showing significant improvement over Boyle *et al.*

In addition to constructing FSS for distributed point functions (DPF), we extend our approach to distributed comparison and interval functions, achieving the most efficient key size to date. Our distributed comparison function exhibits a key-size reduction by a factor of  $q^{p-1}$ , where  $q$  denotes the size of the algebraic group used in the scheme's construction. The reduced key size of the comparison function has practical implications, particularly in applications like privacy-preserving machine learning (PPML), where thousands of comparison functions are employed in each neural network layer. To demonstrate the effectiveness of our improvements, we design and prototype-implement a scalable privacy-preserving framework for neural networks over distributed models. Specifically, we implement a distributed rectified linear unit (ReLU) activation function using our distributed comparison function, showcasing the efficacy of our proposed scheme.

**Keywords:** Function Secret Sharing · Distributed Comparison Function · ReLU

## 1 Introduction

Function secret sharing (FSS), introduced in [GI14, BGI15], is a natural extension of additive secret-sharing to functions. For a class  $\mathcal{F}$  of efficiently computable functions (with succinct descriptions)  $f : \{0, 1\}^n \rightarrow \mathbb{G}$  where  $\mathbb{G}$  is an Abelian group, an FSS scheme for  $\mathcal{F}$  allows splitting each  $f \in \mathcal{F}$  into  $p$  succinctly described functions  $f_i : \{0, 1\}^n \rightarrow \mathbb{G}$  for  $1 \leq i \leq p$  such that: (i)  $\sum_{i=1}^p f_i = f$ , and (ii) any  $(p - 1)$ -sized subset of the functions

---

E-mail: [cchaudhary278@kgpian.iitkgp.ac.in](mailto:cchaudhary278@kgpian.iitkgp.ac.in) (Chandan Kumar), [sikhar.patranabis@ibm.com](mailto:sikhar.patranabis@ibm.com) (Sikhar Patranabis), [debdeep@cse.iitkgp.ac.in](mailto:debdeep@cse.iitkgp.ac.in) (Debdeep Mukhopadhyay)



$f_i$  hides  $f$ . In other words, an FSS for  $\mathcal{F}$  enables succinct *additive* secret sharing of functions from  $\mathcal{F}$ . FSS has found wide use in various privacy-preserving techniques, including anonymous communication [CGBM15, ECGZB21, NSSD22, VSH22], private set intersection [TSS<sup>+</sup>20, DIL<sup>+</sup>20, DIL<sup>+</sup>22, GRS22, GRS23], secure computation of RAM program [DS17, BKKO20, GKW18, GO96, VHG23], secure aggregation and statistical analysis [BGI16, BBCG<sup>+</sup>21], encrypted search system [DFL<sup>+</sup>20], pseudorandom correlation and oblivious linear evaluation [BCG<sup>+</sup>19], and many more.

**Distributed Point Function (DPF).** A distributed point function (DPF), introduced in [GI14] and further studied in [BGI15, BGI16, BGIK22], is a specific instance of FSS. It can be seen as an FSS for the class  $\mathcal{F}$  of point functions  $f : \{0, 1\}^n \rightarrow \mathbb{G}$  ( $\mathbb{G}$  is an Abelian group with identity element  $1_{\mathbb{G}}$ ) such that  $f$  evaluates to  $1_{\mathbb{G}}$  on all but at most one input. For  $\alpha \in \{0, 1\}^n$  and  $\beta \in \mathbb{G}$ , we denote by  $f_{\alpha, \beta}$  the following point function:

$$f_{\alpha, \beta}(x) = \begin{cases} \beta, & x = \alpha, \\ 1_{\mathbb{G}}, & \text{Otherwise} \end{cases}$$

There are two variants of DPF in the state-of-the-art literature: the 2-party DPF and the  $p$ -party DPF (with  $p \geq 3$ ). The original 2-party DPF schemes proposed in [BGI15, BGI16] are based on pseudorandom generators (PRGs) or, more generally, one-way functions (OWFs). The construction was later extended to a wide range of function families, such as the family of interval functions, where  $f(x)$  evaluates to  $\beta$  for all inputs in the range  $[a, b]$  (for some  $a$  and  $b$  in the domain of  $f$ ) and  $0$  (identity element) for all other inputs. 2-party DPF construction is extensively studied in FSS literature due to its efficient key sizes (asymptotically polynomial in  $O(n)$  for a function with domain size  $N = 2^n$ ). The majority of applications primarily concentrate on 2-party DPF schemes. However, in real-world scenarios, enhancing trustworthiness involves distributing trust across many parties. Hence, the other variant of  $p$ -party DPF, built from symmetric-key primitives such as PRG or a public-key primitive such as seed-homomorphic PRGs, becomes essential for FSS literature. This study exclusively focuses on  $p$ -party DPF; henceforth, ‘‘DPF’’ refers to  $p$ -party DPF.

**Distributed Comparison Function (DCF).** Another very popular class of function in FSS literature is the distributed comparison function (DCF). FSS for the class  $\mathcal{F}^<$  of comparison function  $f^< : \{0, 1\}^n \rightarrow \mathbb{G}$  such that  $f^<$  evaluates to  $1_{\mathbb{G}}$  (an identity element of group  $\mathbb{G}$ ) for all inputs smaller than a threshold value else a particular element in  $\mathbb{G}$ . For a threshold value  $\alpha \in \{0, 1\}^n$  and a  $\beta \in \mathbb{G}$ , a comparison function  $f_{\alpha, \beta}^<$  can be represented as follows:

$$f_{\alpha, \beta}^<(x) = \begin{cases} \beta, & x \geq \alpha, \\ 1_{\mathbb{G}}, & \text{Otherwise} \end{cases}$$

Similar to DPF, DCF also exists in two variants: the 2-party DCF and the  $p$ -party DCF (with  $p \geq 3$ ). The 2-party DCF was introduced by Boyle et al. in [BGI15], and subsequent improvements in key sizes were made in [BGI19, RPB20]. 2-party DCF has been extensively studied in the literature such as [RPB20, JGB<sup>+</sup>24, HLC<sup>+</sup>23, Wag22, YJG<sup>+</sup>23, GJM<sup>+</sup>23] for constructing distributed rectified liner units (ReLU) function to enable secure computation in privacy-preserving machine learning. However, the  $p$ -party variant has received less attention. The only existing construction based on pseudo random generator (PRG) or more generally, one-way function, suffers from impractically large key sizes, making it unsuitable for practical applications.

**Key Sizes in DPF and DCF.** The naïve solution for  $p$ -party DPF and DCF is to additively secret share the evaluation table among  $p$ -parties, resulting in key sizes equal to the evaluation table. For a point or comparison function of the form  $f : \{0, 1\}^n \rightarrow \mathbb{G}$  (resp.

Table 1: Key size of distributed point function (DPF) where  $p$  denotes number of parties,  $2^n$  function domain size,  $m = \log_2(|\mathbb{G}|)$  for group  $\mathbb{G}$ ,  $\lambda$  denotes security parameter,  $\text{rows}$  denotes a row-sized vector, and  $\text{cols}$  denotes a column-sized vector, PP denotes public parameter.

Schemes	Assumptions	Key-Size	
		Share	PP
Boyle et al. [BGI15]	PRG	$2^{p-1} \cdot (\text{rows}) + 2^{p-1} \cdot (\text{cols})$	-
		$2^{n/2} 2^{(p-1)/2} (\lambda + 2^{(p-1)} m)$	-
Riposte [CGBM15]	Seed-homomorphic PRG	$2 \cdot (\text{rows}) + (\text{cols})$	-
		$\frac{2^{n/2}}{2^{(p-1)/2}} (2 \cdot \lambda + 2^{(p-1)} m)$	-
Proposed DPF	Seed-homomorphic PRG	$(\text{rows}) + (\text{cols})$	-
		$\frac{2^{n/2}}{2^{(p-1)/2}} (\lambda) + m$	$2^{n/2} 2^{(p-1)/2} (m)$

$f^< : \{0, 1\}^n \rightarrow \mathbb{G}$ ), the key size in the naïve construction for each party is  $|\mathbb{G}| * 2^n$ -bits as there are  $2^n$  entries in the table each of length  $|\mathbb{G}|$ . The huge key size of trivial construction renders it inefficient for large domains of the function. In the non-trivial construction, the one-dimension evaluation table is split into 2 or more dimensions. The key size for such non-trivial DPF and DCF schemes depends on how the corresponding evaluation table is split into rows and columns. The key size reported in Table 1 and 2 for DPF and DCF respectively is due to a 2-dimension evaluation table with column ( $\mu = \lceil 2^{n/2} \cdot 2^{(p-1)/2} \rceil$ ) and row ( $\nu = \lceil 2^n / \mu \rceil$ ) reported in [BGI15]. To date, the best known (additive)  $p$ -party DPF construction achieves a key size of  $(2^{n/2} 2^{(p-1)/2} (\lambda + 2^{p-1} \log_2 |\mathbb{G}|))^1$  (in the symmetric key setting) and  $(2^{n/2} 2^{-(p-1)/2} (2 \cdot \lambda + 2^{p-1} \log_2 |\mathbb{G}|))$  (in the public-key setting), where  $\lambda$  is the security parameter. Similarly, for a large number of parties, the DCF construction by Boyle et al. has an inefficient key size of  $2^{n/2} \cdot q^{(p-1)/2} \log_2(q)$ , where  $q$  denotes the size of the algebraic group used in the construction. This raises a critical question: can we build a  $p$ -party DPF/DCF scheme that achieves a smaller key size or, more generally, a flexible trade-off between key size and computational efficiency without compromising the succinctness of the overall DPF/DCF schemes? Motivated by the need for practical efficiency of DPF and DCF schemes, we ask the following question:

*Can we construct  $p$ -party FSS schemes (where  $p \geq 3$ ) with **more compact** key sizes for point, comparison, and interval functions?*

## 1.1 Our Contributions

In this paper, we answer the above questions in the affirmative. Our study of  $p$ -party FSS yields the following main results:

**DPF with Shorter Keys.** For the  $p$ -party case, our DPF construction achieves a key size of  $(2^{n/2} 2^{-(p-1)/2} (\lambda + 2^{(p-1)} \log_2 |\mathbb{G}|) + |\mathbb{G}|)$  ( $\lambda$  represents the security parameter), which is asymptotically  $O(2^p)$  times smaller than  $(2^{n/2} 2^{(p-1)/2} (\lambda + 2^{(p-1)} \log_2 |\mathbb{G}|))$  (in the symmetric-key setting) and concretely  $2^{n/2} 2^{-(p-1)/2} (\lambda)$ -bit smaller than  $(2^{n/2} 2^{-(p-1)/2} (2 \cdot \lambda + 2^{(p-1)} \log_2 |\mathbb{G}|))$  (in the public-key setting) key size of the best (additive) DPF construction to date (ref. Table 1). This reduction in key size is due to utilising a seed homomorphic pseudorandom generator that allows using only one seed per row of the evaluation table, as opposed to the approach in [BGI15], which required  $2^{p-1}$  seeds per row (see Figure 1). In the public key settings, we reduce the key size by a row-sized vector (a vector with a length equal to the number of rows). This improvement is made possible by introducing an additional comparison that the decoder must perform after DPF evaluation by different parties.

<sup>1</sup>The key size reported in [BGI15] at page 15 seems to have some error. They report their key size as  $(2^{n/2} 2^{(p-1)/2} (\lambda + m))$  with  $m = \log_2 |\mathbb{G}|$ , however, if we follow their row and column splitting and perform an addition of  $\nu \lambda \cdot 2^{p-1} + \mu m \cdot 2^{p-1}$ , the key size comes out to be  $(2^{n/2} 2^{(p-1)/2} (\lambda + 2^{p-1} m))$ .

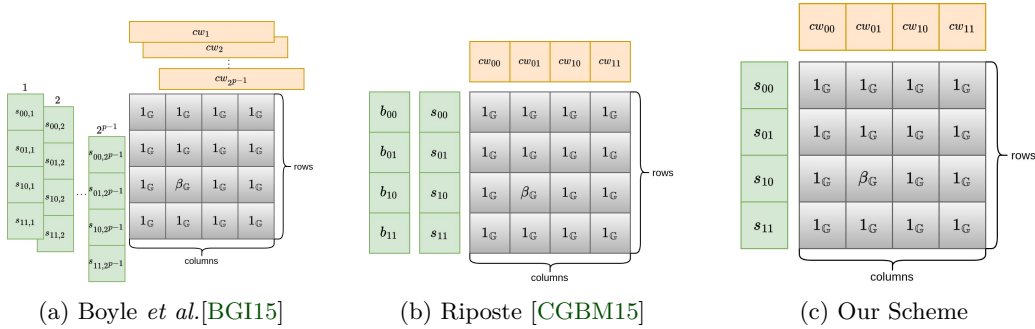


Figure 1: Pictorial representation of key size for different  $p$ -party DPF.

Though Table 1 provides key sizes for row and column splitting mentioned in [BGI15], one can use any splitting for row and column, and our proposed scheme will always show the improvement over the prior two schemes. For instance, let us consider a particular scenario in which the rows and columns are equally split with the following values:  $p = 5, n = 32, \text{row} = 2^{16}, \text{column} = 2^{16}, \lambda = 256, |\mathbb{G}| = 256$ . According to Boyle *et al.* [BGI15], the key size is 32MB, Riposte [CGBM15] reports a key size of 6MB, while our proposed scheme has a key size of approximately 4MB.

**Extension to Comparison and Interval Functions.** We extend our DPF to achieve the first  $p$ -party FSS scheme for a wider class of functions that are useful for practical applications, including comparison and interval functions. Our construction of  $p$ -party FSS for comparison functions achieves similarly small key sizes (and similarly flexible tradeoffs between key size and computational overheads). The previous  $p$ -party comparison function by Boyle *et al.* [BGI15] incurs a huge key size as they generate  $q^{(p-1)}$  seeds per row (ref. Table 2), restricting the construction from achieving practicality. We achieved a substantial reduction in key size at the cost of two extra comparisons, which the decoder must perform once all parties' partial evaluations are available. As shown in Table 2, the share size (secret part of the key) for the distributed comparison function (DCF) in our scheme remains the same as for the DPF, while the public parameter size has increased by a factor of  $3\times$ . For an interval function, we show a naïve solution where we utilize two DCFs to build a distributed interval function (DIF). Consequently, the key size for DIF is  $2\times$  that of the DCF.

**Practical Use Case.** We consider a privacy-preserving machine learning (PPML) scenario where a trained model is distributed (secretly shared) over a specific number of servers. For a client, the objective is to perform inference on the distributed model while maintaining data privacy. We eventually facilitate our compact key-size comparison function to develop a distributed ReLU function. We build this application under the assumption that the client is assisted by a trusted decoder, which is resourceful enough to execute small operations like addition and comparison locally. The secure computation with preprocessing through FSS construction as described in [BCG+21, BGI19] enables the implementation of distributed ReLU functions employing *offset* functions. In such cases, a client or a specified party has to perform the aggregation (decode) in all FSS-based schemes. However, in our proposed schemes, they must also perform two comparisons besides the basic decoder's operation. We provide several evaluation results for distributed ReLU to demonstrate the effectiveness of the distributed ReLU function.

As already noted, utilizing prior  $p$ -party DCF schemes to construct ReLU would result in significant inefficiencies due to its huge key size. To practically implement ReLU in such contexts, it is crucial for the DCF to have smaller key sizes, given that thousands of DCFs are employed to evaluate a single layer of a privacy-preserving machine learning (PPML) model. Before detailing our efficient DPF and DCF constructions, we discuss the challenges

Table 2: Key size of distributed comparison function (DCF) and distributed interval function (DIF) where  $(m = \log_2(q) = \log_2(|\mathbb{G}|))$ , with  $q$  being the size of the group  $\mathbb{Z}_q$  from where seeds for the PRG are randomly selected, PP represents public parameter

Schemes	Assumptions	Key-Size	
		Share-Size	PP
Boyle et al.[BGI15]	PRG	$2^{n/2} \cdot q^{(p-1)/2} \log_2(q)$	-
Proposed DCF	Seed-homomorphic PRG	$\frac{2^{n/2}}{2^{(p-1)/2}}(\lambda) + m$	$\frac{2^{n/2}}{2^{(p-1)/2}}(3 \cdot 2^{(p-1)}m)$
Proposed DIF	Seed-homomorphic PRG	$2 \cdot (\frac{2^{n/2}}{2^{(p-1)/2}}(\lambda) + m)$	$2 \cdot (\frac{2^{n/2}}{2^{(p-1)/2}}(3 \cdot 2^{(p-1)}m))$

of extending a prior  $p$ -party DPF directly to a  $p$ -party DCF. Along the way, we also justify the choice of incorporating a non-linear decoder in our design.

**Choice of Non-linear Decoder.** The barrier to constructing DCF from prior DPF constructions lies in the difference in their evaluation tables. DCF’s evaluation table contains three types of rows (see Table 4), requiring three sets of correction words or embedding all target rows as part of corrections. While Boyle’s OWF-based solution adopts the latter, it leads to impractical key sizes. Natural approaches to extending Riposte’s DPF construction to the construction of DCF with three correction words pose the following challenges.

- In the  $p$ -party DPF scheme by Riposte [CGBM15], there is only one correction word (referred to as “v” in their paper) to embed  $\beta$  of the target row, which enables linear reconstruction in their scheme. Specifically, the partial computation on an input  $x = (\gamma, \delta)$ , with  $\gamma$  row and  $\delta$  column is  $g[\delta] + b[\gamma]v[\delta]$ , where  $g$  represents the seed-homomorphic PRG,  $b$  is a vector with all 0s except a 1 at index  $\gamma$ , and  $v$  denotes correction words. This expression facilitates linear/additive reconstruction when  $v[\delta]$  is fixed, ensuring correct evaluation in DPF.
- Conversely, in DCF, the evaluation table includes three types of rows: all 0’s, all  $\beta$ ’s, and mixed rows (see Table 4). Embedding  $\beta$  here requires more than one correction word since  $\beta$  appears in two different types of rows. Consequently, non-linear reconstruction is necessary to ascertain which correction word contributed to the correct evaluation.
- When extending Riposte’s DPF to DCF utilising the concept of more than one correction word, the partial evaluation requires computation for all correction words independently. However, only one of them results in the correct evaluation of DCF. This mandates a comparison in the final evaluation, prompting the introduction of a non-linear decoder. Despite incorporating a non-linear decoder, comparing three independent partial computations remained challenging. Therefore, we adjusted the partial computation expression and released correction words as public parameters to assist in the comparison process.

In summary, extending Riposte-based DPF to DCF encounters several seemingly inherent challenges, leading us to pursue an alternate approach for designing DCFs with non-linear decoding/reconstruction.

## 1.2 Comparison with FSS from Public-Key Techniques

In this subsection, we present a comparison of our proposed approach with existing approaches for designing FSS schemes from public-key techniques.

**Comparison with Riposte [CGBM15].** Our scheme has two significant advancement compared to Riposte [CGBM15]:

- Our core techniques yield a DPF construction where the key size is smaller than that of Riposte. Concretely, Riposte [CGBM15] uses two vectors of column length, while our construction requires only one column vector (ref. Table 1 and Figure 1). While this is a relatively simple and natural improvement on top of Riposte, it paves the way for our main contributions, as explained below.
- We propose a novel distributed comparison function leveraging a seed-homomorphic pseudorandom generator (PRG), paving the way for practical implementation of the  $p$ -party DCF. The earlier proposed distributed comparison function by [BGI15] incurs a key size overhead of  $O(q^p)$ , where  $q = |\mathbb{G}|$ , for some group  $\mathbb{G}$  (see Table 2). Our DCF construction takes advantage of an observation to identify three distinct types of rows present in the evaluation table of DCF (see Table 4). This approach enables us to utilize three correction word vectors, resulting in the most compact distributed comparison function to date. Reducing the key size of DCF enhances the efficiency of applications such as distributed machine learning inference (concretely, distributed ReLU computation), where servers have to potentially store an enormous number of keys.

**Comparison with Spooky Encryption [DHRW16].** Spooky encryption, introduced in [DHRW16], yields an approach for realizing FSS for general functions. However, it relies on computationally heavy public-key cryptographic tools such as multi-key fully homomorphic encryption (FHE), for which practically efficient constructions are rare. Our approach, which is based on seed-homomorphic PRGs, is significantly more lightweight and is amenable to more practically efficient instantiations based on either discrete log-hard cyclic groups or lattice-based assumptions.

### 1.3 Applications

Our proposed techniques yield better practical efficiency for all practical applications of FSS where a client (assisted by a trusted decoder) engages in a protocol with multiple servers to execute aggregate and comparison operations (where the function to be computed is distributed across the servers). In this subsection, we discuss one such application.

**Distributed Machine Learning Inference.** The main practical application that we focus on in this paper is machine learning inference, where the (pre-trained) model is distributed across multiple servers, and inference is performed on the distributed model. We first explain the system and security models and then describe the technique of performing inference over a distributed (pre-trained) model.

A model owner trains a model, potentially employing a distributed approach, and securely distributes the model parameters among a set of  $p$  servers. This distribution ensures that no group of  $(p - 1)$  servers may converge to get the information about model parameters. Consider  $M$  as the trained model, where  $M_1, M_2, \dots, M_p$  represent  $p$  shares of model  $M$  that are distributed to the respective servers. Specifically, the shares of the model basically represent the shares of model parameters such as weights. A client seeking to provide an input, say  $x$ , to obtain an inference from the distributed model desires to prevent any unauthorized disclosure of their data to the servers. Hence, the client does not send its data in plain, rather it masks the input and transmits the masked input to each server independently. The server partially computes the function over the masked input and returns the intermediate partial inference to the client. The client finally aggregates and decodes the result to prepare the input for the computation of subsequent layers. The inference on the (distributed) model is divided into multiple layers, which are categorised as linear layers and non-linear layers. For linear layers, we employ the techniques of Beaver’s triples, which allows us to perform the multiplication of model weights and masked input securely. For the non-linear layers, we utilize our proposed FSS



technique of distributed comparison function to perform the computations of non-linear layers such as ReLU. However, the distributed comparison function in plain does not allow computation on masked input. Hence, we utilize the concept of *offset* comparison function [BCG<sup>+</sup>21], which manipulates masked input and produces a masked output. Below, we briefly explain the idea of secure computation of ReLU, first for the 2-party computation and later extending it using our proposed distributed comparison function for  $p$ -party scenario.

**Secure Computation of ReLU.** The FSS-based approach of secure computation comprises of computation in preprocessing mode, also called as online-offline mode. To securely compute a gate  $g$  using two parties  $P_0$  and  $P_1$ , it follows the following invariant: for the input ( $x_{in}$ ) and output ( $x_{out}$ ) in  $g$ , both parties learn the masked values  $x_{in} + r_{in}$  and  $x_{out} + r_{out}$  respectively where  $r_{in}$  and  $r_{out}$  are input and output masks generated as correlated randomness. It is easy to achieve this invariant at the input level, as for any input  $x_{in}$ , owned by party  $P_\sigma$  ( $\sigma \in \{0, 1\}$ ), this party can compute and send  $x_{in} + r_{in}$  to the other party. For the invariant to hold for the output wire, a trusted dealer (which can be emulated by a 2PC protocol; for a detailed description, refer to Appendix A of [BCG<sup>+</sup>21]) uses an FSS scheme for the class of *offset* functions  $\mathcal{F}$  that includes all functions of the form  $g_{r_{in}, r_{out}}(x_{in}) = g(x_{in} - r_{in}) + r_{out}$ . The dealer splits the function  $g_{r_{in}, r_{out}}$  into two functions with keys  $k_0, k_1$ , and delivers each key  $k_\sigma$  to party  $P_\sigma$ . Now, each party evaluates their FSS share on common masked input  $x_{in} + r_{in}$  and obtains additive shares of  $x_{out} + r_{out}$ , which they can exchange among each other and maintain the invariant of masked output values. Ultimately, the dealer discloses the mask ( $r_{out}$ ) of the output wire to both parties to reconstruct the output. Now, considering the above gate as a comparison function, one can compute ReLU securely as follows: Let  $\text{ReLU}_{r_{in}}(x_{in}) = \text{ReLU}(x_{in} - r_{in})$ , where  $r_{in}$  is the input mask, be the *offset* function for ReLU. One can readily verify that for a given input  $x_{in} + r_{in}$ , the *offset* function eventually evaluates  $\text{ReLU}(x_{in})$ . We can write  $\text{ReLU}_{r_{in}}(x_{in}) = x_{in} - r_{in}$ , if  $x_{in} \geq r_{in}$  and 0 otherwise, as  $f_{\alpha, \beta}^<(x)$ , representing it in the form of spline polynomial with coefficients  $\beta = (\beta_0, \beta_1)$ , where  $(\beta_0, \beta_1) = (1, -r_{in})$  if  $x_{in} > r_{in}$  and  $(\beta_0, \beta_1) = (0, 0)$  otherwise. After computing shares of  $(\beta_0, \beta_1)$ , parties can locally compute shares of  $[\beta_0](x_{in} + r_{in}) + [\beta_1]$  which in fact are shares of the  $\text{ReLU}_{r_{in}}(x_{in})$  function as  $[\beta_0]$  is share of 1 and  $[\beta_0]$  is share of  $-r_{in}$  for the case  $x_{in} \geq r_{in}$ .

**Extending secure 2-party ReLU to  $p$ -parties.** For our proposed application of distributed machine learning inference, we extend the linear layer as well as non-linear layer computation to  $p$ -parties. For the linear layer computation using Beavers triples, let us suppose for two parties  $P_0$  and  $P_1$ , we generated beavers triple as  $(a, [ab]_0)$  for party  $P_0$  and  $(b, [ab]_1)$  for party  $P_1$  then we keep the first triples as it is while we further generate  $p$  shares of  $(b, [ab]_1)$  as  $(b_1, [ab]_{11}), \dots, (b_p, [ab]_{1p})$  and distribute the shares to  $p$ -parties respectively. Similarly, for the non-linear layers like ReLU, we utilise our  $p$ -party DCF in *offset* (shifted) mode, which enables distributed computation of ReLU with compact key sizes. Additionally, the output mask  $r_{out}$  is secret shared among  $p$ , which helps maintain the secure computation invariant in our application. Unlike two-party FSS-based ReLU,  $p$ -party also necessitates decoding after each computation of each ReLU layer. We provide the detailed description of  $p$ -party ReLU in Section 4.

## 1.4 Technical Overview of Our Constructions

In this section, we present a high-level overview of our proposed DPF and DCF constructions considering a toy example of a point function of the form  $f_{\alpha, \beta} : \{0, 1\}^4 \rightarrow \mathbb{G}$ , for  $\alpha = 1001$  and some  $\beta \in \mathbb{G}$ , where  $(\mathbb{G}, \otimes)$  is an Abelian group.

**Notations.** For any  $g \in \mathbb{G}$ , we denote by  $\text{Inv}(g)$  the inverse of  $g$ . Also, we use  $1_{\mathbb{G}}$  to denote the identity element of  $\mathbb{G}$  and similarly  $0_S$  denotes the identity element of group  $(S, \oplus)$ .

Overloading the notations, for  $u = (g_1, g_2, g_3, g_4) \in (\mathbb{G})^4$  and  $v = (h_1, h_2, h_3, h_4) \in (\mathbb{G})^4$ , we write  $u \otimes v$  to denote the component-wise group operation as follows,

$$u \otimes v := (g_1 \otimes h_1, g_2 \otimes h_2, g_3 \otimes h_3, g_4 \otimes h_4).$$

**Seed-Homomorphic PRG.** We will consider an example of seed-homomorphic PRG of the form  $\mathcal{G} : S \rightarrow (\mathbb{G})^4$  where  $(S, \oplus)$  is also an Abelian group s.t.

$$\mathcal{G}(s_1 \oplus s_2) = \mathcal{G}(s_1) \otimes \mathcal{G}(s_2).$$

For example, suppose that  $\mathbb{G}$  is a cyclic group of prime order  $q$  such that the DDH assumption holds over  $\mathbb{G}$ . Then for a given list of generators  $g_1, g_2, g_3, g_4 \leftarrow \mathbb{G}$ , the following is an example of such a seed-homomorphic PRG:  $\mathcal{G} : S \rightarrow (\mathbb{G})^4$  where

$$\mathcal{G}(s \in S) = (g_1^s, g_2^s, g_3^s, g_4^s).$$

The homomorphism of the above PRG can be expressed as follows:

$$\begin{aligned} \mathcal{G}(s) &= \mathcal{G}(s_1 \oplus s_2) = \mathcal{G}(s_1) \otimes \mathcal{G}(s_2) \\ &= (g_1^{s_1}, g_2^{s_1}, g_3^{s_1}, g_4^{s_1}) \otimes (g_1^{s_2}, g_2^{s_2}, g_3^{s_2}, g_4^{s_2}) \end{aligned}$$

The component-wise group operations gives  $(g_1^{s_1 \oplus s_2}, g_2^{s_1 \oplus s_2}, g_3^{s_1 \oplus s_2}, g_4^{s_1 \oplus s_2}) = (g_1^s, g_2^s, g_3^s, g_4^s) = \mathcal{G}(s)$ .

**Overview of Our DPF Construction.** For ease of exposition, we show our construction for 3-parties  $(P_1, P_2, P_3)$  where each party is provided with their individual key to partially compute the function  $f_{1001, \beta}$  on some given input  $x \in \{0, 1\}^4$ . The 3-party DPF construction ensures that the partial computations of all the 3-parties, when reconstructed, produce  $f_{1001, \beta}(x)$ , while the partial computations of 2 or fewer parties reveal no information about the point function parameters  $\alpha = 1001$  and  $\beta$ . For the given function  $f_{1001, \beta}$ , the evaluation table can be given as Table 3, where  $\beta$ 's position can be located by splitting  $\alpha = 1001$  into two parts  $(10, 01)$ . The first part ( $\gamma = 10$ ) denotes the row number, while the second part ( $\delta = 01$ ) denotes the column number of  $\beta$ 's position in the evaluation table.

Table 3: Two-dimensional evaluation table for  $f_{1001, \beta}$

$f_{1001, \beta}$	00	01	10	11
00	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$
01	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$
10	$1_{\mathbb{G}}$	$\beta_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$
11	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$

**Key Generation.** The key generation algorithm consists of two phases: 1) correction word generation, and 2) secret share generation. We iterate each phase one by one.

- *Correction Word Generation:* In this phase, a trusted dealer randomly selects a seed  $s_{\gamma} \in S$  where  $\gamma = 10$  corresponds to the target row of the evaluation table. We call the row (resp. column) the target row (resp. column) if that row (resp. column) corresponds to the position of  $\beta$  in the evaluation table. The dealer then computes the accompanying errors with the target  $\beta$  for all the columns corresponding to the target row  $\gamma$ , as correction words. Here, the correction words  $\text{cw}_{\delta'}, \forall \delta' \in \{00, 01, 10, 11\}$  are computed as follows;

$$\text{cw}_{\delta'} = \begin{cases} \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}^{(\delta')}(s_{\gamma})), & \delta' = 01 \text{ (target column)} \\ 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}^{(\delta')}(s_{\gamma})), & \text{otherwise,} \end{cases}$$



(here,  $\text{Inv}(\cdot)$  denotes the inverse of group element,  $\mathcal{G}^{(\delta')}$  denotes the  $\delta'$ -th component of the output of  $\mathcal{G}$ , for example, if suppose  $\mathcal{G}$  outputs  $(1_{\mathbb{G}}, \beta_{\mathbb{G}}, 1_{\mathbb{G}}, 1_{\mathbb{G}})$ , then 00-th component is  $1_{\mathbb{G}}$ , 01-th component is  $\beta_{\mathbb{G}}$ , 10-th component is  $1_{\mathbb{G}}$ , and 11-th component is  $1_{\mathbb{G}}$ ). Note that the correction words  $\text{CW} = (\text{cw}_{00}, \text{cw}_{01}, \text{cw}_{10}, \text{cw}_{11})$  have been computed only for the target row; however, it remains indistinguishable for an adversary to guess which row it has been computed for.

- *Secret Share Generation:* In this phase, the dealer generates seed share for the parties corresponding to each row of the evaluation table. The key idea is that for all rows except  $\gamma = 10$ , it generates shares of  $0_S$  while for row  $\gamma = 10$ , it generates shares of  $s_{\gamma}$ . If  $\text{sh}_{i,j}$  denotes the share for  $i$ -th party and  $j$ -th row, then  $\{\text{sh}_{1,00}, \text{sh}_{2,00}, \text{sh}_{3,00}\}$ ,  $\{\text{sh}_{1,01}, \text{sh}_{2,01}, \text{sh}_{3,01}\}$ , and  $\{\text{sh}_{1,11}, \text{sh}_{2,11}, \text{sh}_{3,11}\}$  are the shares of  $0_S$  independently generated for all the 3-parties corresponding to rows 00, 01, and 11. These shares adhere to the following relation,

$$\text{sh}_{1,j} \oplus \text{sh}_{2,j} \oplus \text{sh}_{3,j} = 0_S, \forall j \in \{00, 01, 11\}$$

For row  $\gamma = 10$ ,  $\{\text{sh}_{1,10}, \text{sh}_{2,10}, \text{sh}_{3,10}\}$  are the shares of  $s_{\gamma}$  for all the 3-parties and follow the relation,

$$\text{sh}_{1,j} \oplus \text{sh}_{2,j} \oplus \text{sh}_{3,j} = s_{\gamma}$$

The corresponding shares  $\text{sh}_{i,j}$  are sent to the corresponding party  $P_i$  for all  $j \in \{00, 01, 10, 11\}$  and  $\text{CW} = (\text{cw}_{00}, \text{cw}_{01}, \text{cw}_{10}, \text{cw}_{11})$  is released as a public parameter.

**Evaluation and Reconstruction.** During evaluation, for a given input  $x = (\gamma^*, \delta^*)$ , each party  $P_i$  uses shares corresponding to row  $\gamma^*$  and perform partial computation as follows,

$$\text{PartComp}_i = \mathcal{G}^{(\delta^*)}(\text{sh}_{i,\gamma^*})$$

In the reconstruction phase, a decoder collects  $\text{PartComp}_1$ ,  $\text{PartComp}_2$ , and  $\text{PartComp}_3$  from the parties and use public parameter  $\text{cw}_{\delta^*}$  to obtain the evaluation of result as follows,

$$\text{Res} = \text{cw}_{\delta^*} \otimes \text{PartComp}_1 \otimes \text{PartComp}_2 \otimes \text{PartComp}_3$$

Simplifying the above equation,  $\text{Res} = \text{cw}_{\delta^*} \otimes \mathcal{G}^{(\delta^*)}(\text{sh}_{1,\gamma^*}) \otimes \mathcal{G}^{(\delta^*)}(\text{sh}_{2,\gamma^*}) \otimes \mathcal{G}^{(\delta^*)}(\text{sh}_{3,\gamma^*}) = \text{cw}_{\delta^*} \otimes \mathcal{G}^{(\delta^*)}(\text{sh}_{1,\gamma^*} \oplus \text{sh}_{2,\gamma^*} \oplus \text{sh}_{3,\gamma^*})$ . Now, we exhaustively proof the correctness of the DPF by considering three different examples. In the first example, input  $x = 1101$ , with  $\gamma^* = 11$  and  $\delta^* = 01$  cover the case where neither  $\gamma^*$  nor  $\delta^*$  matches the target row/column. For this case,  $\text{Res} = \text{cw}_{01} \otimes \mathcal{G}^{(01)}(0_S) = \text{cw}_{01}$  (Note that the shares for row 11 were generated due to the formula  $\text{sh}_{1,j} \oplus \text{sh}_{2,j} \oplus \text{sh}_{3,j} = 0_S$ ). Whenever  $\text{Res}$  equals to  $\text{cw}_{\delta_x}$  or  $1_{\mathbb{G}}$ , the decoder outputs  $1_{\mathbb{G}}$ . In the second example, input  $x = 1001$ , with  $\gamma^* = 10$  and  $\delta^* = 01$  cover the case where both  $\gamma^*$  and  $\delta^*$  matches the target row/column. so reconstructing  $\text{sh}_{1,10} \oplus \text{sh}_{2,10} \oplus \text{sh}_{3,10}$  will produce  $s_{\gamma}$ . Hence,  $\text{Res} = \text{cw}_{01} \otimes \mathcal{G}^{(01)}(s_{10}) = \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}^{(01)}(s_{10})) \otimes \mathcal{G}^{(01)}(s_{10}) = \beta_{\mathbb{G}}$ . In the third example, input  $x = 1011$  with  $\gamma^* = 10$  and  $\delta^* = 11$  cover the case where  $\gamma^*$  matches the target row while  $\delta^*$  does not match the target row. One can verify the evaluation for this example to be equal to  $1_{\mathbb{G}}$ .

The security of DPF guarantees that no subset of size less than  $p$  parties can retrieve any information about function parameter  $\alpha$  and  $\beta$ . Note that the above guarantee holds only if the discrete log problem is hard in the output group  $(\mathbb{G}, \otimes)$ . This hardness assumption is required to computationally hide the secret information stored in public parameter (correction words).

**Overview of Our DCF Construction.** We adhere to the previous example of three parties  $(P_1, P_2, P_3)$  for constructing a distributed comparison function  $f_{1001,\beta}^<$  to compute any given input  $x \in \{0, 1\}^4$ . Unlike DPF, DCF evaluation table (see Table 4) can have

more than one target row. Like in Table 4, row 10 and 11 can contribute to the evaluation of  $\beta_{\mathbb{G}}$ . To handle multiple target rows, we need more than one correction words. Our construction takes advantage of an observation to identify three distinct types of rows present in the evaluation table (see Table 4) which enables us to utilize three correction words ( $CW^1, CW^2, CW^3$ ) for DCF construction. Correction word  $CW^1$  corresponds to rows with all entries  $1_{\mathbb{G}}$ .  $CW^2$  corresponds to row with mixed entries, and  $CW^3$  corresponds to rows with only  $\beta_{\mathbb{G}}$  as entries. For the given function  $f_{1001,\beta}^<$  with  $\gamma = 10$  and  $\delta = 01$ ,  $\gamma$  and  $\delta$  denotes the position in evaluation table where entries for  $\beta_{\mathbb{G}}$  begins. Note that rows ( $< \gamma$ ) have only  $1_{\mathbb{G}}$  as entries, and similarly, rows ( $> \gamma$ ) have only  $\beta_{\mathbb{G}}$  as entries in the evaluation table.

Table 4: Two dimensional evaluation table for comparison function  $f_{1001,\beta}^<$

$f_{1001,\beta}^<$	00	01	10	11
00	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$
01	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$	$1_{\mathbb{G}}$
10	$1_{\mathbb{G}}$	$\beta_{\mathbb{G}}$	$\beta_{\mathbb{G}}$	$\beta_{\mathbb{G}}$
11	$\beta_{\mathbb{G}}$	$\beta_{\mathbb{G}}$	$\beta_{\mathbb{G}}$	$\beta_{\mathbb{G}}$

**Key Generation.** Similar to DPF, the key generation algorithm in DCF also has two phases: 1) correction word generation, 2) secret share generation. We will iterate through both these phases one by one.

- *Correction Word Generation:* As noted above, we need 3 correction words for each type of rows. The trusted dealer randomly selects 3-seeds,  $(s_{\gamma^1}, s_{\gamma^2}, s_{\gamma^3}) \in (S)^3$ . Seed  $s_{\gamma^1}$  corresponds to rows that have only  $1_{\mathbb{G}}$ 's i.e., rows 00 and 01. Seed  $s_{\gamma^2}$  corresponds to the row that has mixed  $1_{\mathbb{G}}$ 's and  $\beta_{\mathbb{G}}$ 's, i.e., row 10. Similarly, seed  $s_{\gamma^3}$  corresponds to rows that have only  $\beta$ 's as their entry, i.e., row 11. We randomly select 3 correction words (see proof 3.3 for why we are selecting it randomly) and overwrite the correction words using a formula similar to that of DPF. The first correction word  $CW^1 = (cw_{00}^1, cw_{01}^1, cw_{10}^1, cw_{11}^1)$  is calculated as  $cw_{\delta'}^1 = 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}^{\delta'}(s_{\gamma^1}))$ ,  $\forall \delta' \in \{00, 01, 10, 11\}$ , the second correction word  $CW^2 = (cw_{00}^2, cw_{01}^2, cw_{10}^2, cw_{11}^2)$  is calculated as  $cw_{\delta'}^2 = \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}^{\delta'}(s_{\gamma^2}))$ ,  $\forall \delta' \in \{01, 10, 11\}$  and  $cw_{\delta'}^2 = 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}^{\delta'}(s_{\gamma^2}))$ ,  $\forall \delta' = 00$ . Similarly, the third correction word  $CW^3 = (cw_{00}^3, cw_{01}^3, cw_{10}^3, cw_{11}^3)$  is calculated as  $cw_{\delta'}^3 = \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}^{\delta'}(s_{\gamma^3}))$ ,  $\forall \delta' \in \{00, 01, 10, 11\}$ .
- *Secret Share Generation:* In this phase, the dealer generates seed shares for the parties corresponding to each row of the evaluation table. The main idea is to generate the secret shares of  $0_s$  for rows where all entries are  $1_{\mathbb{G}}$ . For the mixed rows, it generates the secret shares of  $s_{\gamma^2}$  and for rows with all  $\beta_{\mathbb{G}}$ , it generates the secret shares of  $s_{\gamma^3}$ .

**Evaluation and Reconstruction.** The partial evaluation of DCF differs from DPF in the sense that every party  $P_i$  calculates two partial computations,  $\text{PartComp}_{i,1}$  and  $\text{PartComp}_{i,2}$ . For a given input  $x = (\gamma^*, \delta^*)$ ,  $\text{PartComp}_{i,1}$  is computed on column  $\delta^*$ , and  $\text{PartComp}_{i,2}$  is computed on column  $\delta^* \pm 1$  keeping the row as  $\gamma^*$  in both computations. The decoder uses different combinations of partial computations and correction words  $CW^1, CW^2, CW^3$  to determine the correct final evaluation. It first computes the result as  $\text{Res}^1 = cw_{\delta^*}^1 \otimes \text{PartComp}_{1,1} \otimes \text{PartComp}_{2,1} \otimes \text{PartComp}_{3,1}$ . If  $\text{Res}^1$  equals to  $cw_{\delta^*}^1$ , it outputs the result as  $1_{\mathbb{G}}$ , otherwise it computes  $\text{Res}^2$  and  $\text{Res}^3$  as  $\text{Res}^2 = cw_{\delta^*}^2 \otimes \text{PartComp}_{1,1} \otimes \text{PartComp}_{2,1} \otimes \text{PartComp}_{3,1}$  and  $\text{Res}^3 = cw_{\delta^* \pm 1}^3 \otimes \text{PartComp}_{1,2} \otimes \text{PartComp}_{2,2} \otimes \text{PartComp}_{3,2}$ . If  $\text{Res}^2$  equals  $\text{Res}^3$ , it outputs the result as  $\text{Res}^2$ , otherwise, it computes  $\text{Res}^4 = cw_{\delta^*}^2 \otimes \text{PartComp}_{1,1} \otimes \text{PartComp}_{2,1} \otimes \text{PartComp}_{3,1}$  as the final output.

Now, we evaluate 4 different inputs to cover all the different possible cases of evaluations. 1)  $x = 0110$  with  $\gamma^* = 01$  and  $\delta^* = 10$  cover the case where input  $x$  is smaller than  $\alpha$  and  $\gamma^* = 01$  represents the rows with all entries as  $1_{\mathbb{G}}$ . For this case, the shares of  $0_S$  were generated, and hence the combination of partial computation will output  $1_{\mathbb{G}}$ . Therefore,  $\text{Res}^3$  will equal to  $\text{cw}_{\delta^*}^1$ . In this case, the decoder will stop here and output the result as  $1_{\mathbb{G}}$ . 2)  $x = 1101$  with  $\gamma^* = 11$  and  $\delta^* = 01$ , covers the case when input  $x$  is bigger than  $\alpha$ . In this case, the combination of partial computations will reconstruct the seed  $s_{\gamma^3}$ . The decoder computes  $\text{Res}^1$ , however this result will not be equal to  $\text{cw}_{\delta^*}^1$  as  $\text{cw}_{\delta^*}^1$  embeds seed  $s_{\gamma^1}$  inside it. So, decoder computes  $\text{Res}^2$  and  $\text{Res}^3$ , and  $\text{Res}^2$  equals to  $\text{Res}^3$  as  $\gamma^* > \gamma$  and for such rows all columns have embedding of  $\beta_{\mathbb{G}}$ . The decoder will stop here and output  $\text{Res}^2$  which should be equal to  $\beta_{\mathbb{G}}$ . 3)  $x = 1000$  with  $\gamma^* = 10$  and  $\delta^* = 00$ , covers the case when row  $\gamma^* = \gamma$ , still the output should be  $1_{\mathbb{G}}$ . In this case, neither  $\text{Res}^1 == \text{cw}_{\delta^*}^1$  nor  $\text{Res}^2 == \text{Res}^3$  as the correction word  $\text{cw}_{\delta^*}^2$  can only evaluate it to  $1_{\mathbb{G}}$ . 4)  $x = 1010$  with  $\gamma^* = 10$  and  $\delta^* = 10$  works similar to the previous case, and one can verify the output to be  $\beta_{\mathbb{G}}$ .

The security of DCF guarantees that no subset of size less than  $p$  parties can retrieve any information about function parameters  $\alpha$  and  $\beta$ . It is important to note that the above guarantee holds only if the discrete log problem is hard in the output group  $(\mathbb{G}, \otimes)$ . This hardness assumption is required to computationally hide the secret information stored in public parameter (correction words).

**Regarding Function Privacy.** In an FSS scheme, function privacy ensures that unless an adversary has evaluated and decoded all possible values of input  $x$ , the function parameters remains indistinguishable from any random element in the input set. In the above FSS with non-linear decoding, an adversary who can see all parties' partial computations can leak function parameters by querying for  $2^{n/2}$  values of  $x$ . The initial  $2^{n/2}$  queries correspond to all the rows of the evaluation table. Also, exactly one of the rows is the target row, which can be leaked by looking at the aggregated values of partial computations. Hence, we additionally ensure parameter hiding by requiring a trusted decoder which is natural for our target application, but is not universally true, and we leave achieving unconditional function privacy for FSS with non-linear decoding as an open question. Our idea is to mask the partial output of each party with shares of output mask  $r_{out}$ . Specifically, the key generation algorithm, along with FSS keys also generates  $p$  shares of  $r_{out}$  as  $[r_{out}]_1, \dots, [r_{out}]_p$ . Then, it sends  $r_{out}$  to the trusted decoder, while share  $[r_{out}]_i$  to party  $P_i$ . Each party masks their partial computation  $\text{PartComp}_i$  by computing  $\text{PartComp}_i = \text{PartComp}_i \otimes [r_{out}]_i$ . In this setup, a collective effort of  $p$  parties will not divulge any information about function parameters, as the output mask  $r_{out}$  is retained with the decoder. Thus, an adversary must evaluate and aggregate the function over all  $2^n$  possible values of  $x$  to gain any information about function parameters.

## 2 Preliminaries and Background

In this section, we introduce the notations, cryptographic background, and definitions for function secret sharing with the non-linear decoder.

### 2.1 Notations

We write  $x \xleftarrow{R} \mathcal{X}$  to represent that an element  $x$  is sampled independently and uniformly at random from a set or distribution  $\mathcal{X}$ . The output  $x$  of a deterministic algorithm  $\mathcal{A}$  is denoted by  $x = \mathcal{A}$  and the output  $x'$  of a randomized algorithm  $\mathcal{A}'$  is denoted by  $x' \leftarrow \mathcal{A}'$ . For  $a \in \mathbb{N}$  such that  $a \geq 1$ , we denote by  $[a]$  the set of integers lying between 1 and  $a$  (both inclusive). We refer to  $\lambda \in \mathbb{N}$  as the security parameter and denote by  $\text{poly}(\lambda)$  and  $\text{negl}(\lambda)$

any generic (unspecified) polynomial function and negligible function in  $\lambda$ , respectively.  $\text{Inv}(\cdot)$  represents the inverse function of the group element on which it is applied.

## 2.2 Cryptographic Definitions

Here, we define some important cryptographic primitives, which we will use later to build the proposed scheme.

### 2.2.1 Function Secret Sharing.

FSS [BGI15] provides a method to split function  $f$  into separate keys, where each key enables a party to efficiently generate a standard secret share of the evaluation  $f(x)$ , and yet each key individually does not reveal information about which function  $f$  has been shared. The FSS schemes can have multiple variants as per the underlying procedure of recovering  $f(x)$  from the parties' computed share. In the seminal work [BGI15], they defined *decoder* as a function that is employed to perform aggregation over the partial outputs received from each party and produce  $f(x)$ . Similarly, we define a non-linear decoder for the FSS proposed in this paper as this decoder also perform a comparison (the non-linear operation) after aggregating the partial computations from each party. Unlike the decoder in [BGI15], our decoder uses some public parameters as input, which are utilized after aggregation to determine the function output  $f(x)$ . We define our non-linear decoder as follows:

**Definition 1** (Non-linear Output Decoder). A  $p$ -party non-linear output decoder DEC is a tuple  $(Y_1, \dots, Y_p, \text{PP}, R, \text{Dec})$  specifying: share space vectors  $Y_1, \dots, Y_p$  for all  $p$  parties; public parameter PP; output space  $R$ ; and a decoder function  $\text{Dec} : (Y_1 \times \dots \times Y_p, \text{PP}) \rightarrow R$  taking  $p$  parties' shares to an output.

**Definition 2** (Function Secret Sharing (FSS)). For  $p \in \mathbb{N}$ , let  $P = \{P_1, \dots, P_p\}$  denote the set of  $p$ -parties, and  $P' = \{P_1, \dots, P_{p'}\}, 1 \leq p' < p$ , denotes the invalid set of parties with cardinality less than  $p$  for an FSS scheme with respect to non-linear output decoder  $\text{DEC} = (Y_1, \dots, Y_p, \text{PP}, R, \text{Dec})$  and function class  $\mathcal{F}$  (defined over domain  $D$  and range  $R$ ) is a pair of *PPT* algorithms  $(\text{Gen}, \text{Eval})$  with the following syntax:

- $\text{Gen}(1^\lambda, f)$  : Taking input as security parameter  $1^\lambda$  and function description  $f \in \mathcal{F}$ , the key generation algorithm outputs  $p$  keys,  $(k_1, \dots, k_p)$  and a public parameter PP.
- $\text{Eval}(i, k_i, x)$  : Taking input as index  $i$ , key  $k_i$  (each  $k_i$  is assumed to encode input and output domains  $D, R$  of the shared function), and input  $x \in D$ , the evaluation algorithm outputs a value  $y_i \in Y_i$ , corresponding to  $i^{\text{th}}$  party's share of  $f(x)$ .

satisfying the following correctness and security requirements:

- **Correctness:** For all  $f \in \mathcal{F}, x \in D$ ,

$$\Pr \left[ (\{k_1, \dots, k_p\}, \text{PP}) \stackrel{R}{\leftarrow} \text{Gen}(1^\lambda, f) : \right. \\ \left. \text{Dec}(\{\text{Eval}(1, k_1, x), \dots, \text{Eval}(p, k_p, x)\}, \text{PP}) = f(x) \right] = 1.$$

- **Security:** Consider the following indistinguishability challenge experiment for corrupted  $P'$ :
  1. The adversary outputs  $(f_0, f_1) \leftarrow \mathcal{A}(1^\lambda)$ , where  $f_0, f_1 \in \mathcal{F}$  on domain  $D$ .
  2. The challenger samples  $b \leftarrow \{0, 1\}$  and  $(\{k_1, \dots, k_p\}, \text{PP}) \leftarrow \text{Gen}(1^\lambda, f_b)$ .

3. The adversary outputs a guess  $b' \leftarrow \mathcal{A}(\{k_i\}_{P_i \in P'}, \text{PP})$ , given the keys for corrupted  $P'$ .

Let  $\text{Adv}(1^\lambda, \mathcal{A}) := \Pr[b = b'] - 1/2$  denotes the advantage of adversary  $\mathcal{A}$  in guessing  $b$  in the above experiment, where probability is taken over the randomness of the challenger and of  $\mathcal{A}$ . We say the FSS scheme defined above is  $(T, \epsilon)$ -secure, if there exist a negligible function  $\text{negl}$  such that for all non-uniform PPT adversary  $\mathcal{A}$ , it holds that  $\text{Adv}(1^\lambda, \mathcal{A}) \leq \epsilon$  where  $\epsilon = \text{negl}(\lambda)$  and  $T = \text{poly}(\lambda)$  is the polynomial running time.

In the above definition of FSS with a non-linear decoder, the function parameter  $\alpha$  for a point function  $\mathcal{F}_{\alpha, \beta}$  might be leaked as an attacker can retrieve some bits of  $\alpha$  after aggregating the partial computations of parties. To avoid any function parameter leakage, we assume the existence of a trusted decoder in our construction, which can perform minimal aggregation and comparison operation. In this setting, besides partial computation, each party is also provided with the shares of the output mask. More formally, we represent the *offset* (shifted) version of function  $f$  as  $g_{r_{in}, r_{out}}(x) = g(x - r_{in}) + r_{out}$ , where  $r_{in}$  is input mask and  $r_{out}$  is output mask. The decoder keeps  $r_{out}$  with itself while all  $p$  parties are given the shares of  $r_{out}$  to mask their partial outputs. We provide the formal definition of decoder and FSS in the shifted version below.

**Definition 3** (Non-linear Output Decoder (shifted version)). A  $p$ -party non-linear output decoder DEC is a tuple  $(Y_1, \dots, Y_p, \text{PP}, \mathcal{R}, \text{R}, \text{Dec})$  specifying: share space vectors  $Y_1, \dots, Y_p$  for all  $p$  parties; public parameter PP; share space of output mask  $\mathcal{R}$ , output space of offset function as  $\text{R}$ ; and a decoder function  $\text{Dec} : (Y_1 \times \dots \times Y_p, \text{PP}, \mathcal{R}) \rightarrow \text{R}$  taking  $p$  parties' partial computation to an output of the offset function.

**Definition 4** (Function Secret Sharing (shifted version)). Let  $P = \{P_1, \dots, P_p\}$ ,  $p \in \mathbb{N}$  denotes the set of  $p$ -parties, and  $P' = \{P_1, \dots, P_{p'}\}$ ,  $1 \leq p' < p$ , denotes an invalid set of parties. An FSS scheme with respect to a trusted non-linear output decoder  $\text{DEC} = (Y_1, \dots, Y_p, \text{PP}, \mathcal{R}, \text{R}, \text{Dec})$  and *offset* function class  $\mathcal{F}$  (defined over domain  $D$  and range  $\text{R}$ ) with input mask  $r_{in} \in D$  and output mask  $r_{out} \in \mathcal{R}$  is a pair of PPT algorithms  $(\text{Gen}, \text{Eval})$  with the following syntax:

- $\text{Gen}(1^\lambda, g_{r_{in}, r_{out}})$ : Taking input as security parameter  $1^\lambda$  and function description  $g_{r_{in}, r_{out}} \in \mathcal{F}$ , the key generation algorithm outputs  $p$  keys,  $(k_1, \dots, k_p)$ , a public parameter PP and  $p$  additive shares of  $r_{out}$ .
- $\text{Eval}(i, k_i, x_m, [r_{out}]_i)$ : Taking input as index  $i$ , key  $k_i$ , and masked input string  $x_m = x + r_{in}$ , the output mask share  $[r_{out}]_i$  corresponding to party  $P_i$ , the evaluation algorithm outputs a value  $y_i \in Y_i$ , corresponding to  $i^{\text{th}}$  party's share of  $g_{r_{in}, r_{out}}(x_m)$ .

This definition should satisfy the following correctness and security:

- **Correctness:** For all  $g_{r_{in}, r_{out}} \in \mathcal{F}$ ,  $x_m \in D$ ,

$$\Pr \left[ (\{k_1, \dots, k_p\}, \text{PP}, [r_{out}]_{i \in [p]}) \stackrel{\text{R}}{\leftarrow} \text{Gen}(1^\lambda, g_{r_{in}, r_{out}}) : \right. \\ \left. \text{Dec}(\{ \text{Eval}(1, k_1, x_m, [r_{out}]_i) \}_{i \in [p]}, \text{PP}, r_{out}) = g_{r_{in}, r_{out}}(x_m) \right] = 1.$$

- **Security:** Consider the following indistinguishability challenge experiment for corrupted  $P'$ :
  1. The adversary outputs  $(g_0, g_1) \leftarrow \mathcal{A}(1^\lambda)$ , where  $g_0, g_1 \in \mathcal{F}$  on domain  $D$ .
  2. The challenger samples  $b \leftarrow \{0, 1\}$  and  $(\{k_1, \dots, k_p\}, \text{PP}, [r_{out}]_{i \in [p]}) \leftarrow \text{Gen}(1^\lambda, g_b)$ .
  3. The adversary outputs a guess  $b' \leftarrow \mathcal{A}(\{k_i\}_{P_i \in P'}, \text{PP}, [r_{out}]_{i \in [p']})$ , given the keys for corrupted  $P'$ .

Let  $\mathbf{Adv}(1^\lambda, \mathcal{A}) := \Pr[b = b'] - 1/2$  denotes the advantage of adversary  $\mathcal{A}$  in guessing  $b$  in the above experiment, where probability is taken over the randomness of the challenger and of  $\mathcal{A}$ . We say the FSS scheme defined above is  $(T, \epsilon)$ -secure, if there exists a negligible function  $\mathit{negl}$  such that for all non-uniform PPT adversary  $\mathcal{A}$ , it holds that  $\mathbf{Adv}(1^\lambda, \mathcal{A}) \leq \epsilon$  where  $\epsilon = \mathit{negl}(\lambda)$  and  $T = \mathit{poly}(\lambda)$  is the polynomial running time.

*Remark 1.* The above FSS definition with a trusted decoder does not leak any information about function parameters in the sense that the bare aggregation of partial computation does not leak any information about function parameters. Since a dedicated trusted decoder retains  $r_{out}$  with it, the final aggregation without decoder output does not result in the evaluation of the function  $g_{r_{in}, r_{out}}(x_m)$  and an adversary cannot exploit partial computations of parties to leak information about function parameter.

### 2.2.2 Seed Homomorphic Pseudorandom Generator.

We stem the seed homomorphic pseudorandom generator concept from [BLMR13] as follows.

**Definition 5** (Seed Homomorphic PRGs). An efficiently computable function  $\mathcal{G} : \mathcal{X} \rightarrow \mathcal{Y}$ , where  $(\mathcal{X}, \oplus)$  and  $(\mathcal{Y}, \otimes)$  are groups, is said to be seed homomorphic PRG if the following two properties hold:

- $\mathcal{G}$  is a secure PRG.
- For every  $s_1, s_2 \in \mathcal{X}$ , we have  $\mathcal{G}(s_1) \otimes \mathcal{G}(s_2) = \mathcal{G}(s_1 \oplus s_2)$

**Example 1.** Let  $\mathbb{G}$  be an elliptic curve group of order  $q$  in which ECDH assumptions hold. For a group  $(S, \oplus)$ , consider a PRG  $\mathcal{G}_{\text{SHPRG}} : S \rightarrow \mathbb{G} \times \mathbb{G}$  with group generators  $gg = (g, h)$  where  $g$  and  $h$  are uniformly chosen generators of  $\mathbb{G}$  during the setup phase. The output of  $\mathcal{G}_{\text{SHPRG}}$  with parameter  $gg$  and seed  $s$  is defined as,  $\mathcal{G}_{\text{SHPRG}}(s) = (g^s, h^s)$ . Security of  $\mathcal{G}_{\text{SHPRG}}$  follows immediately from the ECDH assumptions: when  $s$  is uniformly taken from  $S$ , then  $\mathcal{G}_{\text{SHPRG}}(s)$  is indistinguishable from a random sample in  $\mathbb{G} \times \mathbb{G}$ . The homomorphic properties hold as follows.

$$\mathcal{G}_{\text{SHPRG}}(s_1 \oplus s_2) = \mathcal{G}_{\text{SHPRG}}(s_1) \otimes \mathcal{G}_{\text{SHPRG}}(s_2)$$

**Definition 6.** ( $\mu$ -stretchable Seed Homomorphic PRGs). An efficiently computable seed homomorphic PRG  $\mathcal{G}$ , is said to be  $\mu$ -stretchable seed homomorphic when the output of  $\mathcal{G}$  is stretched by  $\mu$  generators from  $\mathbb{G}$ .

In example 1, the given *seed homomorphic* PRG is  $\mu = 2$  stretchable as it uses 2 generators  $g$  and  $h$  to scale the output.

## 3 FSS with Compact Key-Size

In this section, we present a novel  $p$ -party FSS construction for different classes of functions like point function, comparison function, interval function, correctness proof, and security proof, followed by a comparison with state-of-the-art literature.



### 3.1 Proposed Distributed Point Function

We present a  $p$ -party distributed point function scheme using seed-homomorphic PRGs and additive secret sharing as the building blocks for our construction. A  $p$ -party DPF construction can be given as follows:

**Construction 1** ( $p$ -party Distributed Point Function). *A  $p$ -party distributed point function DPF is a tuple of two probabilistic polynomial time algorithms (DPF.Gen, DPF.Eval) with set of parties  $P = \{P_1, \dots, P_p\}$  and a trusted output decoder DEC to combine the partial evaluation from parties and decode the final output.*

A dealer takes a security parameter  $\lambda$  as input. It randomly chooses input mask,  $r_{in}$  and output mask,  $r_{out}$  of the point function  $\mathcal{F}_{\alpha,\beta} : \{0, 1\}^n \rightarrow \mathbb{G}$  with  $n$ -bit inputs and produces output in group  $\mathbb{G}$  (*output group*) of order  $2^m$ , for some integer  $m$ . It also establishes a seed homomorphic pseudorandom generator (SHPRG) and sets various parameters such as  $gg$ ,  $q$ ,  $p$ ,  $n$ , and  $m$ , which we will explain below. Let us consider a seed homomorphic pseudorandom generator  $\mathcal{G}_{\text{SHPRG}} : S \rightarrow \mathbb{G}^\mu$ , which takes an input seed from group  $S$  and outputs a  $\mu$ -sized vector with elements in group  $\mathbb{G}$  where

- $\mathbb{G}$  is an *output group* defined on the range of function  $\mathcal{F}_{\alpha,\beta}$ .
- $(S, \oplus)$  is a group over integers modulo prime  $q$ .
- $\lambda \approx \log(q)$
- $gg$  is a randomly chosen set of  $\mu$  generators from  $\mathbb{G}$  represented as  $gg = (g_1, g_2, \dots, g_\mu)$  with  $\mu$  as a scaling factor of SHPRG.
- $p$  is the number of parties,
- $n$  and  $m = \log_2(|\mathbb{G}|)$  denote the domain and range size in bits for the class of point function  $\mathcal{F}_{\alpha,\beta}$ .

1. DPF.Gen( $\lambda, \mathcal{F}_{\alpha,\beta}$ ): This algorithm takes a function with parameters  $\alpha$ ,  $\beta$ , and a security parameter  $\lambda$  as inputs and generates keys for all parties. The input  $\alpha$  is written down as a pair  $(\gamma, \delta)$ , where  $\gamma \in [\nu]$ ,  $\delta \in [\mu]$  with  $\mu \leftarrow \lceil 2^{n/2} \cdot 2^{(p-1)/2} \rceil$  and  $\nu \leftarrow \lceil 2^n / \mu \rceil$  (we follow the splitting of  $\nu$  and  $\mu$  from [BGI15]). Here,  $\nu$  and  $\mu$  represent the evaluation table's rows and columns. Note that  $(\gamma, \delta)$  locate the position of  $\beta$  in the evaluation table.

- (a) **Generation of Correction Word:** Let us write the function output  $\beta$  as  $\beta_{\mathbb{G}}$ . Randomly select a seed  $s_\gamma \in S$  corresponding to row  $\gamma$ . For  $\gamma' \in [\nu]$  whenever  $\gamma' = \gamma$ , generate correction word such that  $\forall \delta' \in [\mu]$ ,

$$\text{cw}_{\delta'} = \begin{cases} \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{(\delta)}(s_\gamma)), & \delta' = \delta \\ 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{(\delta')}(s_\gamma)), & \delta' \neq \delta \end{cases}$$

where  $\text{CW} = (\text{cw}_1, \text{cw}_2, \dots, \text{cw}_\mu)$ ,  $1_{\mathbb{G}}$  is the identity element of the *output group*  $(\mathbb{G}, \otimes)$ . In our current definition of FSS, we consider  $\text{CW}$  as a public parameter  $\text{PP}$  and make it public.

- (b) **Generation of Shares:** To generate the shares of the function for parties, the main idea is to generate the shares of  $0_S$  for all rows not belonging to the target row and to generate the shares of  $s_\gamma$  corresponding to the target row  $\gamma$ . Note that  $s_\gamma$  is a randomly chosen seed that has been used for the formation of  $\text{CW}$ . To generate the secret shares of  $0_S$  (for all non-target rows, i.e  $\gamma' \neq \gamma$ ) one can pick  $p - 1$  random shares  $\text{sh}_{i,\gamma'} \in S, i \in [1, p - 1]$ , and set  $\text{sh}_{p,\gamma'} = \text{Inv}(\bigoplus_{i=1}^{p-1} \text{sh}_{i,\gamma'})$ ,  $\forall \gamma' \in [\nu]$ . For target row  $\gamma$ , one can randomly generate shares of  $p - 1$  parties as  $\text{sh}_{i,\gamma} \in S, \forall i \in [1, p - 1]$  and use seed  $s_\gamma$  to compute the share of  $p$ -th party as follows,

$$\text{sh}_{p,\gamma} = s_\gamma \oplus \text{Inv}\left(\bigoplus_{i=1}^{p-1} \text{sh}_{i,\gamma}\right)$$

- (c) The dealer also generates  $p$  additive shares of output mask  $r_{out} \in \mathbb{G}$  for all the parties as  $([r_{out}]_1, \dots, [r_{out}]_p)$ . It sends  $r_{out}$  to the trusted decoder while  $[r_{out}]_i$  to party  $P_i$  as a part of their FSS key.
- (d) Send key  $K_i = ([r_{out}]_i, \{\text{sh}_{i,\gamma'}\}_{\forall \gamma' \in [\nu]})$  to party  $P_i$  and release PP = CW in public domain.
2. **DPF.Eval**( $i, K_i, x$ ): A party  $P_i$  uses this algorithm to evaluate the partial computation of the function using  $K_i$  and  $x$ . The input  $x$  is written down as a pair  $(\gamma^*, \delta^*)$ , where  $\gamma^* \in [\nu]$ ,  $\delta^* \in [\mu]$  with  $\mu \leftarrow \lceil 2^{n/2} \cdot 2^{(p-1)/2} \rceil$  and  $\nu \leftarrow \lceil 2^n / \mu \rceil$ . Parse  $K_i = ([r_{out}]_i, \{\text{sh}_{i,\gamma'}\}_{\forall \gamma' \in [\nu]})$ . Then, party  $P_i$  will use its seed share corresponding to row  $\gamma^*$  and use the following equation to calculate the partial computation.

$$\text{PartComp}_i = \mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(\text{sh}_{i,\gamma^*}) \otimes [r_{out}]_i$$

Note that  $\mu$  is the scaling factor of  $\mathcal{G}_{\text{SHPRG}}(\cdot)$  and hence it outputs  $\mu$ -sized vector with each element in group  $(\mathbb{G}, \otimes)$ . Here, we use the notation  $\mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(\cdot)$  to represent the  $\delta^*$ -th element of the vector. Each party computes  $Y_i = \text{PartComp}_i \in \mathbb{G}$  which the decoder can use for revealing the function output as given in the following description.

**Theorem 1.** *Suppose  $\mathcal{G} : S \rightarrow \mathbb{G}^\mu$  is a seed homomorphic pseudorandom generator. Our scheme  $\text{DPF} = (\text{DPF.Gen}, \text{DPF.Eval})$  is a correct and secure  $p$ -party distributed point function for the family of point function  $\mathcal{F}_{\alpha,\beta}$  with shared secret size  $2^{(n-p+1)/2}(\lambda) + m$  bit, and public parameter of size  $2^{(n+p-1)/2}(m)$  bit where  $p$  is the total number of parties.*

*Proof.* We prove this theorem by proving correctness (Claim 3.1) and security (Claim 3.1) of the proposed DPF scheme below.

**Claim** (Correctness). *Our  $p$ -party DPF scheme is correct for point function  $\mathcal{F}_{\alpha,\beta}$ , for a masked input  $x$  and for party  $P_i \in P$  for a specific output decoder (ref. Definition 1) function  $\text{Dec}^\otimes : (Y_1 \otimes \dots \otimes Y_p, \text{PP}, \mathcal{R}) \rightarrow R$ , with  $Y_i \in \mathbb{G}$  and  $R \in \mathbb{G}$ , it follows,*

$$\Pr\left[\left(\{k_1, \dots, k_p\}, \text{PP}, [r_{out}]_{i \in [p]}\right) \stackrel{R}{\leftarrow} \text{DPF.Gen}(1^\lambda, \mathcal{F}_{\alpha,\beta}) : \right. \\ \left. \text{Dec}^\otimes(\{\text{DPF.Eval}(i, k_i, x, [r_{out}]_i)\}_{i \in [p]}, \text{PP}, r_{out}) = \mathcal{F}_{\alpha,\beta}(x)\right] = 1.$$

**Proof 1.** Each party  $P_i \in P$  has the key of the form,  $K_i = \{\text{sh}_{i,\gamma'}\}_{\forall \gamma' \in [\nu]}$ , using which it partially computes the function and outputs  $Y_i = \text{PartComp}_i$ . A decoder  $\text{Dec}^\otimes$  takes all parties' public parameters PP = CW and  $\text{PartComp}_i$  as inputs. It then picks  $\text{cw}_{\delta^*}$  from CW and finally produces Res after performing *output group* operation as follows,

$$\begin{aligned} \text{Res} &= \text{Dec}^\otimes(\{Y_1, \dots, Y_p\}, \text{CW}, r_{out}) = \text{Inv}(r_{out}) \otimes \text{cw}_{\delta^*} \otimes \bigotimes_{i=1}^p Y_i \\ &= \text{Inv}(r_{out}) \otimes \text{cw}_{\delta^*} \otimes r_{out} \otimes \bigotimes_{i=1}^p \mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(\text{sh}_{i,\gamma^*}) = \text{cw}_{\delta^*} \otimes \mathcal{G}_{\text{SHPRG}}^{(\delta^*)}\left(\bigoplus_{i=1}^p \text{sh}_{i,\gamma^*}\right) \end{aligned}$$

The decoder has nullified the effect of  $r_{out}$  from the result Res by operating with the inverse of  $r_{out}$  and processes the decoding of the DPF scheme for the following cases.

*Case 1.* ( $x \neq \alpha$ ). As  $x = (\gamma^*, \delta^*)$  and  $\alpha = (\gamma, \delta)$ , it covers the case when atleast the row or column is not a target row/column. For  $x \neq \alpha$  raises two different possibilities mentioned below:

1. When  $\gamma^* \neq \gamma$ , the reconstruction operation  $\bigoplus_{i=1}^p \text{sh}_{i,\gamma^*}$  yields 0 (recall that during DPF.Gen, for  $\gamma^* \neq \gamma$ , the shares for parties are generated such that it adds to 0). Hence,  $\text{Res} = \text{cw}_{\delta^*} \otimes \mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(0) = \text{cw}_{\delta^*}$ . The decoder interprets the result as  $1_{\mathbb{G}}$  whenever  $\text{Res} = \text{cw}_{\delta^*}$ .
2. When  $\gamma^* = \gamma$  but  $\delta^* \neq \delta$ , the reconstruction operation  $\bigoplus_{i=1}^p \text{sh}_{i,\gamma^*}$  yields  $s_\gamma$  (recall that during DPF.Gen, for  $\gamma = \gamma^*$ , the additive shares of  $s_\gamma$  were generated). Hence,  $\text{Res} = \text{cw}_{\delta^*} \otimes \mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(s_\gamma) = 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(s_\gamma)) \otimes \mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(s_\gamma) = 1_{\mathbb{G}}$ .

*Case 2.* ( $x = \alpha$ ). As we have  $x = \alpha$  so we have  $\gamma^* = \gamma$  and  $\delta^* = \delta$  and this covers the case when the row  $\gamma^*$  is a target row and column  $\delta^*$  is also a target column. So, the reconstruction operation  $\bigoplus_{i=1}^p \text{sh}_{i,\gamma^*}$  yields  $s_\gamma$  (recall from DPF.Gen, for  $\gamma = \gamma^*$ , the shares of  $s_\gamma$  were generated). Hence,  $\text{Res} = \text{cw}_\delta \otimes \mathcal{G}_{\text{SHPRG}}^{(\delta)}(s_\gamma) = \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{(\delta)}(s_\gamma)) \otimes \mathcal{G}_{\text{SHPRG}}^{(\delta)}(s_\gamma) = \beta_{\mathbb{G}}$ .

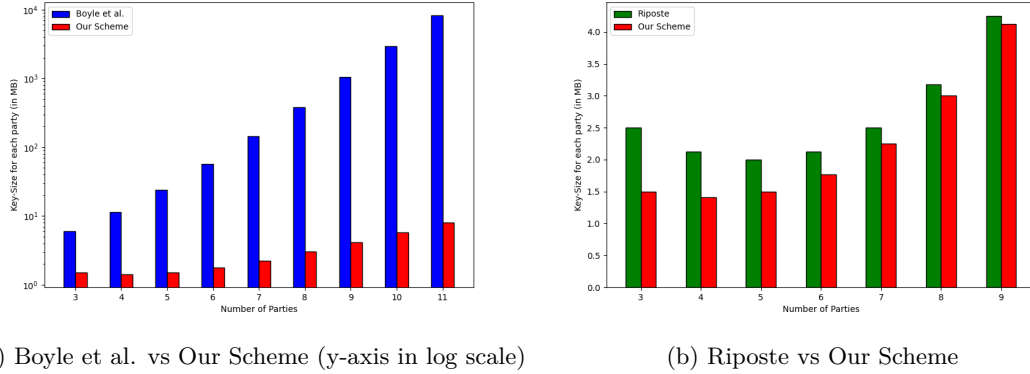
**Claim (Security).** *For any polynomial  $p(n) \in \text{poly}(n)$  such that, given an additive Secret Sharing Scheme (SSS) defined over group  $(S, \oplus)$  and a seed homomorphic PRG compatible with SSS, the scheme (DPF.Gen, DPF.Eval) is a  $(T', \epsilon')$ -secure DPF scheme for  $T' = T_{(\text{SHPRG}+\text{SSS})} - p(n)$  and  $\epsilon' = \epsilon_{\text{SSS}} + 2\epsilon_{\text{SHPRG}}$  where  $\epsilon_{\text{SSS}}$  and  $\epsilon_{\text{SHPRG}}$  are the winning advantage of an adversary in SSS and SHPRG respectively.*

**Proof 2** (Proof overview). Recall from Section 3.1, the key  $K_i = ([r_{out}]_i, \{\text{sh}_{i,\gamma'}\}_{\forall \gamma' \in [\nu]})$  is privately shared with party  $P_i$  and public parameter CW is generated using SHPRG. At a high level, we will show that the correction word  $\text{CW} = (\text{cw}_1, \text{cw}_2, \dots, \text{cw}_\mu)$  are *pseudorandom* given the remaining view of every party. Thus, we will have the indistinguishability of keys generated for two different functions for some corrupted set of parties  $P'$  with  $|P'| < p$ . Formally, for any pair of functions  $\mathcal{F}_{\alpha,\beta}$  and  $\mathcal{F}_{\hat{\alpha},\hat{\beta}}$ , we consider a sequence of hybrid distributions that begin with an honestly generated DPF key for  $\mathcal{F}_{\alpha,\beta}$  and end with an honestly generated DPF key for  $\mathcal{F}_{\hat{\alpha},\hat{\beta}}$ . Note that the adversary has access to the keys for two functions  $\mathcal{F}_{\alpha,\beta}$  and  $\mathcal{F}_{\hat{\alpha},\hat{\beta}}$ , generated only for a corrupted set of parties  $P'$ . We aim to show that an adversary who wins in the DPF security game with an advantage greater than  $\epsilon'$  must succeed in distinguishing between the key distributions of  $\mathcal{F}_{\alpha,\beta}$  and  $\mathcal{F}_{\hat{\alpha},\hat{\beta}}$ , and thus distinguishes between the adjacent hybrids with the advantage that contradicts the security of one of the underlying tools. The underlying tool is the SHPRG and secret sharing scheme to construct DPF. We formally prove the security by defining four sets of hybrids in Appendix A.

Now, we describe the size of public parameter (correction word) and secret shares as mentioned in Theorem 1. The correction word CW is a column ( $\mu$ ) length vector with each element in  $\mathbb{G}$ . Considering  $m = \log_2(|\mathbb{G}|)$  and  $\mu = \lceil 2^{n/2} \cdot 2^{(p-1)/2} \rceil$ , one can calculate  $m\mu = 2^{(n+p-1)/2}(m)$  bit. Similarly, the size of secret shares is due to the share size for each row of the evaluation table and the size of  $r_{out}$  share. Considering  $\nu = \lceil 2^n/\mu \rceil$  rows in our evaluation table, the length of each seed share as  $\lambda$ -bit, and the length of  $r_{out}$  share as  $m$ -bit, the total secret shares size is  $\lambda\nu + m$  which is equal to  $2^{(n-p+1)/2}(\lambda) + m$  bits.

### 3.2 Comparing $p$ -party DPF with state-of-the-art

The  $p$ -party variant of our proposed DPF overlaps with the  $p$ -party version of [BGI15] and [CGBM15] yet keeping a direct improvement in key-size by a factor of  $O(2^p)$  in [BGI15] and has reduced the key-size by  $2^{n/2}2^{-p}(\lambda)$ -bits in [CGBM15]. To achieve  $p$ -party DPF, we used additive secret sharing, which satisfies the reconstruction by all the involved parties.



(a) Boyle et al. vs Our Scheme (y-axis in log scale)

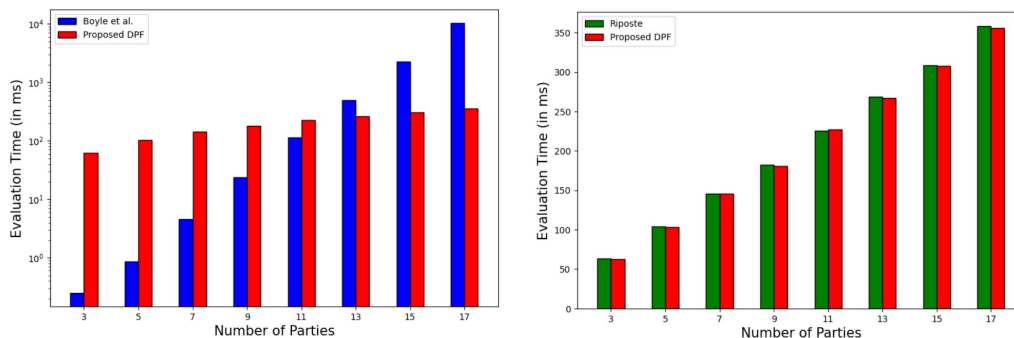
(b) Riposte vs Our Scheme

Figure 2: Key size comparison with other schemes

Besides improvement in key size, we also reduce the number of PRG invocations during DPF.Eval with respect to PRG invocations in [BGI15]. Each party in [BGI15] performs  $2^{p-1}$  PRG invocation, which in our construction has been reduced to only one invocation by each party. Although we use seed homomorphic PRG (public-key setting) whose invocation is a bit more expensive than the (symmetric-key setting) PRG, the  $2^{p-1}$  number of PRG invocations by each party proves to be computationally more expensive than our use of seed-homomorphic PRG when the number of parties crosses 12 (refer to Figure 3). Figure 3 shows the comparative evaluation time of our scheme and [BGI15].

We choose NIST P-256 elliptic curve with modulo  $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$  satisfying  $y^2 = x^3 - 3x + 41058363725152142129326129780047268409114441015993725554835256314039467401291$ , for our prototype implementation but other elliptic curves also perform well in this setting. We used Montgomery ladder-based scalar multiplication and point addition to perform operations during Gen and Eval. We employ seed homomorphic PRG based on the ECDH assumption for the above curve to support 128-bit security for our proposed implementation. The implementation results have been recorded on the system equipped with an Intel® Core™ i5-9500 CPU and 3.00GHz clock frequency, with 20GB of RAM. We use the standard double precision IEEE 754 floating point conversion for the floating point numbers to convert all the floating point numbers to 64bit binary numbers. However, our implementation is not limited to 64-bit inputs subjected to the availability of appropriate floating point standard representation.

We show some comparative experimental results in Figure 2a, 2b, and Figure 3 to justify the use of seed homomorphic PRG when the number of parties increases linearly. Figure 2a shows a key-size comparison between Boyle et al.[BGI15] and our proposed scheme in logarithmic scale. Figure 2b shows the key-size comparison between Riposte and our proposed scheme. Figure 3 shows the plot between the evaluation time of Boyle et al.[BGI15] and our scheme on a logarithmic scale. The exponential blow-up in the evaluation time of Boyle et al.[BGI15] is due to exponential PRG invocations with a linear increase in the number of parties. When comparing our evaluation time with Riposte[CGBM15], we have improvements with one multiplication time. However, the seed-homomorphic PRG computation dominates the evaluation time; as a result, the seed-homomorphic PRG computation largely suppresses our improvement of one multiplication time as compared to Riposte. Figure 2a plot shows a very similar trend as Figure 3 stating that the key corresponding to each row of Boyle et al.[BGI15] has  $2^{p-1}$  seeds leading to an exponential blow up in key-size for Boyle et al.[BGI15]. The key size in our scheme has drastically reduced as each party's seed share to be stored per row is only  $\lambda$ -bits. Deviating from the row and column split used in [BGI15], we keep the row and column size the same,



(a) Boyle et al. vs Our Scheme(y-axis in log scale)

(b) Riposte vs Our Scheme

Figure 3: Evaluation time comparison with other schemes

i.e.,  $2^{n/2}$ . Also we use  $n = 32$ ,  $m = 32$ , and  $\lambda = 256$  for all the measurements.

**Extension to other classes of function.** In the following section, we extend our compact  $p$ -party DPF construction to other functions, specifically distributed comparison and interval functions. To our knowledge, the proposed distributed comparison and interval functions for  $p$ -parties have the most efficient key size to date.

### 3.3 Distributed Comparison Function (DCF)

On a high level, the comparison function can be considered a particular multi-point function that outputs  $\beta$  at more than one point (instead, it outputs  $\beta$  continuously after a specific point). To recall, the evaluation table of the comparison function  $f_{1001,\beta}^<$  (ref. Table 4) has three types of rows: 1) all entries  $1_{\mathbb{G}}$ ; 2) mixed  $1_{\mathbb{G}}$  and  $\beta_{\mathbb{G}}$  entries; and 3) all entries as  $\beta_{\mathbb{G}}$ . The idea is to use only three seeds repeatedly, like  $s_{\gamma_1}$  for the first type of rows,  $s_{\gamma_2}$  for the second type, and  $s_{\gamma_3}$  for the third type of rows, and secret share these seeds independently with all the parties. Since there are three different seed values, we need to generate 3 CW here and release them as public parameters. Note that 3 CW produced during key generation remains indistinguishable from any randomly generated binary string of length  $3m\mu$ .

**Construction 2** ( $p$ -party Distributed Comparison Function). *A  $p$ -party distributed comparison function DCF is a tuple of two PPT algorithms (DCF.Gen, DCF.Eval) with a set of parties  $P = \{P_1, \dots, P_p\}$  and an output decoder DEC to combine the partial evaluations of the parties and decode the final output.*

Taking security parameter  $\lambda$  as input, a dealer chooses  $q \in \mathbb{N}$ ,  $n \in \mathbb{N}$ ,  $m \in \mathbb{N}$ ,  $\mathbb{PP} \in \mathbb{G}^\mu$  ( $\mu$  being the number of columns in the evaluation table), decides on the description of the comparison function  $\mathcal{F}_{\alpha,\beta}^<$  and seed homomorphic pseudorandom generator SHPRG. It randomly chooses input mask,  $r_{in}$  and output mask,  $r_{out}$  of the comparison function  $\mathcal{F}_{\alpha,\beta}^< : \{0, 1\}^n \rightarrow \mathbb{G}$  where  $\mathbb{G}$  is an output group. Consider a  $\mu$ -stretchable seed homomorphic pseudorandom generator as  $\mathcal{G}_{\text{SHPRG}} : S \rightarrow \mathbb{G}^\mu$ , where symbols have their usual meaning as defined in Section 3.1.

1. DCF.Gen( $\lambda, \mathcal{F}_{\alpha,\beta}^<$ ): In this algorithm, the input  $\alpha$  to the comparison function  $\mathcal{F}_{\alpha,\beta}^<$  is written down as a pair  $(\gamma, \delta)$ , where  $\gamma \in [\nu]$ ,  $\delta \in [\mu]$  with  $\mu \leftarrow \lceil 2^{n/2} \cdot 2^{(p-1)/2} \rceil$  and  $\nu \leftarrow \lceil 2^n / \mu \rceil$ .

- (a) Write the function output  $\beta$  as  $\beta_{\mathbb{G}}$  randomly choose 3 seeds  $s_{\gamma^1}$ ,  $s_{\gamma^2}$  and  $s_{\gamma^3}$  and randomly initialize  $\{\text{cw}_{\delta'}^1, \text{cw}_{\delta'}^2, \text{cw}_{\delta'}^3\}_{\delta' \in [\mu]}$  with group  $\mathbb{G}$  elements and then overwrite the correction word  $\text{cw}_{\delta'}^1$ ,  $\text{cw}_{\delta'}^2$  and  $\text{cw}_{\delta'}^3$ , as follows, Generate 3 correction words for following 3 cases as, for any  $\gamma' \in [\mu]$ ,

If  $\gamma' < \gamma$ ,

$$\text{cw}_{\delta'}^1 = 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{\delta'}(s_{\gamma^1})), \forall \delta' \in [\mu]$$

If  $\gamma' = \gamma$ ,

$$\text{cw}_{\delta'}^2 = \begin{cases} \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{\delta'}(s_{\gamma^2})), & \delta' \geq \delta \in [\mu] \\ 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{\delta'}(s_{\gamma^2})), & \delta' < \delta \in [\mu] \end{cases}$$

If  $\gamma' > \gamma$ ,

$$\text{cw}_{\delta'}^3 = \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{\delta'}(s_{\gamma^3})), \forall \delta' \in [\mu]$$

where  $\text{CW}^1 = (\text{cw}_1^1, \text{cw}_2^1, \dots, \text{cw}_\mu^1)$ ,  $\text{CW}^2 = (\text{cw}_1^2, \text{cw}_2^2, \dots, \text{cw}_\mu^2)$ , and  $\text{CW}^3 = (\text{cw}_1^3, \text{cw}_2^3, \dots, \text{cw}_\mu^3)$ . It is important to note that random initialization of  $(\text{CW}^1, \text{CW}^2, \text{CW}^3)$  is required in the beginning.

- (b) In the previous step, the 3-seeds  $s_{\gamma^1}$ ,  $s_{\gamma^2}$ , and  $s_{\gamma^3}$  were randomly chosen that need to be distributed across the rows of evaluation table as follows.

$$s_{\gamma'} = \begin{cases} s_{\gamma^1}, & \gamma' < \gamma \\ s_{\gamma^2}, & \gamma' = \gamma \\ s_{\gamma^3}, & \gamma' > \gamma \end{cases}$$

For all rows  $\gamma' < \gamma$  we use the seed  $s_{\gamma^1}$ , for row  $\gamma' = \gamma$  we use  $s_{\gamma^2}$  and similarly for all rows having  $\gamma' > \gamma$  we use the seed  $s_{\gamma^3}$ . Following a similar strategy of seed-sharing, we either generate the shares of  $s_{\gamma'}$  or  $0_S$  depending on the row for which secret shares are generated. One can generate the secret shares of  $0_S$  by picking  $p-1$  random shares  $\text{sh}_{i,\gamma'} \in S, i \in [1, p-1]$ , and  $\text{sh}_{p,\gamma'} = \text{Inv}(\bigoplus_{i=1}^{p-1} \text{sh}_{i,\gamma'}), \forall \gamma' < \gamma$ . For all  $\gamma' \geq \gamma$ , we can randomly generate shares of  $p-1$  parties as  $\text{sh}_{i,\gamma'} \in S, \forall i \in [1, p-1]$  and use seed  $s_{\gamma'}$  to compute share of  $p$ -th party as follows,

$$\text{sh}_{p,\gamma'} = s_{\gamma'} \oplus \text{Inv}\left(\bigoplus_{i=1}^{p-1} \text{sh}_{i,\gamma'}\right)$$

Concretely, for rows ( $< \gamma$ ) we generate shares of  $0_S$ , for row ( $= \gamma$ ), we generate shares of  $s_{\gamma^2}$ , and for rows ( $> \gamma$ ) we generate shares of  $s_{\gamma^3}$  independently.

- (c) The dealer also generates  $p$  additive shares of output mask  $r_{out} \in \mathbb{G}$  for all the parties as  $([r_{out}]_1, \dots, [r_{out}]_p)$ . The output mask  $r_{out}$  is kept with the decoder while  $[r_{out}]_i$  is sent to party  $P_i$ .
- (d) Send key  $K_i = ([r_{out}]_i, \{\text{sh}_{i,\gamma'}\}_{\forall \gamma' \in [\nu]})$  to party  $P_i$  and  $\text{PP} = \{\text{CW}^1 || \text{CW}^2 || \text{CW}^3\}$  is released in public domain.
2.  $\text{DCF.Eval}(i, K_i, x)$ : Each party  $P_i$  evaluate the partial functions using  $K_i$  and  $x$ . The input  $x$  is written down as a pair  $(\gamma^*, \delta^*)$ , where  $\gamma^* \in [\nu]$ ,  $\delta^* \in [\mu]$  with  $\mu \leftarrow \lceil 2^{n/2} \cdot 2^{(p-1)/2} \rceil$  and  $\nu \leftarrow \lceil 2^n / \mu \rceil$ . Parse  $K_i = ([r_{out}]_i, \{\text{sh}_{i,\gamma'}\}_{\forall \gamma' \in [\nu]})$  and  $\text{PP} = \{\text{CW}^1 || \text{CW}^2 || \text{CW}^3\}$ . Then, use the following equation to calculate two partial computations  $\text{PartComp}_{i,1}$  and  $\text{PartComp}_{i,2}$  corresponding to  $\delta^*$  and  $\delta^* \pm 1$ .

$$\text{PartComp}_{i,1} = \mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(\text{sh}_{i,\gamma^*}) \otimes [r_{out}]_i, \quad \text{PartComp}_{i,2} = \mathcal{G}_{\text{SHPRG}}^{(\delta^* \pm 1)}(\text{sh}_{i,\gamma^*}) \otimes [r_{out}]_i$$



The selection of  $\delta^* + 1$  or  $\delta^* - 1$  depends on the availability of valid  $\delta^* \pm 1$  for the row  $\gamma^*$ . Suppose the given  $\delta^*$  is 0, so  $\delta^* + 1$  will be selected. Similarly in another case, if  $\delta^*$  is equal to  $\mu - 1$ , then  $\delta^* - 1$  will be chosen. For case, when  $0 < \delta^* < \mu - 1$ , use  $\delta^* + 1$  though one can also choose  $\delta^* - 1$  through out the construction. Recall  $\mathcal{G}_{\text{SHPRG}}(\cdot)$  outputs a  $\mu$ -sized vector for each element belonging to  $\mathbb{G}$  group, and we used the notation  $\mathcal{G}_{\text{SHPRG}}^{(\delta^*)}(\cdot)$  to represent the  $\delta^*$ -th element of the vector. Each party assigns  $Y_{i,1} = \text{PartComp}_{i,1}$  and  $Y_{i,2} = \text{PartComp}_{i,2}$ , and then sends  $Y_{i,1}, Y_{i,2}$  to the decoder for the final computation of the comparison function.

*Remark 2.* Unlike DPF, two **PartComps** are required for DCF to correctly compute the row that embeds  $\beta$  in it. The  $\delta^* \pm 1$  is considered for cases where  $\beta$  might be embedded at the first or the last column in row  $\gamma^*$ .

**Theorem 2.** *Suppose  $\mathcal{G} : S \rightarrow \mathbb{G}^\mu$  is a seed homomorphic pseudorandom generator. Our scheme  $\text{DCF} = (\text{DCF.Gen}, \text{DCF.Eval})$  is a correct and secure  $p$ -party distributed comparison function for the family of comparison function  $\mathcal{F}_{\alpha,\beta}^<$  with shared secret size  $2^{(n-p+1)/2}(\lambda) + m$  bit, and public parameter of size  $3 \cdot 2^{(n+p-1)/2}(m)$  bit where  $p$  is the total number of parties.*

*Proof.* We prove this theorem by proving correctness (Claim 3.3) and security (Claim 3.3) of the proposed DCF scheme below.

**Claim (Correctness).** *The resulting scheme DCF (Section 3.3) is correct i.e., for any  $p$ -party distributed comparison function  $\mathcal{F}_{\alpha,\beta}^<$ , on masked input  $x$  and for a specific output decoder function  $\text{Dec}^< : (\{Y_{i,1}, Y_{i,2}\}_{i \in [p]}, \text{PP}, \mathcal{R}) \rightarrow R$ , with  $(Y_{i,1}, Y_{i,2}) \in \mathbb{G}^2, \mathcal{R} \in \mathbb{G}, R \in \mathbb{G}$ , it follows,*

$$\Pr \left[ (\{K_i\}_{i \in [p]}, \text{PP}, r_{out}) \stackrel{R}{\leftarrow} \text{DCF.Gen}(1^\lambda, \mathcal{F}_{\alpha,\beta}^<) : \right. \\ \left. \text{Dec}^< (\{\text{DCF.Eval}(i, K_i, x, [r_{out}]_i)\}_{i \in [p]}, \text{PP}, r_{out}) = \mathcal{F}_{\alpha,\beta}^<(x) \right] = 1.$$

**Proof 3.** Each party  $P_i$  possesses key  $K_i$ . Using this key, it partially computes the function and assigns  $Y_{i,1} = \text{PartComp}_{i,1}$  and  $Y_{i,2} = \text{PartComp}_{i,2}$ . The decoder  $\text{Dec}^<$  takes input as  $\{Y_{i,1}, Y_{i,2}\}_{i \in [p]}$ , and  $\text{PP} = \{\text{CW}^1 || \text{CW}^2 || \text{CW}^3\}$  and outputs result as follows:

$$\text{Res}^1 = \text{Inv}(r_{out}) \otimes \text{cw}_{\delta^*}^1 \otimes \bigotimes_{i=1}^p Y_{i,1}$$

The decoder first nullifies the effect of  $r_{out}$  by taking its inverse and operating it with the aggregated value of partial decryption to obtain  $\text{Res}^1$  and checks if  $\text{Res}^1 == \text{cw}_{\delta^*}^1$ , it outputs  $1_{\mathbb{G}}$ ; otherwise it computes,

$$\text{Res}^2 = \text{Inv}(r_{out}) \otimes \text{cw}_{\delta^*}^3 \otimes \bigotimes_{i=1}^p Y_{i,1}, \quad \text{Res}^3 = \text{Inv}(r_{out}) \otimes \text{cw}_{\delta^* \pm 1}^3 \otimes \bigotimes_{i=1}^p Y_{i,2}$$

The decoder nullifies  $r_{out}$  and computes  $\text{Res}^2$  and  $\text{Res}^3$ . If  $\text{Res}^2 == \text{Res}^3$ , it outputs  $\text{Res}^2$  as the final evaluation which is equal to  $\beta_{\mathbb{G}}$ , else it computes

$$\text{Res}^4 = \text{Inv}(r_{out}) \otimes \text{cw}_{\delta^*}^2 \otimes \bigotimes_{i=1}^p Y_{i,1}$$

The decoder outputs  $\text{Res}^4$  as the final output.

**Claim (Security).** *For any polynomial  $p(n) \in \text{poly}(n)$  such that, given an additive Secret Sharing Scheme (SSS) defined over group  $(S, \oplus)$  and a seed homomorphic PRG compatible with SSS, the scheme (DCF.Gen, DCF.Eval) is a  $(T', \epsilon')$ -secure DCF scheme for  $T' = T_{(\text{SHPRG} + \text{SSS})} - p(n)$  and  $\epsilon' = \epsilon_{\text{SSS}} + 2\epsilon_{\text{SHPRG}}$  where  $\epsilon_{\text{SSS}}$  and  $\epsilon_{\text{SHPRG}}$  are the winning advantage of an adversary in SSS and SHPRG respectively.*

**Proof 4.** The security proof for the proposed DCF is implied by the security of the DPF. Hence, we skip the formal proof for DCF and discuss certain *corner cases* to argue that our scheme leaks no information about the function.

For the corner cases like when  $\gamma$  is 0 or  $\nu - 1$  if  $\text{CW}^1$  or  $\text{CW}^3$  remains uninitialized, it might leak information about  $\alpha$ . To handle such cases, we initially assign the random values to  $\text{CW}^1$ ,  $\text{CW}^2$ , and  $\text{CW}^3$ , which will get overwritten depending on the value of  $\alpha$ . Suppose we have  $\alpha$  with 1)  $\gamma = 0$  and 2)  $\gamma = \nu - 1$ , then, when  $\gamma = 0$ , we can have input  $x = (\gamma', \delta')$  with  $\gamma'$  always greater than equal to 0. Let us see how our decode algorithm works in this case.

- Case 1: The decoder will compute  $\text{Res}^1$ . However,  $\text{Res}^1$  will equal to  $\text{cw}_{\delta'}^1$ , only with probability  $1/(|\mathbb{G}|)$  which is negligible as  $\text{cw}_{\delta'}^1$  was randomly initialised. So, the decoder will go for computing  $\text{Res}^2$  and  $\text{Res}^3$ .
- Case 2: If  $\text{Res}^2 == \text{Res}^3$  and it will return  $\text{Res}^2$  as the evaluation result as  $\gamma' > 0$  is true in this situation.
- Case 3: Finally, the decoder computes  $\text{Res}^4$  using  $Y_{i,1}$  and  $\text{cw}_{\gamma'}^2$ . This is the case for  $\gamma' == 0$ , which will be evaluated correctly.

When  $\gamma = \nu - 1$ , we can only have input  $x = (\gamma', \delta')$  with  $\gamma' \leq \nu - 1$ . In this setting, our decoder works as follows:

- Case 1:  $\text{cw}_{\delta'}^1 == \text{Res}^1$  is true only if  $\gamma' < \nu - 1$ , and outputs  $1_{\mathbb{G}}$ .
- Case 2:  $\text{Res}^2 == \text{Res}^3$  will never be true because  $\text{cw}_{\delta'}^3$  will have random initialization. It is highly unlikely that  $\text{cw}_{\delta'}^3 == \text{cw}_{\delta'+1}^3$  (in fact, this condition is true only with probability  $1/(|\mathbb{G}|)$ ), and also in case if  $\text{cw}_{\delta'}^3 == \text{cw}_{\delta'}^3$ , another term associated with  $\text{Res}^2$  is  $G^{\delta'}(s_{\gamma'})$  and with  $\text{Res}^3$  is  $G^{\delta'+1}(s_{\gamma'})$  and again  $G^{\delta'}(s_{\gamma'})$  equals to  $G^{\delta'+1}(s_{\gamma'})$  with probability  $1/(|\mathbb{G}|)$ . Hence, the overall probability of getting  $\text{Res}^2 == \text{Res}^3$  for this particular case is  $(1/(|\mathbb{G}|))^2$  which is negligible.
- Case 3:  $\text{Res}^4$  will always output the correct answer for  $\gamma' = \nu - 1$ .

Hence, randomized initialization of  $\text{CW}^1$ ,  $\text{CW}^2$ , and  $\text{CW}^3$  will never allow any leak about  $\alpha$ , including the corner cases mentioned above.

### 3.3.1 Distributed Interval Function (DIF).

An interval function  $\mathcal{F}_{\alpha_1, \alpha_2}^{\langle \beta \rangle}$  outputs  $\beta$  if the input falls in the range of  $[\alpha_1, \alpha_2]$ . On a high level, the interval function combines two comparison functions. The first comparison function indicates the start of the interval function, while the second indicates the end of the interval. Let  $f_{\alpha_1, \beta}^{\langle \beta \rangle}$  be the first comparison function, which means that for any input  $x$  if  $x$  is greater than equal to  $\alpha_1$ , then it will output  $\beta - \beta'$ , otherwise 0. Similarly, the second comparison function is  $f_{\alpha_2, \beta'}^{\langle \beta \rangle}$ , which means that for any input  $x$ , if  $x$  is lesser than equal to  $\alpha_2$ , it will return  $\beta'$ , otherwise 0. Adding outputs of these two comparison functions yields the output of the interval function. To extend our distributed function secret sharing to the interval function, we rely on the compact construction of the comparison function discussed in Section 3.3.

## 4 Application: Secure and Distributed Neural Network Inference

This section delves into a well-explored application where our DCF play a pivotal role in ensuring secure non-linearity within the system. This application pertains to privacy-preserving machine learning (PPML), enabling secure inferences between the model and data owners. Many PPML approaches, like [JVC18, RRK<sup>+</sup>20, MLS<sup>+</sup>20] have relied on secure multi-party computation (MPC) based on secret sharing, Garbled Circuits and oblivious transfer. However, MPC-based secure inference has high communication costs and requires multiple communication rounds. More specifically, in terms of communication overhead, fully homomorphic encryption (FHE) based machine learning inference frameworks typically incur the lowest costs. Conversely, techniques based on multi-party computations (MPC), such as Garbled Circuits, Secret Sharing, and Oblivious Transfers, tend to impose the highest communication overhead. Privacy-preserving machine learning (PPML) inference systems built on FSS-based secure computation lie between these extremes. This observation stems from various FSS-based prior work, including AriaNN [RPB20], FSSNN [YJG<sup>+</sup>23], and Fastsecnet [HLC<sup>+</sup>23], particularly in the evaluation of ReLU functions. Notably, in Garbled Circuit-based implementations, ReLU evaluation typically requires at least four rounds of online communication, whereas the most efficient FSS-based ReLU evaluation entails only one round. This demonstrates that MPC-based secure inference generally necessitates more online communication rounds than FSS-based secure inference. FSS has been used in PPML research, such as the work in [RPB20, JGB<sup>+</sup>24, HLC<sup>+</sup>23, Wag22, YJG<sup>+</sup>23, GJM<sup>+</sup>23], which shows how easy it is to use for secure two-party computation protocols in the dealer model. However, they do not support inferences when the model is distributed among more than two parties. In general, extending schemes beyond two parties distributes trust among multiple entities, thereby decentralizing the system and increasing its resilience against adversarial attacks. In machine learning, models can be owned/shared by multiple parties rather than solely owned by one party. This distributed ownership enhances the realism of machine learning inference and significantly bolsters resistance against model theft. To compromise the model, an adversary would need to compromise all participating servers, thus necessitating the need for more than two-party ML inference. Hence, we pose the following question:

*Can we leverage our proposed FSS schemes for comparison functions to enable secure neural network inference when the trained model is distributed among multiple servers?*

In the following section, we answer the above question in the affirmative by demonstrating the construction of distributed ReLU from a distributed comparison function. It is worth noting that the non-linear layers, such as the ReLU operation, tend to be more resource-intensive, requiring substantial communication and involving a high round complexity. For instance, the ReLU operation accounts for a significant 93% of the ResNET32’s online runtime in Delphi [MLS<sup>+</sup>20]. The above state-of-the-art literature on PPML deals with two-party secure protocols. However, in our specific application, we assume that the trained model is distributed (secret shared) among  $p$ -servers, with each server holding the necessary keys for distributed ReLU to carry out inference across the distributed model. MPC-based inference protocols, such as [MZ17], typically rely on trusted dealers to generate Beaver’s triples in the offline phase. In our protocol, similar to conventional MPC approaches, we utilize a trusted dealer solely for Beaver’s triple generation in the offline phase. Additionally, we introduce a trusted decoder to minimize computation and communication. Our FSS-based techniques significantly reduce communication overhead between the client and server. While the most efficient MPC-based inference protocol requires at least 4 online rounds of communication to evaluate ReLU, our protocol accomplishes this in just one round. Moreover, our protocol addresses the limitation of existing protocols, which are often

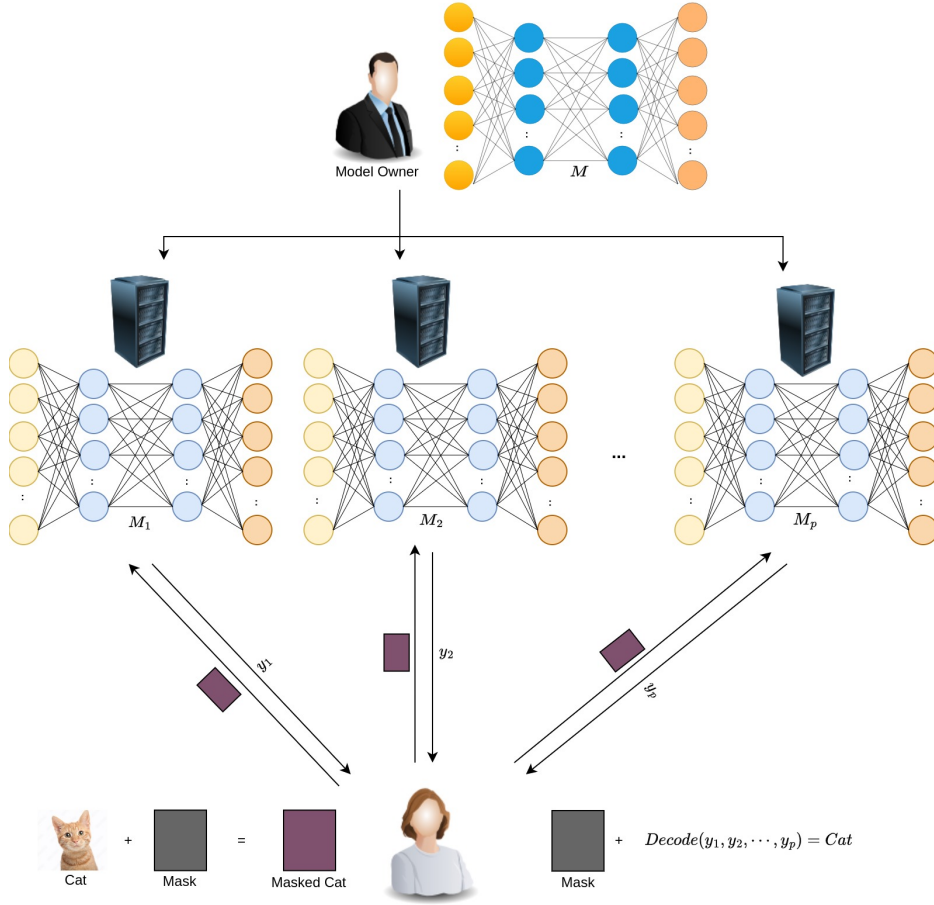


Figure 4: Machine learning Inference Framework.

restricted to 2 or 3 parties and lack generalization for arbitrary numbers of parties Table 6 demonstrates the computationally lightweight nature of a decode function. It is important to note that, similar to other FSS-based machine learning inference frameworks, our framework also requires interaction and aggregation after executing each non-linear layer. This results in layer-wise communication, which is still better than the communication overhead of any MPC-based inference. Moreover, it appears that the computational efficiency of FSS-based applications offers an additional benefit on top of communication efficiency. More concretely, among existing 3-party MPC-based PPML frameworks, such as [MZ17, SCS<sup>+</sup>20, WGC19], each has distinct characteristics. [MZ17] shares the ML model between two servers only, [SCS<sup>+</sup>20] retains the model with one server, and [WGC19], although sharing the model among 3 servers, lacks scalability to more servers. Despite differences in our model-sharing approach compared to existing literature, we provide an approximate comparison with [SCS<sup>+</sup>20] and [WGC19], taking ReLU evaluation times from Table 3 of [SCS<sup>+</sup>20]. For an input size of  $64 \times 16$ , [SCS<sup>+</sup>20] takes 1.65s, and [WGC19] takes 15.71s, whereas our ReLU evaluation takes 0.138s for 3-servers and 0.220s for 5-servers (see Table 5). Note that [SCS<sup>+</sup>20] and [WGC19] lack scalability to any arbitrary number of servers.

**Building distributed ReLU from DCF.** We employ similar techniques in [BCG<sup>+</sup>21, RPB20, HLC<sup>+</sup>23] to construct distributed ReLU from DCF. They rely on *offset*  $\text{ReLU}_r(x) =$

$\text{ReLU}(x - r)$ , where  $r$  is randomly sampled from  $\{0, 1\}^n$ . One can readily verify that for a given input  $x + r$ , the *offset* function eventually evaluates  $\text{ReLU}(x)$ . We can write  $\text{ReLU}_r(x) = x - r$ , if  $x \geq r$  and 0 otherwise, as  $f_{\alpha, \beta}^<(x)$ , representing it in the form of spline polynomial with coefficients  $\beta = (\beta_0, \beta_1)$ , where  $(\beta_0, \beta_1) = (1, -r)$  if  $x > r$  and  $(\beta_0, \beta_1) = (0, 0)$  otherwise. After computing shares of  $(\beta_0, \beta_1)$ , parties can locally compute shares of  $[\beta_0](x + r) + [\beta_1]$  which in fact are shares of the  $\text{ReLU}_r(x)$  function as  $[\beta_0]$  is share of 1 and  $[\beta_1]$  is share of  $-r$  for the case  $x \geq r$ . It is worth noting that the key size for distributed ReLU is twice that of the distributed comparison function. Additionally, the evaluation time of the distributed ReLU and the distributed comparison function differ by just one group operation.

#### 4.1 Distributed Machine Learning (ML) Inference from Proposed FSS

This section describes our application framework, which utilizes our proposed FSS scheme for secure distributed machine learning inference. By distributed keyword, we mean a particular model is secretly shared among multiple servers. By secretly sharing a model, we mean secret sharing weights and biases among servers. Such a distributed model would either have been trained in a distributed way or could have been distributed among servers after training. Please note that the trusted dealer shares the model among  $p$ -servers only once at the outset; after that, each server independently evaluates input  $x^*$  using its corresponding model shares. Also, the trusted decoder only interacts during nonlinear layers, like ReLU. The frequency of interactions varies based on the chosen network architecture. For instance, in an AlexNet CNN model with 5 convolution layers, each followed by ReLU, the decoder interacts only 5 times. We employ secure, online-offline-based FSS and secret sharing techniques to achieve secure inference from the distributed model. The system has three entities (see Figure 4): servers, a client, and a trusted dealer. A client is a data owner who wants to make inferences on the distributed model without revealing its data to the servers. The model is distributed among  $p$ -servers  $(S_1, \dots, S_p)$  such that if all  $p$ -servers combine their model shares, they can reconstruct the original model. A dealer is a trusted third party that generates keys for the FSS scheme, prepares correlated randomness and sends them to respective servers and the client. Similar to the inference technique in [MLS<sup>+</sup>20] and [HLC<sup>+</sup>23], we split the inference pipeline into an offline and an online phase. We mainly target achieving distributed model inference while maintaining the online overhead, the same as the prior secure inference techniques, especially when evaluating non-linear layers. For the non-linear layer, we use FSS-based techniques for ReLU and Maxpool, where  $p$ -party FSS keys are generated in the offline phase. Figure 5 and 6 depict the computation for online and offline phases of linear and non-linear layers. We formally define the cryptographic protocol below, followed by a description of the protocol and its security analysis.

**Definition 7** (Distributed ML Inference Protocol). A protocol  $\Pi$  involving a set of  $p$ -servers with model parameters secret shared among them as  $M = (M_1, \dots, M_p)$  and a client with input  $x$  is considered a distributed ML inference protocol if there exists a trusted decoder capable of securely aggregating and decoding the  $p$ -servers' output. Additionally, the protocol must satisfy the following guarantees.

- **Correctness.** For every set of model parameters  $M$  secret shared by the servers and for every input  $x$  provided by the client, the output produced by the client with the assistance of the trusted decoder, at the end of the protocol corresponds to the correct inference  $M(x)$ .
- **Security:**

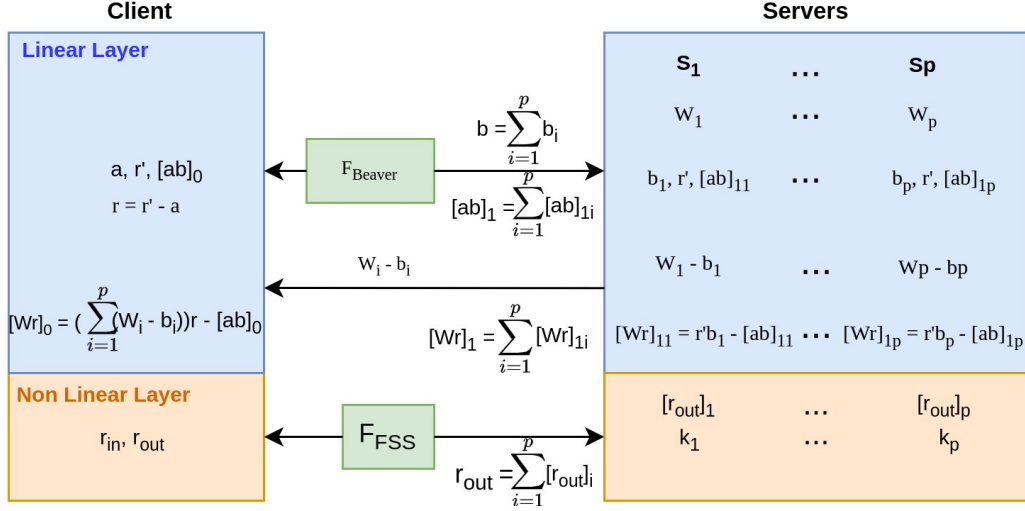


Figure 5: Offline linear and non-linear layer calculation

- **When at least one of the servers is honest.** A corrupted, semi-honest client should not learn anything about servers' model parameters  $M$ . Formally, this entails the existence of an efficient simulator  $\text{Sim}_C$  which simulates the honest computation such that  $\text{View}_C^\Pi \approx_c \text{Sim}_C(x, \text{Res})$ , where  $\text{View}_C^\Pi$  denotes the view of the client in the execution of  $\Pi$  (the view includes the client's input, randomness, and the transcript of the protocol), and  $\text{Res}$  denotes the output of the inference.
- **When only the client is honest.** The corruption of  $p$  semi-honest servers altogether does not learn anything about the private input  $x$  of the client. Formally, this necessitates the existence of an efficient simulator  $\text{Sim}_S$  such that  $\text{View}_S^\Pi \approx_c \text{Sim}_S(M)$ , where  $\text{View}_S^\Pi$  denotes the view of the servers in the execution of  $\Pi$  protocol.

The above inference protocol is structured into two phases: the offline phase and the online phase. We provide a detailed description of both these phases below.

**Offline Phase.** In this phase, the client and servers conduct pre-computations to prepare for the online phase of the protocol. Importantly, this phase remains independent of the client's input  $x$ . In the context of this protocol, distributing a model among  $p$  servers means additively distributing the model's weights. The linear layer computation can be formalised as the multiplication operation  $y = Wx = W(x - r) + Wr$ . Specifically, we focus on securely computing  $Wr$  (see Figure 5) in the offline phase as follows;

1. The dealer generates Beaver's multiplication triples as  $(a, b, ab)$ , where  $a, b \in Z_q$ . It then generates  $p + 1$  additive shares of  $ab$  as  $[ab]_0, [ab]_{11}, \dots, [ab]_{1p}$ . Note that in the original Beaver triples for two parties, we generate the shares of  $ab$  as  $[ab]_0$  and  $[ab]_1$ . While in our setting, we further distribute  $[ab]_1$  among  $p$  servers as  $([ab]_{11}, \dots, [ab]_{1p})$ . Similarly,  $p$  shares of  $b$  are  $[b]_1, \dots, [b]_p$ . It also randomly samples  $r' \in Z_q$  and then sends  $(a, r', [ab]_0)$  to the client and  $([b]_i, r', [ab]_{1i})$  to server  $i$ .
2. Each server computes  $(W_i - b_i)$  and sends the result to the client. Additionally, each server locally computes  $[Wr]_{11}, \dots, [Wr]_{1p}$  where  $[Wr]_{1i} = r'b_i - [ab]_{1i}$ ,  $i \in [p]$ .
3. The client on the other side computes  $[Wr]_0 = (\sum_{i=1}^p (W_i - b_i)r - [ab]_0)$ .



4. For the non-linear layer, the dealer generates FSS keys for the ReLU function. As the ReLU function comprises of two distributed comparison functions, it generates keys for two DCFs with  $\beta = (\beta_0, \beta_1)$  where  $(\beta_0, \beta_1) = (1, -r)$  if  $x > r$  and  $(\beta_0, \beta_1) = (0, 0)$  otherwise. The client receives the corresponding input and output masks as  $r_{in}$  and  $r_{out}$  for each comparison function. The dealer also generates  $p$  shares of  $r_{out}$  such that share  $[r_{out}]_i$  is sent to server  $S_i$  as a part of its FSS keys.

**Online Phase.** In the online phase, the client and servers use their pre-processed shares of  $Wr$ , DCFs keys, and output mask to evaluate linear and non-linear layers of the neural network. Figure 6 shows the online phase of the protocol. The computation goes as follows:

1. The client masks the input with  $r$ , and send  $(x - r)$  to all servers.
2. Each server uses its weight  $W_i$  and pre-computed results to compute  $[y]_{1i} = (x - r)W_i + [Wr]_{1i}$  which eventually becomes the share of  $Wx$ . All servers collectively combine their individual share  $[y]_{1i}$  and aggregate them to obtain the aggregated value  $[y]_1 = \sum_{i=1}^p [y]_{1i}$ .
3. The client holds the share of  $Wx$  in the form of  $[y]_0 = [Wr]_0$ . One can readily verify that  $[y]_0 + [y]_1 = Wx$ .
4. In the non-linear layer computation, the input shares are indeed the output shares of the previous layer like  $[x]_0 = [y]_0$  and  $[x]_1 = [y]_1$  with  $[y]_0$  and  $[y]_1$ . The client masks  $[x]_0$  with  $r_{in}$  and sends the masked input  $[x]_0 + r_{in}$  to servers.
5. Each server calculates  $x + r_{in}$  and uses it as the input for the ReLU function. After computing ReLU, each server masks its partial computation with shares of  $r_{out}$  received during the offline phase. Let us elaborate a bit more on the evaluation of ReLU. As mentioned before, ReLU uses two DCFs with  $\beta = (\beta_1, \beta_2)$ , so all servers evaluate two DCFs on input  $x + r_{in}$  and return the corresponding partial evaluation to the decoder.

After the online phase, a decoder aggregates the partial computations for each of the DCFs. The decoder uses its own  $r_{out}$  value corresponding to each DCF to output the final evaluation for both DCFs. Then, it multiplies the output of the first DCF with  $x + r_{in}$  and adds it with the evaluation result of the second DCF to compute the distributed ReLU function.

Note that, for each input  $x$  that needs evaluation during the online phase, the dealer randomly selects an input mask ( $r_{in}$ ) and generates the corresponding correlated FSS keys during the offline phase. The dealer then sends these keys to the servers and the mask ( $r_{in}$ ) to the client. For each new input, the client receives a different mask, say  $r'_{in}$ , and the corresponding FSS keys are distributed among the servers. This technique of using a different mask for each input is analogous to using Beaver triples, where each evaluation utilizes a unique set of triples. In fact, our approach of using a freshly sampled mask for each input is along the lines of prior works on FSS in the online-offline paradigm [BGI19, BCG<sup>+</sup>21, YJG<sup>+</sup>23].

The Maxpool function computes the maximum value over  $d$  elements  $x_1, x_2, \dots, x_d$ . We can split the inputs into tree reduction architectures, which recursively partition the input into two halves and then compare the elements of each half. For each two-element  $x_i$  and  $x_j$ , the client and server compute  $\max([x_i], [x_j]) = \text{ReLU}([x_i] - [x_j]) + [x_j]$ . Hence, the evaluation complexity of maxpool comes from the  $d - 1$  evaluation of ReLU.

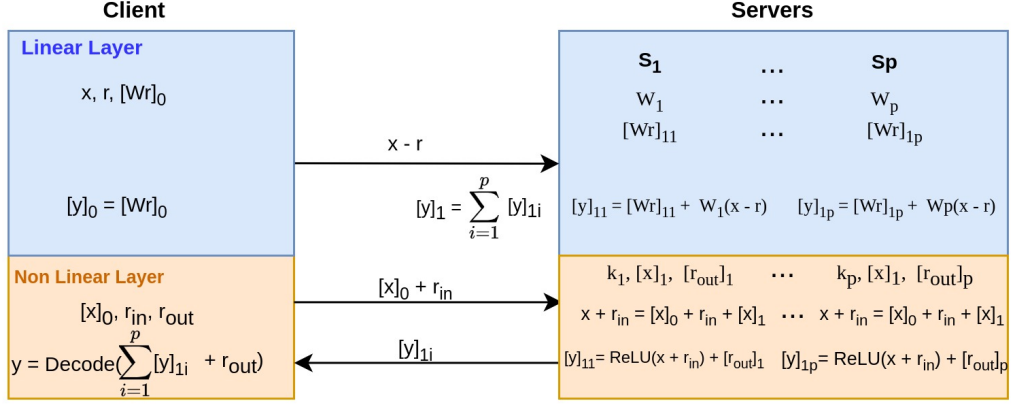


Figure 6: Online linear and non-linear layer calculation

## 4.2 Security

**Theorem 3.** *Assuming the existence of a secure  $p$ -party distributed point function (DCF) and a secure protocol for Beaver's triple generation and multiplication procedure, the protocol  $\Pi$  described above is a cryptographic inference protocol (see Definition 7).*

*Proof.* Following the security notion of cryptographic inference protocol from [MLS<sup>+</sup>20], we provide the description of simulators for two exhaustive and mutually exclusive cases: 1) when at least one of the servers is honest, 2) when only the client is honest. For each case, we argue that given the existence of a corresponding simulator, we prove the protocol's security using hybrid-based arguments and show that two consecutive hybrids are computationally indistinguishable from each other. Hence, the real-world view of the client (server) and the simulated view of the client (server) remains indistinguishable.

**At least one of the servers is honest.** Without loss of generality, let us assume server  $S_i$  to be a honest server, and rest  $p-1$  servers and the client is passively corrupted. We consider the case of maximal corruption here, and hence, it automatically incorporates the other cases where more than one server is honest. We denote the simulator as  $\text{Sim}_C(x, \text{Res})$ . The simulator takes in the client's input  $x$  and proceeds as follows to output  $\text{Res}$ :

### 1. Offline Phase.

- (a)  $\text{Sim}_C$  receives tuple  $(b_i, r', [ab]_{1i})$  from the dealer as a share of Beaver's triple. It selects a random weight share as  $\hat{W}_i$  (in place of the honest weight share  $W_i$ ) and returns  $\hat{W}_i - b_i$  to the client.
- (b)  $\text{Sim}_C$  also computes shares  $[Wr]_{1i} = r'b_i - [ab]_{1i}$ . Note that it relies on the simulator for Beaver's triple generator for simulated values of  $b_i$  and  $[ab]_{1i}$ .
- (c) For the non-linear layer, it receives the DCF key  $k_i$  (corresponding to the ReLU function) and uses it for partial computation of ReLU.

2. **Online Phase.** In the online phase,  $\text{Sim}_C$  receives  $x-r$  from client and computes  $[y]_{1i} = [Wr]_{1i} - \hat{W}_i(x-r)$ . It then broadcasts its  $[y]_{1i}$  to other servers. After receiving  $[y]_{1i}$  from rest of the servers, it aggregates it as  $[y]_1 = \sum_{i=1}^p [y]_{1i}$ . For the non-linear layer, the  $\text{Sim}_C$  receives  $[x]_0 + r_{in}$  (here  $[x]_0$  is the output shares from the previous linear layer) from the client and uses its previous linear layer computation as  $[x]_1 = [y]_1$ . The simulator then computes  $x + r_{in} = ([x]_0 + r_{in}) + [x]_1$ . For DCF evaluation, it uses the DCF keys and masks the output with  $[r_{out}]_i$ . Finally, it sends  $[y]_{1i} = \mathcal{F}_{\alpha, \beta}^<(x + r_{in}) + [r_{out}]_i$ .

Using the above simulator, we have three hybrids starting from the real-world distribution of the protocol and going to the ideal-world distribution. In the final simulated distribution, the simulator does not use the weight shares of the honest server.

- **Hybrid<sub>0</sub>** : This hybrid corresponds to the transcripts of real-world distribution where the  $i^{\text{th}}$  server (honest) also participates in the protocol and uses its original weight shares  $W_i$ .
- **Hybrid<sub>1</sub>** : In this hybrid, the simulator  $\text{Sim}_C$  starts by replacing the real Beaver's triples with simulated ones. Specifically, instead of using the real  $(a, b, ab)$  from Beaver's triples generator, the simulator uses simulated values  $(\hat{a}, \hat{b}, \hat{a}\hat{b})$ .

**Indistinguishability argument for Hybrid<sub>0</sub> and Hybrid<sub>1</sub>.** By the security of Beaver's triple generation protocol, the real triples  $(a, b, ab)$  are indistinguishable from the simulated triples  $(\hat{a}, \hat{b}, \hat{a}\hat{b})$ . Since  $\text{Sim}_C$  uses these indistinguishable triples, the view of any corrupted party cannot distinguish whether the triples were real or simulated. Thus, Hybrid<sub>1</sub> is computationally indistinguishable from Hybrid<sub>0</sub>.

- **Hybrid<sub>2</sub>** : This hybrid modifies Hybrid<sub>1</sub> by using random weight shares  $\hat{W}_i$  in place of the honest server's original weight shares  $W_i$ .

**Indistinguishability argument for Hybrid<sub>1</sub> and Hybrid<sub>2</sub>.** Since the original weight  $W_i$  of the honest server is not known to any corrupted party and is uniformly random, replacing it with another uniformly random weight  $\hat{W}_i$  will not alter the distribution observable by the corrupted parties. Thus, the transition from Hybrid<sub>1</sub> to Hybrid<sub>2</sub> remains computationally indistinguishable.

Hence, Hybrid<sub>2</sub> is indistinguishable from Hybrid<sub>0</sub> by the above two arguments of indistinguishability. This proves the weights are secure even if the client and  $p - 1$  servers are corrupted.

**Only client is honest.** In this scenario, the corrupted servers altogether should not learn any information about the client's input  $x$ . Let us denote the simulation for the client in view of corrupted servers as  $\text{Sim}_S$ . Given the servers weight  $W_1, \dots, W_p$  as inputs to  $\text{Sim}_S$ , it proceeds as follows:

#### 1. Offline Phase.

- (a)  $\text{Sim}_S$  receives tuple  $(a, r', [ab]_0)$  from the Beaver's triple generator and computes  $r = r' - a$ .
- (b) For the linear layer,  $\text{Sim}_S$  receives  $W_i - b_i$  from all the servers and locally compute  $[Wr]_0 = \sum_{i=1}^p (W_i - b_i)r - [ab]_0$ .
- (c) For the non-linear layer,  $\text{Sim}_S$ , it samples  $r_{in}$  and  $r_{out}$  and generate DCFs keys for the offset function  $g_{r_{in}, r_{out}}$ . Note that  $\text{Sim}_S$  invokes the simulator for DCF to generate DCF keys corresponding to ReLU. The simulation-based definition of FSS can be derived from the indistinguishability-based definition of FSS, where instead of sending two functions  $g_0, g_1$ , it randomly chooses one function and generates the FSS keys.

#### 2. Online Phase.

- (a) In this phase, the simulator sends a random input  $\hat{x}$  to the servers instead of the client's original input  $x$ .  $\text{Sim}_S$  sends  $\hat{x} - r$ , which comprises of the values  $\hat{x}$  and  $r$ . Since term  $r = r' - a$  contains  $a$  from Beaver triple, the simulator invokes Beaver's triple protocol simulator to generate  $r$ . For the non-linear layer, it masks the previous layer input shares using  $r_{in}$  and sends them to the servers.

Using simulator  $\text{Sim}_S$ , we provide the security of the protocol  $\Pi$  using hybrid-based arguments to show the indistinguishability of two consecutive hybrids, eventually leading to the indistinguishability of the real and ideal world distribution.

- **Hybrid<sub>0</sub>** : This corresponds to the real-world distribution where the client uses its original input  $x$  and the servers use their real weights shares  $W_1, \dots, W_p$ .
- **Hybrid<sub>1</sub>** : In this hybrid, the simulator  $\text{Sim}_S$  invokes the Beaver’s triple generating simulator and uses corresponding  $\hat{a}$  and  $[\hat{a}\hat{b}]_0$  in the protocol.

**Indistinguishability arguments for Hybrid<sub>0</sub> and Hybrid<sub>1</sub>.** By the security of the Beaver’s triples generation protocol, the real triples  $(a, b, ab)$  are indistinguishable from the simulated triples  $(\hat{a}, \hat{b}, \hat{a}\hat{b})$ . As  $\text{Sim}_S$  uses these indistinguishable triples, the view of corrupted servers cannot distinguish if the triples are real or simulated. Thus, Hybrid<sub>1</sub> is computationally indistinguishable from Hybrid<sub>0</sub>.

- **Hybrid<sub>2</sub>** : This hybrid modifies Hybrid<sub>1</sub> by replacing the client’s original input  $x$  with a random input  $\hat{x}$ . The simulator sends  $\hat{x} - r$  instead of  $x - r$  to the servers. Here,  $r$  is generated based on simulated Beaver’s triples  $(\hat{a}, \hat{b}, \hat{a}\hat{b})$  using formula  $r = r' - \hat{a}$ .

**Indistinguishability arguments for Hybrid<sub>1</sub> and Hybrid<sub>2</sub>.** Since  $r$  is a random value and the input  $x$  is masked by this randomness, replacing  $x$  with another random input  $\hat{x}$  results in  $\hat{x} - r$ , which remains indistinguishable from  $x - r$ . Given the randomness of  $r$ , both  $x - r$  and  $\hat{x} - r$  appear as random value to the corrupted servers. Thus, Hybrid<sub>2</sub> is computationally indistinguishable from Hybrid<sub>1</sub>.

Hence, through the series of the above hybrid arguments, we prove that the client’s input remains secure when all servers are corrupted.

### 4.3 Experimental Results.

None of the prior work implements ReLU and distributed ML inference in multi-server client model. We are the first to comprehensively analyze the computation time for the distributed ReLU activation function and implement it based on additive secret sharing and the DDH-based SHPRG assumptions. This instantiation relies on the distributed comparison function. The results are presented in Table 5, and Table 6, which showcases the total evaluation and decoding times, respectively, for the instantiated ReLU function. The tables use the number of parties as columns and various domain sizes for the ReLU function as rows. In this analysis, we primarily focus on online phase evaluation time and decoding time, though one can also infer offline phase evaluation, like key generation for ReLU, as follows: With the increase in the number of parties, there should be a relatively small impact compared to the growth in the number of bits for function inputs. The key generation algorithm generates a key for each row, which scales with the order of  $2^{n/2}$ . Consequently, variations in the value of  $n$  have a more significant effect than changes in the number of parties. This trend becomes evident as we move down the column, showing minimal change in key generation time.

In Table 5, it is clear that the domain size of the function has minimal impact on the evaluation time of the parties. This phenomenon occurs because, unlike the key generation process, the evaluation algorithm is independent of the function’s domain size (the number of invocations to SHPRG remains consistent). In the evaluation algorithm, we already possess the values  $\gamma'_x$  and  $\delta'_x$  to select the share value directly, which means that even with a larger domain size, the evaluation time is not significantly affected. However, an increase in the value of  $p$  leads to an almost linear increase in total evaluation time, as a higher  $p$  implies more parties needing to evaluate, resulting in a longer aggregated time. Each party takes approximately 45 milliseconds to evaluate the distributed ReLU for all domain sizes.

Table 5: Total evaluation time for ReLU (in seconds)

bits/parties	(3)	(5)	(7)	(9)	(11)	(13)	(15)
4	0.138	0.229	0.319	0.405	0.501	0.588	0.662
8	0.139	0.237	0.318	0.401	0.497	0.587	0.674
12	0.138	0.224	0.313	0.405	0.495	0.573	0.669
16	0.137	0.225	0.317	0.401	0.491	0.573	0.670
20	0.136	0.220	0.307	0.394	0.485	0.564	0.654

Table 6: Decode time for ReLU (in milliseconds)

bits/parties	(3)	(5)	(7)	(9)	(11)	(13)	(15)
4	0.072	0.101	0.125	0.150	0.177	0.203	0.229
8	0.070	0.098	0.130	0.155	0.170	0.196	0.230
12	0.069	0.105	0.127	0.153	0.168	0.195	0.228
16	0.076	0.105	0.118	0.158	0.177	0.193	0.222
20	0.075	0.105	0.131	0.151	0.314	0.196	0.224

Finally, Table 6 provides insights into the decoding (aggregation) time for all the partial computations from different parties. Increasing the number of parties results in more *output group* operations on **PartComp** from different parties, leading to a linear increase in decoding time. Yet again, the decoding time remains independent of the domain size of the ReLU function because we already have the values  $\gamma'_x$  and  $\delta'_x$  determined based on the input  $x$ . Although in Table 5 and 6, we provide results for 20-bit and 15 parties, the trends remain similar when scaled to bigger values.

## Acknowledgements

We thank the anonymous reviewers and shepherds for their valuable comments, which have significantly improved the quality of this paper. Additionally, we thank Google Asia Pacific Pte Ltd. for partially supporting this project as part of the Google PhD fellowship program. Debdeep would like to thank the IBM Research OSCP AWARD FOR RESEARCH IN QUANTUM SAFE MULTIKEY HOMOMORPHIC ENCRYPTION.

## References

- [BBCG<sup>+</sup>21] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776. IEEE, 2021. doi:[10.1109/SP40001.2021.00048](https://doi.org/10.1109/SP40001.2021.00048).
- [BCG<sup>+</sup>19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 489–518. Springer, 2019. doi:[10.1007/978-3-030-26954-8\\_16](https://doi.org/10.1007/978-3-030-26954-8_16).
- [BCG<sup>+</sup>21] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 871–900. Springer, 2021. doi:[10.1007/978-3-030-77886-6\\_30](https://doi.org/10.1007/978-3-030-77886-6_30).

- [BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 337–367. Springer, 2015. doi:[10.1007/978-3-662-46803-6\\_12](https://doi.org/10.1007/978-3-662-46803-6_12).
- [BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, page 1292–1303, New York, NY, USA, 2016. Association for Computing Machinery. doi:[10.1145/2976749.2978429](https://doi.org/10.1145/2976749.2978429).
- [BGI19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *Theory of Cryptography Conference*, pages 341–371. Springer, 2019. doi:[10.1007/978-3-030-36030-6\\_14](https://doi.org/10.1007/978-3-030-36030-6_14).
- [BGIK22] Elette Boyle, Niv Gilboa, Yuval Ishai, and Victor I. Kolobov. Information-Theoretic Distributed Point Functions. In Dana Dachman-Soled, editor, *3rd Conference on Information-Theoretic Cryptography (ITC 2022)*, volume 230 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITC.2022.17>, doi:[10.4230/LIPIcs.ITC.2022.17](https://doi.org/10.4230/LIPIcs.ITC.2022.17).
- [BKKO20] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed oram. In *International Conference on Security and Cryptography for Networks*, pages 215–232. Springer, 2020. doi:[10.1007/978-3-030-57990-6\\_11](https://doi.org/10.1007/978-3-030-57990-6_11).
- [BLMR13] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic prfs and their applications. In *Annual Cryptology Conference*, pages 410–428. Springer, 2013. doi:[10.1007/978-3-642-40041-4\\_23](https://doi.org/10.1007/978-3-642-40041-4_23).
- [CGBM15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015. doi:[10.1109/SP.2015.27](https://doi.org/10.1109/SP.2015.27).
- [DFL<sup>+</sup>20] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. {DORY}: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1101–1119, 2020. URL: <https://www.usenix.org/conference/osdi20/presentation/dauterman-dory>.
- [DHRW16] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 93–122, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:[10.1007/978-3-662-53015-3\\_4](https://doi.org/10.1007/978-3-662-53015-3_4).
- [DIL<sup>+</sup>20] Samuel Dittmer, Yuval Ishai, Steve Lu, Rafail Ostrovsky, Mohamed Elsabagh, Nikolaos Kiourtis, Brian Schulte, and Angelos Stavrou. Function secret sharing for PSI-CA: With applications to private contact tracing. Cryptology ePrint Archive, Paper 2020/1599, 2020. <https://eprint.iacr.org/2020/1599>. URL: <https://eprint.iacr.org/2020/1599>.



- [DIL<sup>+</sup>22] Samuel Dittmer, Yuval Ishai, Steve Lu, Rafail Ostrovsky, Mohamed Elsabagh, Nikolaos Kiourtis, Brian Schulte, and Angelos Stavrou. Streaming and unbalanced psi from function secret sharing. In *International Conference on Security and Cryptography for Networks*, pages 564–587. Springer, 2022. doi:10.1007/978-3-031-14791-3\_25.
- [DS17] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, page 523–535, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3133967.
- [ECGZB21] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1775–1792. USENIX Association, August 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/eskandarian>.
- [GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 640–658. Springer, 2014. doi:10.1007/978-3-642-55220-5\_35.
- [GJM<sup>+</sup>23] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. Sigma: Secure gpt inference with function secret sharing. Cryptology ePrint Archive, Paper 2023/1269, 2023. <https://eprint.iacr.org/2023/1269>. URL: <https://eprint.iacr.org/2023/1269>.
- [GKW18] S Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server oram. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 141–157. Springer, 2018. doi:10.1007/978-3-030-03332-3\_6.
- [GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, may 1996. doi:10.1145/233551.233553.
- [GRS22] Gayathri Garimella, Mike Rosulek, and Jaspal Singh. Structure-aware private set intersection, with applications to fuzzy matching. In *Annual International Cryptology Conference*, pages 323–352. Springer, 2022. doi:10.1007/978-3-031-15802-5\_12.
- [GRS23] Gayathri Garimella, Mike Rosulek, and Jaspal Singh. Malicious secure, structure-aware private set intersection. In *Annual International Cryptology Conference*, pages 577–610. Springer, 2023. doi:10.1007/978-3-031-38557-5\_19.
- [HLC<sup>+</sup>23] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, and Tianwei Zhang. Fastsecnet: An efficient cryptographic framework for private neural network inference. *IEEE Transactions on Information Forensics and Security*, 18:2569–2582, 2023. doi:10.1109/TIFS.2023.3262149.
- [JGB<sup>+</sup>24] N. Jawalkar, K. Gupta, A. Basu, N. Chandran, D. Gupta, and R. Sharma. Orca: Fss-based secure training and inference with gpus. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 66–66, Los Alamitos, CA, USA, may 2024. IEEE Computer Society. URL: <https://doi.ieeecomput>

- [ersociety.org/10.1109/SP54263.2024.00063](https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar), doi:10.1109/SP54263.2024.00063.
- [JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, Baltimore, MD, August 2018. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar>.
- [MLS<sup>+</sup>20] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2505–2522. USENIX Association, August 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/mishra>.
- [MZ17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017. doi:10.1109/SP.2017.12.
- [NSSD22] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 229–248, Renton, WA, April 2022. USENIX Association. URL: <https://www.usenix.org/conference/nsdi22/presentation/newman>.
- [RPB20] Théo Ryffel, David Pointcheval, and Francis R. Bach. ARIANN: low-interaction privacy-preserving deep learning via function secret sharing. *CoRR*, abs/2006.04593, 2020. URL: <https://arxiv.org/abs/2006.04593>, arXiv:2006.04593.
- [RRK<sup>+</sup>20] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20*, page 325–342, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3417274.
- [SCS<sup>+</sup>20] Liyan Shen, Xiaojun Chen, Jinqiao Shi, Ye Dong, and Binxing Fang. An efficient 3-party framework for privacy-preserving neural network inference. In *European Symposium on Research in Computer Security*, pages 419–439. Springer, 2020. doi:10.1007/978-3-030-58951-6\_21.
- [TSS<sup>+</sup>20] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epi-one: Lightweight contact tracing with strong privacy. *CoRR*, abs/2004.13293, 2020. URL: <https://arxiv.org/abs/2004.13293>, arXiv:2004.13293.
- [VHG23] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A Bandwidth-Efficient distributed ORAM for 2- and 3-party computation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3907–3924, Anaheim, CA, August 2023. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/vadapalli>.
- [VSH22] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. Sabre: Sender-anonymous messaging with fast audits. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1953–1970, 2022. doi:10.1109/SP46214.2022.9833601.

- [Wag22] Sameer Wagh. Pika: Secure computation using function secret sharing over rings. *Proc. Priv. Enhancing Technol.*, 2022(4):351–377, 2022. URL: <https://doi.org/10.56553/popets-2022-0113>, doi:10.56553/POPET S-2022-0113.
- [WGC19] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019. doi:10.2478/popets-2019-0035.
- [YJG<sup>+</sup>23] Peng Yang, Zoe L. Jiang, Shiqi Gao, Jiehang Zhuang, Hongxiao Wang, Junbin Fang, Siuming Yiu, Yulin Wu, and Xuan Wang. FssNN: Communication-efficient secure neural network training via function secret sharing. Cryptology ePrint Archive, Paper 2023/073, 2023. <https://eprint.iacr.org/2023/073>. URL: <https://eprint.iacr.org/2023/073>.

## A Proof of Theorem 1

*Proof.* We define the hybrid experiments with distribution description as follows by marking the modifications with a gray rectangle in each step.

**Hybrid<sub>0</sub>** : This hybrid is the real key distribution for a function  $\mathcal{F}_{\alpha,\beta}$ . In  $H_0(P', \alpha, \mathcal{F}_{\alpha,\beta})$ ,  $P'$  denotes the set of corrupted parties with  $|P'| < p$ . The function  $\text{SSS}(\cdot)$  takes input as a seed and generates additive shares for all parties in  $P = \{P_1, \dots, P_p\}$ . However, the hybrids have the key distribution only for corrupted parties in  $P'$ .

$$H_0(P', \alpha, \mathcal{F}_{\alpha,\beta}) := \left( \text{CW}, \{\text{sh}_{i,\gamma'}\}_{P_i \in P'} : \begin{array}{l} s_\gamma \leftarrow \{0, 1\}^\lambda, \\ \text{sh}_{p,\gamma'} \leftarrow \begin{cases} \text{SSS}(s_\gamma), & \gamma' = \gamma \\ \text{SSS}(0), & \gamma' \neq \gamma \end{cases} \\ \{\text{sh}_{i,\gamma'}\}_{P_i \in P} \leftarrow \text{DPF.Gen}(\lambda, \mathcal{F}_{\alpha,\beta}) \\ \text{cw}_{\delta'} = \begin{cases} \beta_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{(\delta')} (s_\gamma)), & \delta' = \delta \\ 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{(\delta')} (s_\gamma)), & \delta' \neq \delta \end{cases} \end{array} \right)$$

$$H_1(P', \alpha, \mathcal{F}_{\alpha,\beta}) := \left( \text{CW}, \{\text{sh}_{i,\gamma'}\}_{P_i \in P'} : \begin{array}{l} \boxed{R \leftarrow \{0, 1\}^{m\mu}}, \\ s_\gamma \leftarrow \{0, 1\}^\lambda, \\ \text{sh}_{p,\gamma'} \leftarrow \begin{cases} \text{SSS}(s_\gamma), & \gamma' = \gamma \\ \text{SSS}(0), & \gamma' \neq \gamma \end{cases} \\ \{\text{sh}_{i,\gamma'}\}_{P_i \in P} \leftarrow \text{DPF.Gen}(\lambda, \mathcal{F}_{\alpha,\beta}) \\ \text{cw}_{\delta'} = \begin{cases} \beta_{\mathbb{G}} \otimes \boxed{R^{\delta'}}, & \delta' = \delta \\ 1_{\mathbb{G}} \otimes \boxed{R^{\delta'}}, & \delta' \neq \delta \end{cases} \end{array} \right)$$

**Hybrid<sub>1</sub>** : Replace  $\mathcal{G}_{\text{SHPRG}}(s_\gamma)$  with  $R$ , where  $R$  is randomly sampled from  $\{0, 1\}^{m\mu}$  (see grayed boxes). Note that all remaining steps are still performed with respect to  $\mathcal{F}_{\alpha,\beta}$  as given in Hybrid<sub>0</sub>.

**Claim.** For any polynomial  $\mathfrak{p}_1(n)$ , and  $(T, \epsilon_{\text{SHPRG}})$ -secure SHPRG, given a corrupted set of parties  $P'$ ,  $\alpha \in \{0, 1\}^n$ ,  $\mathcal{F}_{\alpha,\beta}$  and auxiliary input  $z$ , no adversary running in time  $T - \mathfrak{p}_1(n)$  can distinguish the distributions  $(H_0(P', \alpha, \mathcal{F}_{\alpha,\beta}), z)$  and  $(H_1(P', \alpha, \mathcal{F}_{\alpha,\beta}), z)$  with advantage greater than  $\epsilon_{\text{SHPRG}}$ .

**Proof 5.** Suppose there exists a set of corrupted parties  $P^*$ ,  $\alpha^*$ ,  $\mathcal{F}_{\alpha,\beta}^*$ , auxiliary input  $z^*$ , and an adversary  $\mathcal{A}^*$  that runs in the time  $T''$  for which,

$$\left| \Pr[K_{P^*} \leftarrow H_0(P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*) : \mathcal{A}^*(K_{P^*}, z^*) = 1] - \Pr[K_{P^*} \leftarrow H_1(P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*) : \mathcal{A}^*(K_{P^*}, z^*) = 1] \right| > \epsilon_{\text{SHPRG}}$$

We will use this adversary  $\mathcal{A}^*$  to construct an adversary  $\mathcal{B}$  for the underlying SHPRG. Define auxiliary input  $z_{\mathcal{B}} := \{P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*, z^*\}$ . In the SHPRG challenge,  $\mathcal{B}$  receives a correction word CW, which could either have been generated using random  $R \in \{0, 1\}^{m\mu}$  or using SHPRG like  $R = \mathcal{G}_{\text{SHPRG}}(s_\gamma) : s_\gamma \leftarrow \{0, 1\}^\lambda$ .

Adversary  $\mathcal{B}(1^\lambda, R, z_{\mathcal{B}})$ :

1. Parse  $z_{\mathcal{B}} := \{P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*, z^*\}$ .
2. Sample  $s_{\gamma^*}^* \in \{0, 1\}^\lambda$ .
3. Generate additive shares of  $s_{\gamma^*}^*$  if  $\gamma' = \gamma^*$ , and of 0 otherwise for all  $\gamma' \in \nu$ .
4. Consolidate shares of each party as  $\{\text{sh}_{i, \gamma'}\}_{P_i \in P^*}$  using a key generation algorithm.
5. Compute  $\text{cw}_{\delta'}$  as follows,

$$\text{cw}_{\delta'} = \begin{cases} \beta_{\mathbb{G}} \otimes R^{\delta'}, & \delta' = \delta^* \\ 1_{\mathbb{G}} \otimes R^{\delta'}, & \delta' \neq \delta^* \end{cases}$$

6. Define  $K_{P^*} = \{\text{CW} = \text{cw}_1, \dots, \text{cw}_\mu\} \parallel \{\text{sh}_{i, \gamma'}\}_{P_i \in P^*}$
7. Let  $\text{guess} \leftarrow \mathcal{A}^*(K_{P^*}, z^*)$ . Output the  $\text{guess}$ .

The running time of  $\mathcal{B}$  is equal to  $\text{time}(\mathcal{A}^*) + \text{time}(\text{DPF.Gen}) + \text{time}(\text{SHPRG}) = \text{time}(\mathcal{A}^*) + \text{p}_1(n)$  for some polynomial  $\text{p}_1(n)$ . By construction, if  $R$  is pseudorandom, then  $K_{P^*}$  is distributed precisely as  $H_0(P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*)$ , whereas if it is sampled randomly from  $\{0, 1\}^{m\mu}$  then  $K_{P^*}$  is distributed precisely as  $H_1(P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*)$ . Thus, the advantage of  $\mathcal{B}$  in the SHPRG security game is equal to the advantage of  $\mathcal{A}^*$  in distinguishing distributions, which is greater than  $\epsilon_{\text{SHPRG}}$ . If  $\mathcal{A}^*$  runs in time  $T'' \leq T - \text{p}_1(n)$ , then  $\mathcal{B}$  runs in time less than  $T$  and distinguishes SHPRG output from a truly random string of length  $m\mu$ , which would contradict the  $(T, \epsilon_{\text{SHPRG}})$ -security of the underlying SHPRG tool.

**Hybrid<sub>2</sub>** : Randomly generate  $\hat{s}_\gamma \in \{0, 1\}^\lambda$ . Use  $\hat{s}_\gamma$  to calculate shares and embed  $\hat{\beta}_{\mathbb{G}}$  in the generation of  $\text{cw}_{\delta'}$  for the function  $\mathcal{F}_{\hat{\alpha}, \hat{\beta}}$ , keeping  $R$  still randomly sampled from  $\{0, 1\}^{m\mu}$  same as in Hybrid<sub>1</sub>.

$$H_2(P', \alpha, \hat{\alpha}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) := \left( \text{CW}, \{\text{sh}_{i, \gamma'}\}_{P_i \in P'} : \begin{array}{l} R \leftarrow \{0, 1\}^{m\mu}, \\ s_\gamma \leftarrow \{0, 1\}^\lambda, \\ (\hat{s}_\gamma) \leftarrow \{0, 1\}^\lambda, \\ \text{sh}_{p, \gamma'} \leftarrow \begin{cases} \text{SSS}(\hat{s}_\gamma), & \gamma' = \hat{\gamma} \\ \text{SSS}(0), & \gamma' \neq \hat{\gamma} \end{cases} \\ \{\text{sh}_{i, \gamma'}\}_{P_i \in P} \leftarrow \text{DPF.Gen}(\lambda, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) \\ \text{cw}_{\delta'} = \begin{cases} \beta_{\mathbb{G}} \otimes R^{\delta'}, & \delta' = \hat{\delta} \\ 1_{\mathbb{G}} \otimes R^{\delta'}, & \delta' \neq \hat{\delta} \end{cases} \end{array} \right)$$

**Claim.** For any polynomial  $\mathfrak{p}_2(n)$ , and  $(T, \epsilon_{\text{SSS}})$ -secure SSS, given a corrupted set of parties  $P'$ ,  $\alpha \in \{0, 1\}^n$ ,  $\hat{\alpha} \in \{0, 1\}^n$ ,  $\mathcal{F}_{\alpha, \beta}$ ,  $\mathcal{F}_{\hat{\alpha}, \hat{\beta}}$  and auxiliary input  $z$ , no adversary running in time  $T - \mathfrak{p}_2(n)$  can distinguish the distributions  $(H_1(P', \alpha, \mathcal{F}_{\alpha, \beta}), z)$  and  $(H_2(P', \alpha, \hat{\alpha}, \mathcal{F}_{\alpha, \beta}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}), z)$  with advantage greater than  $\epsilon_{\text{SSS}}$ .

**Proof 6.** Suppose there exists a set of corrupted parties  $P^*$ ,  $\alpha^*$ ,  $\hat{\alpha}^*$ ,  $\mathcal{F}_{\alpha, \beta}^*$ ,  $\mathcal{F}_{\hat{\alpha}, \hat{\beta}}^*$ , auxiliary input  $z^*$ , and an adversary  $\mathcal{A}^*$  that runs in time  $T''$  for which,

$$\left| \Pr[K_{P^*} \leftarrow H_1(P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*) : \mathcal{A}^*(K_{P^*}, z^*) = 1] - \Pr[K_{P^*} \leftarrow H_2(P^*, \alpha^*, \hat{\alpha}^*, \mathcal{F}_{\alpha, \beta}^*, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}^*) : \mathcal{A}^*(K_{P^*}, z^*) = 1] \right| > \epsilon_{\text{SSS}}$$

We will use this adversary  $\mathcal{A}^*$  to construct an adversary  $\mathcal{B}$  for the underlying SSS scheme. Define auxiliary input  $z_{\mathcal{B}} := \{P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*, z^*\}$ . In the SSS based challenge,  $\mathcal{B}$  receives seeds  $S = s_{\gamma^*}^*$  which could either have been  $s_{\gamma} \in \{0, 1\}^\lambda$  or  $\hat{s}_{\hat{\gamma}} \in \{0, 1\}^\lambda$ .

Adversary  $\mathcal{B}(1^\lambda, S, z_{\mathcal{B}})$ :

1. Parse  $z_{\mathcal{B}} := \{P^*, \alpha^*, \mathcal{F}_{\alpha, \beta}^*, z^*\}$ .
2. Write  $S = s_{\gamma^*}^* \in \{0, 1\}^\lambda$ .
3. Generate additive shares of  $s_{\gamma^*}^*$  if  $\gamma' = \gamma^*$ , and of 0 otherwise for all  $\gamma' \in \nu$ .
4. Consolidate shares of each party as  $\{\text{sh}_{i, \gamma'}\}_{P_i \in P^*}$  using a key generation algorithm.
5. Sample  $R \in \{0, 1\}^{m\mu}$  and compute  $\text{cw}_{\delta'}$  as follows,

$$\text{cw}_{\delta'} = \begin{cases} \beta_{\mathbb{G}} \otimes R^{\delta'}, & \delta' = \delta^* \\ 1_{\mathbb{G}} \otimes R^{\delta'}, & \delta' \neq \delta^* \end{cases}$$

6. Define  $K_{P^*} = \{\text{CW} = \text{cw}_1, \dots, \text{cw}_\mu\} \parallel \{\text{sh}_{i, \gamma'}\}_{P_i \in P^*}$
7. Let  $\text{guess} \leftarrow \mathcal{A}^*(K_{P^*}, z^*)$ . Output the  $\text{guess}$ .

The running time of  $\mathcal{B}$  is equal to  $\text{time}(\mathcal{A}^*) + \text{time}(\text{DPF.Gen}) + \text{time}(\text{SSS}) = \text{time}(\mathcal{A}^*) + \mathfrak{p}_2(n)$  for some fixed polynomial  $\mathfrak{p}_2(n)$ . By construction, if  $S$  is  $s_{\gamma} \in \{0, 1\}^\lambda$  then  $K_{P^*}$  is distributed precisely as  $H_1(P^*, \alpha, \mathcal{F}_{\alpha, \beta})$ , whereas if it is  $\hat{s}_{\hat{\gamma}} \in \{0, 1\}^\lambda$  then  $K_{P^*}$  is distributed precisely as  $H_2(P^*, \alpha^*, \hat{\alpha}^*, \mathcal{F}_{\alpha, \beta}^*, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}^*)$ . Thus the advantage of  $\mathcal{B}$  in the SSS security game is equal to the advantage of  $\mathcal{A}^*$  in distinguishing distributions, which is greater than  $\epsilon_{\text{SSS}}$ . If  $\mathcal{A}^*$  runs in time  $T'' \leq T - \mathfrak{p}_2(n)$ , then  $\mathcal{B}$  runs in time lesser than  $T$  and distinguishes between two shares generated for a corrupted number of parties, which would contradict the  $(T, \epsilon_{\text{SSS}})$ -security of the underlying SSS.

**Hybrid<sub>3</sub>** : Replace  $R$  used during  $\text{cw}_{\delta'}$  with  $\mathcal{G}_{\text{SHPRG}}^\delta(\hat{s}_{\hat{\gamma}})$  keeping rest of the steps same as **Hybrid<sub>2</sub>**.

$$H_3(P', \hat{\alpha}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) := \left( \text{CW}, \{\text{sh}_{i, \gamma'}\}_{P_i \in P'} : \begin{array}{l} R \leftarrow \{0, 1\}^{m\mu}, \\ s_\gamma \leftarrow \{0, 1\}^\lambda, \\ (\hat{s}_\gamma) \leftarrow \{0, 1\}^\lambda, \\ \text{sh}_{p, \gamma'} \leftarrow \begin{cases} \text{SSS}(\hat{s}_\gamma), & \gamma' = \hat{\gamma} \\ \text{SSS}(0), & \gamma' \neq \hat{\gamma} \end{cases} \\ \{\text{sh}_{i, \gamma'}\}_{P_i \in P} \leftarrow \text{DPF.Gen}(\lambda, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) \\ \text{CW}_{\delta'} = \begin{cases} \hat{\beta}_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{(\delta')}(\hat{s}_\gamma)), & \delta' = \hat{\delta} \\ 1_{\mathbb{G}} \otimes \text{Inv}(\mathcal{G}_{\text{SHPRG}}^{(\delta')}(\hat{s}_\gamma)), & \delta' \neq \hat{\delta} \end{cases} \end{array} \right)$$

**Claim.** For any polynomial  $\mathfrak{p}_3(n)$ , and  $(T, \epsilon_{\text{SHPRG}})$ -secure SHPRG, given a corrupted set of parties  $P'$ ,  $\alpha \in \{0, 1\}^n$ ,  $\mathcal{F}_{\hat{\alpha}, \hat{\beta}}$  and auxiliary input  $z$ , no adversary running in time  $T - \mathfrak{p}_3(n)$  can distinguish the distributions  $(H_2(P', \alpha, \alpha', \mathcal{F}_{\alpha, \beta}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}), z)$  and  $(H_3(P', \alpha', \mathcal{F}_{\hat{\alpha}, \hat{\beta}}), z)$  with advantage greater than  $\epsilon_{\text{SHPRG}}$ .

**Proof 7.** The proof to the above claim has the same argument as for claim A except for  $\alpha$  and  $\mathcal{F}_{\alpha, \beta}$  replaced with  $\alpha'$  and  $\mathcal{F}_{\hat{\alpha}, \hat{\beta}}$  respectively.

Note that this distribution  $H_3(P', \alpha, \hat{\alpha}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}})$  is now precisely the distribution of honestly generated keys for the function  $\mathcal{F}_{\hat{\alpha}, \hat{\beta}}$  i.e.,  $H_3(P', \hat{\alpha}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) = H_0(P', \alpha, \mathcal{F}_{\alpha, \beta})$ .

We now combine claims A, A and A to complete the security proof of the proposed construction.

**Claim.** For any polynomial  $\mathfrak{p}(n) \in \text{poly}(n)$  such that our proposed scheme  $(\text{DPF.Gen}, \text{DPF.Eval})$  is  $(T', \epsilon')$ -secure DPF scheme for  $T' = T - \mathfrak{p}(n)$  and  $\epsilon' = 2\epsilon_{\text{SHPRG}} + \epsilon_{\text{SSS}}$ .

**Proof 8.** Suppose there exists an adversary  $\mathcal{A}^*$  which in time  $T''$  succeeds in the DPF security game for a set of corrupted parties  $P'$  with advantage greater than  $\epsilon'$ , i.e.,

$$\left| Pr \left[ \begin{array}{l} (f_0, f_1, \text{state}) \leftarrow \mathcal{A}(1^\lambda) \\ \{K_i\}_{P_i \in P} \leftarrow \text{DPF.Gen}(1^\lambda, f_1) \\ \text{guess} \leftarrow \mathcal{A}(K_{P^*}, \text{state}) \end{array} : \text{guess} = 1 \right] - Pr \left[ \begin{array}{l} (f_0, f_1, \text{state}) \leftarrow \mathcal{A}(1^\lambda) \\ \{K_i\}_{P_i \in P} \leftarrow \text{DPF.Gen}(1^\lambda, f_0) \\ \text{guess} \leftarrow \mathcal{A}(K_{P^*}, \text{state}) \end{array} : \text{guess} = 1 \right] \right| > \epsilon'$$

In particular, there exist a pair of function  $f_0 = f_{\alpha, \beta}$  and  $f_1 = f_{\hat{\alpha}, \hat{\beta}}$  and the value of state for which,

$$\left| Pr \left[ \begin{array}{l} \{K_i\}_{P_i \in P} \leftarrow \text{DPF.Gen}(1^\lambda, f_1) \\ \text{guess} \leftarrow \mathcal{A}(K_{P^*}, \text{state}) \end{array} : \text{guess} = 1 \right] - Pr \left[ \begin{array}{l} \{K_i\}_{P_i \in P} \leftarrow \text{DPF.Gen}(1^\lambda, f_0) \\ \text{guess} \leftarrow \mathcal{A}(K_{P^*}, \text{state}) \end{array} : \text{guess} = 1 \right] \right| > \epsilon'$$

Note that the distribution of  $K_{P^*}$  received by  $\mathcal{A}^*$  corresponds exactly to the distribution of  $K_{P^*} \leftarrow H_0(P^*, \alpha_c, \mathcal{F}_{\alpha_c, \beta_c})$  for the corresponding function  $f_{\alpha_c, \beta_c}$  (indeed,  $H_0$  was defined to be the honest key distribution). That is, for this  $(f_0, f_1, \text{state})$ , it holds that

$$\left| Pr[K_{P^*} \leftarrow H_0(P^*, \alpha, \mathcal{F}_{\alpha, \beta}) : \mathcal{A}^*(K_{P^*}, \text{state}) = 1] - Pr[K_{P^*} \leftarrow H_0(P^*, \hat{\alpha}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) : \mathcal{A}^*(K_{P^*}, \text{state}) = 1] \right| > \epsilon'$$

Now, since  $\epsilon' = \epsilon_{\text{SSS}} + 2\epsilon_{\text{SHPRG}}$ , then atleast one of the following must hold:

1.  $|Pr[K_{P^*} \leftarrow H_0(P^*, \alpha, \mathcal{F}_{\alpha, \beta}) : \mathcal{A}^*(K_{P^*}, z^*) = 1] - Pr[K_{P^*} \leftarrow H_1(P^*, \alpha, \mathcal{F}_{\alpha, \beta}) : \mathcal{A}^*(K_{P^*}, z^*) = 1]| > \epsilon_{\text{SHPRG}}$

2.  $|Pr[K_{P^*} \leftarrow H_1(P^*, \alpha, \mathcal{F}_{\alpha, \beta}) : \mathcal{A}^*(K_{P^*}, z^*) = 1] - Pr[K_{P^*} \leftarrow H_2(P^*, \alpha, \hat{\alpha}, \mathcal{F}_{\alpha, \beta}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) : \mathcal{A}^*(K_{P^*}, z^*) = 1]| > \epsilon_{\text{SSS}}$
3.  $|Pr[K_{P^*} \leftarrow H_2(P^*, \alpha, \hat{\alpha}, \mathcal{F}_{\alpha, \beta}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) : \mathcal{A}^*(K_{P^*}, z^*) = 1] - Pr[K_{P^*} \leftarrow H_3(P^*, \hat{\alpha}, \mathcal{F}_{\hat{\alpha}, \hat{\beta}}) : \mathcal{A}^*(K_{P^*}, z^*) = 1]| > \epsilon_{\text{SHPRG}}$

But, by Claims A, A, and A, this cannot happen if  $\mathcal{A}^*$  runs in time  $T < T - \max_{i=1}^3 \mathfrak{p}_i(n)$ , where each  $\mathfrak{p}_i(n)$  is from different corresponding claims. Therefore, security of DPF holds for the polynomial  $\mathfrak{p}(n) = \max_{i=1}^3 \mathfrak{p}_i(n)$ .

Claim A, Claim A, Claim A, and Claim A proves Theorem 1.