Communication Efficient Secure Logistic Regression

Amit Agarwal
UIUC, Champaign
Google, New York
amita2@illinois.edu

Stanislav Peceny
Georgia Tech, Atlanta
Google, New York
stan.peceny@gatech.edu

Mariana Raykova
Google, New York
marianar@google.com

Phillipp Schoppmann
Google, New York
schoppmann@google.com

Karn Seth
Google, New York
karn@google.com

Abstract—**We present a new construction for secure logistic regression training, which enables two parties to train a logistic regression model on private secret shared data. Our goal is to minimize online communication and round complexity, while still allowing for an efficient offline phase. As part of our construction we develop many building blocks of independent interest. These include a new approximation technique for the sigmoid function, which results in a secure evaluation protocol with better communication; secure spline evaluation and secure powers computation protocols for fixed-point values; and a new comparison protocol that optimizes online communication. We also present a new two-party protocol for generating keys for distributed point functions (DPFs) over arithmetic sharing, where previous constructions do this only for Boolean outputs. We implement our protocol in an end-to-end system and benchmark its efficiency. We can securely evaluate a sigmoid in 18 ms online time and 0.5 KB of online communication. Our system can train a model over a database with 4200 samples and 5000 features with online communication of 30 MB and online time of 9 minutes at the cost of 0.52c over WAN. These benchmarks amount to reducing online communication over state of the art by $\approx 10\times$ for sigmoid and $\approx 38\times$ for logistic regression training.**

1. Introduction

One of the most ubiquitous ways to understand and use large amounts of data is to train models which capture the most significant general properties of the underlying data. In many settings the dataset used for the model training is owned by different parties that have agreed to cooperate and create a common model across their datasets but do not want to share record level data. Secure multi-party computation (MPC) [Yao86], [Gol09] enables distributed processing of their joint data which guarantees that neither party learns anything more about the data than its designated output.

We consider the setting of two party computation (2PC) for secure logistic regression training where each party holds a cryptographic share of the input data. Secure protocols in this setting can be used to enable two parties to train a logistic regression model on their joint data by first secret-sharing their inputs. But they also enable processing of data where neither of the computation parties owns the dataset and the receiver of the output may be a different party, assuming the computation parties are not colluding. The latter setting is relevant in scenarios where the dataset consists of entries collected across a large number of users and no single party could have access to the record-level data. In this scenario the data stewardship is distributed across two parties which are in charge of executing a secure computation protocol for the agreed upon functionality. Apart from keeping the input data confidential from any single entity, this model also restricts the data to a specific use case, which the computing parties have to agree on in advance.

Outsourcing a secure computation to a set of non-colluding servers has been applied in practice several times in the past. The first practical application of MPC, which was used to run a sugar beet auction in Denmark in 2009 [BCD$^+$09], relied on three "virtual auctioneers", Danisko, DKS and SIMAP, who had shares of the inputs of all sellers and bidders and executed an MPC protocol for the auction. A second example is a study that was run by the Estonian government, to test whether students working during studies is correlated with worse performance and dropouts [BKK$^+$16]. This study needed to join tax records with education records which are held by different government entities and are not shared. To do this in a privacy preserving manner they executed an MPC with three parties: the Estonian Information System's Authority, the Ministry of Finance IT center, and the company Cybernetica. The two databases were shared among the three parties who executed an MPC protocol implementing the study methodology.

The two-server setting, which we focus on in this work, was leveraged in the system Prio [CB17] which implements a distributed private aggregation protocol where two non-colluding parties receive shares from individual user devices and compute an aggregate histogram over these inputs. This system was later used by Mozilla Firefox to collect browser telemetry [CG20] where the two aggregators were run by

Mozilla and the Internet Security Research Group (ISRG). The same design underlies the Exposure Notifications Private Analytics (ENPA) system implemented by Google and Apple in their Exposure Notifications system [AG21], where the aggregators are the National Cancer Institute (NCI) at the National Institutes of Health (NIH), and ISRG.

An ongoing effort by Google Chrome, called Privacy Sandbox [Goo22c], is developing privacy preserving measurement APIs to support advertising use cases after the deprecation of third party cookies. One of these APIs, the Attribution API [Goo22b], considers a similar measurement goal, which is to compute aggregate measurement across attributed conversions from all users. Again, an MPC system with distributed data stewardship can be used to perform this kind of measurement [HRK$^+$20].

While the previous two examples show that privately aggregating user data into histograms is useful by itself, the functionality needs for measurement systems go far beyond, and require more complex model training. Here, communication between the two computing parties quickly becomes the most expensive part of the system. For example, while it may be beneficial for privacy to place the two servers into data centers operated by different cloud providers (e.g., AWS and Google Cloud in the case of ENPA[AG21]), this incurs egress charges for all traffic between the two servers, which can be significantly higher than intra-cloud traffic costs. These cloud network costs significantly outweigh computation costs in most settings. Low online communication cost is therefore a crucial design goal for practical secure training protocols.

Logistic Regression. Logistic regression is a tool used for many modeling and measurement settings. It is often used for binary classification and prediction in medical [BTC87], [Fre09], engineering [PD09], and finance [AKAA07] applications. It was the functionality of choice in Criteo's challenge for effective use of some of the privacy preserving APIs proposed by Chrome [Gil21]. While not as powerful as Deep Neural Networks (DNNs), it turns out to still be broadly useful for important applications. Consequently, we pay our attention to it in this work.

Online-offline Computation Model. Our constructions consider the online-offline computation model [EKR18] which aims to minimize the complexity of the protocol that is on the critical path of processing inputs when they become available, by outsourcing some of the computation into an input-independent offline phase which can be executed at any time prior to the online stage. The main metric that we optimize for in our constructions is communication complexity which, as we discussed above, could be a major cost in many cross platform two-party computation settings.

We consider two settings. The first one assumes a trusted offline preprocessing that can be executed centrally. This is relevant in scenarios where there is a party which can be trusted to honestly compute the different types of correlated randomness such as multiplication triples, function secret sharing (FSS) keys, and others. For example, in some scenarios regulator parties might be considered trusted for the purposes of this preprocessing. Another way to think about trusted preprocessing is measurement settings over large numbers of clients, where the offline phase is distributed across the clients each of which evaluates a small amount of the required preprocessing and submits the output together with its data shares to the two computation servers.

The second setting that we address in our protocols does not assume a trusted party for the offline stage and proposes that the offline preprocessing is also generated using secure computation between two computation parties. While it is well-known that MPC can be used to distribute any computation that a trusted party could perform [Yao86], [Gol09], efficiency is a concern. We therefore investigate how to efficiently perform the offline phase of our protocols using MPC, though with a greater emphasis on keeping the online phase as cheap as possible.

Differentially Private Output In our scenario, the two computation parties may reveal the output logistic model to a designated output receiver, or alternatively may hold the model shares and later answer inference queries in a distributed manner. While we are not aware of any attacks that use a logistic regression model to recover the input database, the question of how much information different models reveal about the data used for training is an active research area. Making the output differentially private [DMNS06] is one approach to guarantee that it cannot be used to extract individual records. Thus, we also consider the questions of constructing a distributed protocol for differentially private logistic regression training.

Our Contributions. *New Secure Logistic Regression.* We present new constructions for two party secure logistic regression training over a database that is cryptographically shared between the two parties. Our constructions optimize the online communication cost of existing approaches ($\approx 38\times$ reduction over MP-SPDZ [Kel20]) while maintaining accuracy close to plaintext training. We present two different protocols: the first one optimizes solely for online communication, while the second one trades off some of the efficiency in the online phase for supporting efficient distributed computation in the offline phase. Both constructions can facilitate a differentially private output model.

Accurate Secure Sigmoid. The core technical component in our logistic regression construction is a new protocol for secure sigmoid evaluation on input that is shared between two parties. It uses a new approximation approach for the sigmoid functionality which achieves 10^{-4} error using 20 fractional bits. The final protocol offers improved communication cost for its online phase. This cost is $\approx 10\times$ smaller than the communication of the state-of-the-art sigmoid construction of SIRNN [RRG$^+$21] that uses 16-bit ring while our construction uses 63-bit rings. It is also $\approx 31\times$ better than the online communication for sigmoid in MP-SPDZ [Kel20]. A secure sigmoid evaluation runs between 16 and 19ms in different network setting and includes 0.5-1.18 KB of communication.

Communication Efficient Constant Round Secure Comparison. A main building block for our sigmoid construction is a new comparison protocol for l-bit numbers which uses a new reduction to small bit length comparison, all prefix AND along with inner product that works in *constant* number of rounds for online computation. It uses only three communication rounds in total whereas the state-of-the-art SynCirc uses rounds logarithmic in l. Our protocol further improves the online communication cost $\approx 1.3 - 2.6\times$ over SynCirc. The online communication for 128-bit numbers comparison is only 522 bits. Secure comparison is a core building block in a broad range of functionalities with applications far beyond the constructions in this paper, therefore this construction may be of independent interest.

New Techniques for (i)DPFs. The communication and round efficiency of our constructions comes from leveraging (incremental) distributed point functions ((i)DPFs) [BGI15], [BBCG^{+}21] techniques inside our MPC protocols in new ways. We present a secure computation protocols for equality checking of all prefixes of a shared number using iDPFs, which achieves the round efficiency of our new comparison protocol. We also present a new construction for two-party generation of distributed point function (DPF) keys with arithmetically shared output values, which is used for distributed offline preprocessing.

Efficient Constructions with Fixed-Point Inputs. The accuracy of our computation relies on fixed-point representation of shared values. We present new constructions for spline evaluation and secure powers computation with fixed point input representation. These techniques are used for secure comparison and secure Taylor approximation where existing approaches work only for integers.

Implementation and Evaluation. We present end-to-end implementation of our protocols, and to our knowledge we are the first implementation that combines FSS-based and secret-sharing-based techniques. We evaluate the costs of our protocols including microbenchmarks for our building blocks such as a new secure comparison and sigmoid evaluation, as well as an end-to-end evaluation of secure logistic regression training. We can train a model over cryptographically shared data of 4200 samples with 5000 features in about 9min with 30MB of communication, which amounts to 0.52c cost. We achieve accuracy very close to the plaintext trained model (less than 1% difference). We reduce online communication over state of the art by $\approx 2\times$ for comparison, $\approx 10\times$ for sigmoid and $\approx 38\times$ for logistic regression training.

1.1. Our Approach

Secure Logistic Regression. (Section 3) Our construction uses stochastic gradient descent (SGD), which is an iterative training algorithm. Each iteration for the model update consists of matrix operations and a sigmoid evaluation.

Secure Sigmoid Evaluation (Sections 4 and 5). We introduce a new construction for secure sigmoid evaluation where the input is shared between two parties. It leverages a new approximation method for the sigmoid function that relies on three different approximation functions for different input intervals. In particular, for the input interval $[0, 1)$, we use spline approximation which splits the interval in several pieces, each of which is approximated with a linear function. For the interval between 1 and a configurable threshold we use Taylor approximation. For large values above the threshold we approximate the sigmoid value with 1. Negative inputs are handled symmetrically.

To reduce communication of the online phase of our protocol we rely on techniques for function secret sharing [BGI15] which enable non-interactive computation. In particular we use the multiple interval containment (MIC) gate [BCG^{+}21] to identify which interval the input falls into in order to use the approximation function. We also use the MIC gate within the spline approximation on the interval $[0, 1)$ to choose the right linear function.

Distributed Comparison Function (Appendix D). MIC gates leverage distributed comparison functions (DCFs) [BCG^{+}21] which rely on function secret sharing [BGI16] techniques. We introduce a new reduction from DCFs to incremental distributed point functions (iDPFs) [BBCG^{+}21], which is conceptually simpler than the previous construction by [BCG^{+}21].

Secure Powers Computation with Fixed-Point Representation (Section 5.2) A sigmoid is computed as $1/(1 + e^{-x}$. Our sigmoid approximation for values above 1 has two main components: secure exponentiation for evaluation of $r = e^{-x}$ followed by a secure protocol for powers computation that enables the polynomial evaluation for the Taylor series for $1/(1 + r)$. For the first part we leverage the construction for secure exponentiation of Kelkar et al. [KLRS22]. For the second part we present a new construction inspired by the HoneyBadgerMPC secure powers computation protocol [LYK^{+}19], which we extend to work with fractional values in fixed-point representation.

Online-Offline Balanced Protocol (Section 6). The most significant part of the offline computation for our protocol is the generation of the FSS keys, which are needed for the MIC gates. In the setting without trusted preprocessing these keys need to be generated using two party computation, presenting significant costs that challenge the feasible execution of the offline computation. Existing approaches either rely on general-purpose MPC, which in this case is expensive due to the need for secure evaluation of a PRG, or they use the Doerner-Shelat technique [Ds17], which requires computation exponential in the input size. In the application of the MIC gate for the spline approximation this is not an issue for us because the inputs can be made short by truncation, leveraging the fact that the input is a fixed point number with absolute value ≤ 1 for the portion of the input domain on which we perform the spline evaluation. However, in the higher level interval containment functionality which identifies which type of approximation needs to be used, this is no longer the case, since we don't have any simple way to reduce the size of the input. This means that we would need an FSS gate with a large input domain, which would have extremely high offline computation.

Secure Comparison (Section 6.1). To overcome this challenge we modify the protocol to use a secure comparison functionality instead of MIC to determine the first level of input partitions. We introduce a new comparison construction with a highly communication-efficient, constant round online phase while only having modest computation complexity. It relies on a new reduction from n-bit numbers comparison to comparison on single bits using functionality that computes the AND over the bits in all bit prefixes of a number. We present a new non-interactive construction for the latter functionality where the input is split input among two parties, which uses iDPFs. The resulting protocol improves over the online communication of Rathee et al.'s CrypTFlow2 [RRK$^+$20a] by $\approx 2.4\times$ and Couteau [Cou18] by $\approx 4.1\times$ for 64-bit inputs and appropriate parameters. We reduce the number of communication rounds by similar factors, i.e., from 6 and 12 rounds respectively to 3.

Secure comparison is a fundamental building block for higher-level privacy-preserving applications. Couteau [Cou18] present an extensive list of such applications including oblivious sorting, database search constructions, private set intersection, oblivious RAM, machine learning for applications such as classification, feature extraction, and generating private recommendations.

Secure DPF Key Generation (Appendix E). One of the main components for the offline computation in our construction is the generation of FSS keys for distributed point functions (DPFs). Doerner and Shelat [Ds17] present a two party computation protocol for the DPF key generation algorithm of Boyle et al. [BGI16]. However, this protocol only supports DPFs with boolean (that is, XOR-shared) output groups.

Many applications relying DPFs including the works of Boyle et al. [BCG$^+$21], however, require arithmetically-shared output groups. While [BCG$^+$21] refer to [Ds17] in their paper, the construction does not immediately generalize to that setting without additional modifications. One solution is using DPFs with boolean output shares, and then use a share conversion protocol (for example the one from CrypTFlow2 [RRK$^+$20b]). However, this usually requires a number of oblivious transfers proportional to the size of the output shares, which increases the online round and communication complexity

We present a new two-party computation protocol for the DPF key generation algorithm which maintains the property that the evaluation of the DPF keys generate arithmetic shares of the output. Our construction only requires a single additional oblivious transfer in the offline phase, independent of the size of the output shares.

1.2. Related Work

There is an extremely large number of works in the field of secure learning, differing across several dimensions. These dimensions include the type of model being computed (linear/logistic/poisson regression, deep neural nets), the way data is distributed across parties (secret-shared, vertically partitioned, horizontally partitioned, federated), and the type of security provided (differentially private with central or local DP, MPC with semi-honest, malicious security, honest majority, and so on) [MZ17], [KVH$^+$21], [ASSKG19], [RRK18], [WTB$^+$20], [MZ17], [KLRS22], [Kel20], [PSSY21a], [RRG$^+$21], [BLL21], [CKP22], [RMY18], [GSB$^+$17], [DN04], [YLC$^+$19], [BEG$^+$19].

We focus our discussion on the state-of-the-art works that best match our setting and offer informative comparisons. Specifically, we focus on works that compute logistic regression using secure multi-party computation in the semi-honest setting, with data secret-shared among the computing parties. We restrict ourselves to works that have 2 or 3 servers only. The number of works in this setting is relatively manageable, and we group them into 3 categories in our discussion: those with a highly accurate sigmoid approximation, those with a coarse sigmoid approximation, and those based on homomorphic encryption.

For works with highly accurate sigmoid approximations, we restrain our detailed comparison to 2 key works, MP-SPDZ [Kel20] and SIRNN [RRG$^+$21], which are the state of the art in this area, to our best understanding. Our key difference with both of these works is our approach to sigmoid computation. MP-SPDZ takes the approach of computing exponentiation and division in MPC, which results in a large number of rounds. SIRNN does essentially the same, using novel protocols for each. They use a clever approach of adjusting the fixed-point precision to reduce the costs, and introduce a novel Lookup-Table based exponentiation together with a novel adaptation of Goldschmidt's division. However, both these approaches end up taking a large number of rounds ($\approx$ dozens) due to complexity of division and exponentiation in MPC. Our key improvement is to achieve high accuracy with a constant number of rounds and less communication using a combination of FSS primitives and a customized sigmoid approach with different approximations computed over different intervals. A detailed experimental comparison can be found in Sections 7.1 and 7.2.

There are also works such as SecureML [MZ17] and ABY2.0 [PSSY21a] which use a much coarser sigmoid approximation. As a result, both works have worse accuracy on logistic regression. We are focused on preserving accuracy of logistic regression, and so do not engage in a detailed comparison with these works. We observe briefly that these works rely mainly on AND gates and secure comparisons, and since we propose an improved secure comparison approach, replacing the secure comparisons in [MZ17], [PSSY21a] with those in our work is likely to offer an improvement. This is discussed further with concrete accuracy numbers in Section 7.2.

There are several works which leverage homomorphic encryption in order to compute Logistic Regression, for example [BLL21] and [CKP22]. These works are interesting because they do not require interaction between parties:

one party performs the computation on the entire encrypted dataset. However, this approach comes with a large computational and communication overhead compared to MPC-based approaches (where the communication means the size of the initial encrypted datasets). We see this as a significantly different approach and setting, and so we do not perform a detailed comparison with them in the evaluation section. This is consistent with the approach taken by the MPC-based works we cite above.

Separately from logistic regression and sigmoid computation, there are a several related works focused on developing Function-Secret-Sharing or adjacent primitives [BGI19], [BCG+21], [PSSY21a], and using them for machine learning [RTPB20]. Our work is influenced by several of these papers, and our techniques can be seen as building on theirs, specifically by combining them with secret-sharing-MPC based approaches.

Another of our key contributions is a new secure comparison. This is rich area of research [Cou18], [GSV07], [RRK+20b], [BCG+21], [DSZ15], [PSSY21b], [PSSY21a], since comparison is critical for the nonlinear computations in machine learning tasks. We go over the most relevant related works directly in Section 6.1 where we present our new secure comparison protocol. We explain how our approach differs from and builds on these works.

Finally, we discuss an important concurrent work, LLAMA [GKCG22], which builds on SiRnn [RRG+21] by using FSS to reduce the cost of comparisons in the online phase, similarly to us. Since the work is concurrent, we do not provide a detailed comparison in the evaluation section, but describe the key differences here. One difference lies in the way LLAMA computes sigmoid. While we use a custom approximation to the sigmoid function, LLAMA uses a single spline gate that computes a degree-2 polynomial over each interval, using enough intervals so as to to reduce their chosen accuracy metric (ULP error) is small (≤ 4). They use the spline gate from [BCG+21], and also run into its key limitation: this spline construction only works on integer values, because the underlying polynomial evaluation does not perform truncated multiplication needed for fixed point values. The LLAMA authors get around this by assuming the ring is large enough to accommodate untruncated multiplication, and then perform a truncation in a single separate round. In contrast, our work gives a new protocol for computing polynomials over secret-shared fixed point numbers that implicitly handles truncation. In this way, we can work with smaller rings, thereby gaining efficiency. We note that our fix only works for fixed-point numbers that have no integer part (i.e. have absolute value < 1), but this turns out to be fine for our use: we only use polynomial approximations for a fixed region of the input where this condition holds.

2. Preliminaries

Notation. Given a finite set S, $x \leftarrow S$ indicates that an element x is sampled uniformly at random from S. For any positive integer n, $\mathbb{Z}_n$ denotes the set of integers modulo n. $[k]$ denotes the set of integers $\{1, \ldots, k\}$. We use $\mathbf{1}\{b\}$ to denote the indicator function that outputs 1 when b is true and 0 otherwise. λ indicates computational security parameter. For a vector $\mathbf{v}$, $\mathbf{v}_{i\ldots j}$ denotes the vector with elements $v_i, \ldots, v_j$. Likewise, for a matrix M, $M_{i\ldots j}$ denotes the matrix containing rows i through j from M.

Fixed-Point Representation. A fixed-point representation is parameterized by a tuple $(\mathcal{R}, w, s, \mathsf{Fix})$ where $\mathcal{R}$ is a ring, w represents the bitwidth, s represents the scale (or the fractional bitwidth), and $\mathsf{Fix} : \mathbb{R} \to \mathcal{R}$ is a function mapping $x \in \mathbb{R}$ to its fixed-point representation $\widehat{x} \in \mathcal{R}$. In this work, we will work over the ring $\mathbb{Z}_L$ where $L = 2^l$ and $s \leq w < l$. Similar to previous works, we define our mapping function $\mathsf{Fix}(x) = \lfloor x \cdot 2^s \rfloor \bmod L$. In this mapping, all real numbers having absolute value atmost 2^{w-s} have a corresponding fixed-point representation in the ring. Specifically, non-negative real numbers are mapped in the range $[0, 2^w)$ whereas negative real numbers are mapped in the range $(L - 2^w, L)$ in their two's complement representation. Let $\mathcal{R}^* = [0, 2^w) \cup (L - 2^w, L)$ denote the part of the ring where fixed-point numbers are represented. Note that two distinct real values might have the same fixed point representation because of the limited fractional bitwidth. We will use $\widetilde{x}$ to denote the corresponding real-value for a fixed-point value x. We use $\mathcal{R}_{min}$ and $\mathcal{R}_{max}$ to denote the maximum negative and maximum positive values representable in $\mathcal{R}$.

Secure Computation. Secure computation protocols enable functionalities where parties can compute a function on their joint private inputs in a way that guarantees only the output of the computation is revealed. Our protocol constructions are in a two-party setting and provide semi-honest security [Gol09], i.e., the parties are assumed to follow the prescribed protocol. We denote the two parties by P_0 and P_1. The protocol may be divided into an offline preprocessing phase (independent of parties' inputs) and an online phase that depends on parties' inputs. The offline preprocessing may be performed by a trusted third party, or by the parties executing an MPC protocol. Due to space constraints, we do not provide formal proofs of security of our protocols. In the semi-honest model, the security of our protocols follows directly from the security of the underlying primitives.

Secret Sharing. We use $[\![x]\!]^{\mathcal{R}}$ to denote an additive sharing of x in ring $\mathcal{R}$. We drop the superscript $\mathcal{R}$ when it is clear from context. We write $[\![x]\!] = ([\![x]\!]_0, [\![x]\!]_1)$ to denote that P_0 and P_1 get shares $[\![x]\!]_0$ and $[\![x]\!]_1$ respectively, such that $[\![x]\!]_0 + [\![x]\!]_1 = x$ in $\mathcal{R}$. An additive sharing is random if $[\![x]\!]_0$ and $[\![x]\!]_1$ are uniformly distributed in $\mathcal{R}$ subject to $[\![x]\!]_0 + [\![x]\!]_1 = x$. When we discuss additive shares, we generally mean random additive shares. Additive shares are also called arithmetic shares. Analogously, we use $\langle b \rangle$ to denote a random XOR-sharing of a bit $b \in \{0, 1\}$, consisting of bits $\langle b \rangle_0$ and $\langle b \rangle_1$) such that $\langle b \rangle_0 \oplus \langle b \rangle_1 = b$.

Truncation. Suppose parties are holding additive-sharing of a fixed-point value $\widehat{x}$ where the scale is s bits. Then they can use $\mathcal{F}_{\mathsf{truncate}}$ to reduce the scale to s' bits where $0 \leq s' \leq s$. An efficient instantiation of $\mathcal{F}_{\mathsf{truncate}}$ was described in SecureML [MZ17]: Suppose x_0 and x_1 are the shares of $\widehat{x}$ held by party P_0 and P_1 respectively. Then, in order to perform truncate operation, both parties can just locally truncate the last $s - s'$ bits of their individual shares to get new shares x'_0 and x'_1. Let x' denote the true truncated value of x after truncating the last $s - s'$ bits. SecureML [MZ17] shows that $\mathsf{Recon}(x'_0, x'_1) \in \{x' - 1, x', x' + 1\}$. In other words, this non-interactive truncation protocol incurs a small error in the least significant bit of the fractional part of the FXP value. For our purposes, this error will be tolerable as the FXP representation itself admits an error in the least significant bit of the fractional part compared to the actual real value.

2.1. Logistic Regression

Logistic regression is a probabilistic classifier and a supervised learning algorithm [JM09]. The classification function f takes an observation, which is a vector of features $\vec{x}_i$, and outputs the class y with highest likelihood. It leverages the sigmoid functionality $\sigma(z) = \frac{1}{1-e^{-z}}$ to assign probability determining the class to an input feature vector $\vec{x}$, using the weight vector $\vec{w}$ and a bias term b, which form the model. More specifically, $\sigma(\vec{(x)} \cdot \vec{w} + b)$ outputs the probability of mapping $\vec{x}$ to the class 1.

The learning process for logistic regression takes a set of labeled training samples $(\vec{x}_i, y_i)$ and aims to learn parameters $\vec{w}$ that make the predictions y'_i as close as possible to the true labels y_i. This is done by minimizing the (regularized) cross-entropy loss function $\mathcal{L}_{\mathsf{CE}}(y, y') = -(y \log y' + (1-y) \log 1 - y')$ which measures the distance between predicted and true value.

Stochastic gradient descent computes optimal weights $\mathbf{w}$ by minimizing the average loss over the n training samples:

$$\tilde{\mathbf{w}} = \mathsf{argmin}_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{\mathsf{CE}}(f(\mathbf{x_i}, \mathbf{w}), y_i).$$

This is done by computing the gradient $\mathbf{g_i} \leftarrow \nabla_w \mathcal{L}_{\mathsf{CE}}(f(\mathbf{x_i}, \mathbf{w}), y_i)$ of the loss function on a random batch of B training points. The model is then updated as $\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum\limits_{i \in [B]} \mathbf{g}_i$.

In the context of secure computation protocols we will run a fixed number of iterations to avoid leakage about the private samples based on the time for convergence. The mini-batch technique makes each iteration over a subset of the samples rather than the whole batch.

In practice, the regularized cross-entropy loss is often used:

$$\mathcal{L}_{\mathsf{CE}}(y, y') = -(y \log y' + (1-y) \log 1 - y') - \frac{\lambda}{2} ||w||^2$$

The regularization parameter λ guides the model towards weights with smaller magnitude, which reduces overfitting in practice.

2.2. Multiplication Triples

Suppose parties hold additive-shares of values $x, y \in \mathcal{R}$. Then can use a functionality $\mathcal{F}_{\mathsf{Mult}}$ to get additive shares of $z \in \mathcal{R}$ such that $z = x \cdot y$. In the pre-processing model, $\mathcal{F}_{\mathsf{Mult}}$ can be efficiently realized by generating beaver triples in the offline phase, and then consuming them in the offline phase. This incurs an online communication of 2 ring elements per-party.

For multiplying an $n \times m$ matrix $\mathbf{X}$ with another $m \times k$ matrix $\mathbf{Y}$, there is a special matrix multiplication protocol based on *matrix* beaver triples which incurs an online communication of $2(nm + mk)$ per party. We will use $\mathcal{F}_{\mathsf{matMult}}$ to abstractly represent a functionality which enables multiplication of two additively shared matrices.

For multiplying a $n \times m$ matrix $\mathbf{X}$ with a sequence of matrices $\{\mathbf{Y_i}\}_{i \in [n]}$ where $\mathbf{Y_i}$ has dimension $m \times k_i$, there exists another optimization based on *correlated* matrix beaver triples and incurs an online communication of $2(nm + m \sum_i k_i)$ per party. We will use $\mathcal{F}_{\mathsf{corrMatMult}}$ to represent this functionality.

These functionalities can be extended to real numbers represented in fixed-point format by adding an additional protocol for truncation at the end, where s least significant bits are truncated from the result in order to adjust the fractional scale. In our work, we use the non-interactive truncation protocol from SecureML [MZ17] described in Section 2.

2.3. Two Party Computation Functionalities

Oblivious Transfer We will use $\mathcal{F}^k_{\mathsf{OT}}$ to denote the two-party 1-out-of-2 chosen input oblivious transfer functionality, where the sender's input to $\mathcal{F}_{\mathsf{OT}}$ are two strings $m_0, m_1 \in \{0,1\}^k$ and the receiver's input is a choice bit $c \in \{0,1\}$. The receiver obtains m_c as output from $\mathcal{F}^k_{\mathsf{OT}}$ whereas the sender has no output. In the pre-processing model, one can generate random OT (ROT) correlation which consists of (r_0, r_1, b) where $r_0 \in \{0,1\}^k, r_1 \in \{0,1\}^k, b \in \{0,1\}$ and distribute this across the two-parties. The sender gets (r_0, r_1) whereas the receiver gets (b, r_b). In the online phase, these ROT correlations can be consumed to realize $\mathcal{F}^k_{\mathsf{OT}}$ efficiently.

Multiplexer Following [RRK+20b], we will use $\mathcal{F}_{\mathsf{MUX}}$ to denote a multiplexer functionality. Suppose parties hold arithmetic shares of x and Boolean sharing of a selection bit b. Then they can use $\mathcal{F}_{\mathsf{MUX}}$ to get an arithmetic sharing of x if $b = 1$, and arithmetic sharing of 0 otherwise. A protocol for $\mathcal{F}_{\mathsf{MUX}}$ can be realized using 2 (simulataneous) OTs. In some scenarios, a variant of $\mathcal{F}_{\mathsf{MUX}}$, denoted by $\mathcal{F}_{\mathsf{MUX2}}$, might be more useful. It takes arithmetic shares of x_0 and x_1, along with boolean sharing of a selection bit b, and outputs a fresh sharing of x_0 if $b = 0$, and a fresh arithmetic sharing of x_1 otherwise. A protocol for $\mathcal{F}_{\mathsf{MUX2}}$ can be realized using a single call to $\mathcal{F}_{\mathsf{MUX}}$ as follows: Parties locally compute a sharing of $x_1 - x_0$, invoke $\mathcal{F}_{\mathsf{MUX}}$ on it using the share of bit b, and finally locally add the sharing of x_0 to their output from $\mathcal{F}_{\mathsf{MUX}}$.

2.4. Function Secret Sharing

We use Boyle et al.'s definition of function secret sharing (FSS) [BGI15]. A 2-party FSS is an algorithm that efficiently splits a function f into two additive shares f_0 and f_1. These shares must satisfy the following two properties: (1) f_i hides f and (2) $f_0(x)+f_1(x) = f(x)$ for every input x. Output reconstruction in (2) is *additive*. Formally:

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

- $\mathsf{Gen}(1^\lambda, \widehat{f})$, where $\widehat{f}$ is a description of a function f, outputs a pair of keys (k_0, k_1). $\widehat{f}$ explicitly includes the input group description $\mathbb{G}^{in}$ and the output group description $\mathbb{G}^{out}$.
- $\mathsf{Eval}(b, k_b, x)$, given party index b, a key k_b defining $f_b : \mathbb{G}^{in} \to \mathbb{G}^{out}$ outputs $f_b(x) \in \mathbb{G}^{out}$.

Secure Computation via FSS. [BGI19] showed that the FSS paradigm can be used to efficiently evaluate some function families in 2PC in the preprocessing model, where Gen and Eval correspond to the offline and online phase, respectively. Note that unlike in secret-sharing MPC, the FSS inputs and outputs are *public* whereas the function is *secret-shared*. As the parties cannot learn the values on any intermediate circuit wires, the protocol needs to take care to use masked inputs and outputs for each gate $g : \mathbb{G}^{in} \to \mathbb{G}^{out}$. That is, the input is masked with r^{in} and the output with r^{out}. Then, each gate g implements an offset function $g^{[r^{in}, r^{out}]}(x) = g(x - r^{in}) + r^{out}$. Hence, each gate first unmasks the input x and only then executes the function g. The output of g is masked with r^{out} prior to reconstruction. This step is repeated for each gate until both parties evaluate the last circuit gate. Given the output mask of the last gate, P_0 and P_1 can learn the final circuit output without learning any intermediate wire values. Additional definitions and useful preliminaries appear in Appendix A.

3. Secure Logistic Regression

We aim to develop concretely-efficient secure two-party computation protocols for logistic regression training, focusing on online communication and rounds. Like previous works [MZ17], [SGRP19], we leverage arithmetic secret-sharing (see Section 2) and train the model with stochastic gradient descent (SGD).

Our protocol is described in Algorithm 1. It makes heavy use of correlated matrix-vector multiplication using Beaver triples, and crucially depends on an implementation of the sigmoid function in MPC.

Our novel contributions lie in the construction of the sigmoid protocol $\mathcal{F}_{\mathsf{Sigmoid}}$ using a mix of MPC primitives including DCFs, DPFs, Taylor approximation, and an efficient secure exponentiation protocol.

4. Secure Sigmoid

The key challenge of computing a single step of SGD is evaluating real-valued sigmoid function. It requires computing (1) exponentiation of a public base to a secret exponent as well as (2) division by a secret divisor:

Algorithm 1: Logistic Regression Protocol

```
Public inputs: Number of iterations $T$, dataset dimensions $n, k$, batch size $B$, learning rate $\alpha$, regularization parameter $\lambda$.
Private inputs: Secret-shared dataset $[\![X]\!] \in R^{n \times k}$ and labels $[\![\mathbf{y}]\!] \in R^n$.
1  Let $[\![\mathbf{w_0}]\!]$ be the initial secret-shared model with arbitrary weights.
2  for $t = 1$ to $T$ :
3      for $b = 1$ to $\lfloor n/B \rfloor$ :
4          $i \leftarrow (b-1) \cdot B + 1$
5          $j \leftarrow \min(n, b \cdot B)$
6          $[\![X_B]\!] \leftarrow [\![X_{i \ldots j}]\!]$
7          $[\![\mathbf{u}]\!] \leftarrow \mathcal{F}_{\mathsf{corrMatMult}}([\![X_B]\!], [\![\mathbf{w_{t-1}}]\!])$
8          $[\![\mathbf{s}]\!] \leftarrow \mathcal{F}_{\mathsf{Sigmoid}}(\mathbf{u})$
9          $[\![\mathbf{d}]\!] \leftarrow [\![\mathbf{s}]\!] - [\![\mathbf{y}_{i \ldots j}]\!]$
10         $[\![\mathbf{g}]\!] \leftarrow \mathcal{F}_{\mathsf{corrMatMult}}([\![X_B{}^\top]\!], [\![\mathbf{d}]\!])$
11         $[\![\mathbf{w_t}]\!] \leftarrow [\![\mathbf{w_{t-1}}]\!] - (\alpha/B) \cdot ([\![\mathbf{g}]\!] + \lambda \cdot [\![\mathbf{w_{t-1}}]\!])$
12 return $[\![\mathbf{w_T}]\!]$.
```

$$S(x) = \frac{1}{1+e^{-x}}$$

Division is sometimes approximated via Goldschmidt's [Gol64] method, which is expensive. Alternatively, exponentiation can be approximated either by decomposing the exponent into bits [DFK$^+$06], which is costly, or via low-degree polynomials [AS19], which is inaccurate.

In this section, we present our sigmoid functionality (Algorithm 2) which is actually the sigmoid approximation we achieve. We describe how we securely implement this approximate functionality, pointing to Section 5.1 and Section 5.2 for detail on how we implement the more complex components of our functionality.

4.1. Sigmoid Approximation

As can be seen from Figure 4 in the appendix, the sigmoid function is 'symmetric' around the y-axis. More specifically, $S(x) + S(-x) = 1$ for all $x \in R$. This implies we can focus on evaluating $S(x)$ and then compute $S(-x) = 1 - S(x)$ locally.

For $x \geq 0$, we need to compute both division and exponentiation in MPC. First, we demonstrate how we bypass directly computing division.

Note that $\frac{1}{1+e^{-x}}$ is in the form $\frac{1}{1+r}$. Hence, we can apply d-degree Taylor series approximation:

$$\frac{1}{1+r} = 1 - r + r^2 - r^3 + \ldots + r^d$$

This approximation requires to compute additions and powers of r. As a result, it can be expressed as an arithmetic circuit, and thus is MPC-friendly. While addition is a virtually free local operation, computing powers is an expensive interactive operation. We present a concretely efficient protocol for computing powers in Section 5.2 based on the protocol of [LYK$^+$19]. Our protocol computes *all* powers of r (irrespective of the degree) in only 2 communication rounds.

However, this approximation works well only when $r << 1$. We therefore use this approximation only on the interval $[0, \frac{1}{e}]$. As $r = e^{-x}$, we use this technique when $x \geq 1$. In order to compute e^{-x}, we use the 1-round exponentiation technique of [KLRS22]. We note that the exponentiation protocol from [KLRS22] assumes a known (arbitrary) bound on how negative the exponent can be. So in order to comply with that assumption, we do not use this exponentiation protocol if the exponent is too negative. Rather, we just set the sigmoid output directly to 1. We fix the bound as $l_f/\log_2(e)$, i.e. whenever $x \geq l_f/\log_2(e)$, we set the sigmoid output to 1. This bound can be justified by observing that for any $x \geq l_f/\log_2(e)$, $e^{-x} < 2^{-l_f}$. Hence the fixed point representation of the result of exponentiation is exactly 0 in this case.

Now, it remains to explain how we evaluate sigmoid for $x \in [0, 1)$. We evaluate a spline defined piecewise by lines via the FSS spline gate as explained in Section 5.1.

Importantly, neither party should learn which technique is used to compute sigmoid (i.e. in which interval x belongs). Thus, all evaluations are run simultaneously. At the end, the right output is obliviously selected and fresh secret shares are output to each party.

Algorithm 2: Approximate Sigmoid

```
Parameters:
Let $m$ be the number of lines defining a spline.
Let $l_f$ be the number of fractional bits.
Let $d$ be the degree of Taylor series approximation.

Private input:
Let $x \in R$ be the sigmoid input.

Sigmoid($x$) :
1  if $x < 0$ then
2      $S \leftarrow 1 - \mathsf{Sigmoid}(-x)$.
3  else
4      if $x < 1$ then
5          $S \leftarrow \mathsf{Spline}(m, [0, 1))$
6      else
7          if $x \log_2(e) \geq l_f$ then
8              $S \leftarrow 1$
9          else
10             $r \leftarrow e^{-x}$
11             $S = \frac{1}{1+r} \leftarrow 1 - r + r^2 - \ldots \pm r^d$
12 return $S$
```

5. Secure Sigmoid with Trusted Offline Setup

In this section, we describe details of our approach for *securely* computing the sigmoid approximation described in Algorithm 2 with a focus on minimizing the online communication complexity. Towards that end, we assume that the offline phase is part of a trusted setup phase. In practical settings, such a trusted setup can be performed by a trusted third party. Another possibility, when the intermediate models are protected by DP (see Algorithm 10 in Appendix F), is to outsource the setup phase to (semi-honest) clients. These clients may provide a portion of the precomputed setup alongside the inputs they upload to the two MPC parties. In case a trusted setup is infeasible, we discuss how to perform the offline phase in MPC as well in Section 6.

Our sigmoid approximation will work by first using $\mathcal{F}_{\mathsf{MIC}}$ to determine if the shares of the input x lie in the range $[0, 1), [1, \frac{1}{\log_2(e)}), [\frac{1}{\log_2(e)}, \infty)$, or the negative equivalents of these ranges. $\mathcal{F}_{\mathsf{MIC}}$ yields arithmetic shares of 1 if x was in that range, and arithmetic shares of 0 otherwise. In parallel, we compute the sigmoid approximations on each range using the tailored technique for that range described above (spline-approximation, exponentiation-and-Taylor-Approximation, or hardcoding), using the $S(-x) = 1-S(x)$ identity for the negative intervals. We then compute a dot product of the outputs of $\mathcal{F}_{\mathsf{MIC}}$ with the outputs of the tailored sigmoid computations to "select" the output of sigmoid on x using the approximation corresponding to the interval in which x lies. This dot product can be computed using standard Beaver multiplication [Bea92].

In the following sections we discuss how to build the tailored sigmoid implementations for each interval.

5.1. Secure Spline Computation

A spline is a special function defined piecewise by polynomials. Formally, a spline function $S : \mathbb{R} \to \mathbb{R}$ on an interval $[a, b)$ is specified as a partition of m intervals $\{a_i, b_i\}_{i \in [m]}$ with a d degree polynomial p_i defined for each of the intervals. The value of the function S on input $x \in [a, b]$ is equal to $p_i(x)$ where $a \leq x < b$. For our specific use-case of sigmoid approximation, we use degree 1 polynomials on m intervals. Note that such a polynomial $Q(x)$ is of the form $Q(x) = ax + b$ where a, b are publicly known values. Given a secret-sharing of x, parties can locally compute a sharing of $Q(x)$. Note that when computing Q over fixed-point input x, we need to perform a truncate operation on the product ax before adding it to b. This can be performed using the non-interactive truncation protocol described in Section 2.

For constructing a spline protocol, we will let the parties locally evaluate degree 1 polynomials Q_i defined for each of the m intervals. Let $\vec{Q}$ represent a length m vector containing the result of evaluating Q_i on x for each of the m intervals. Now, parties can use MIC gate described earlier to generate shares of a vector $\vec{B} = [b_1, b_2, \ldots, b_m]$ where $b_i = \mathbf{1}\{p_i \leq x \leq q\}$. Finally, they can take a dot-product between $\vec{Q}$ and $\vec{B}$ to derive the actual spline result. Such a dot-product can be securely implemented using a single call to $\mathcal{F}_{\mathsf{matMult}}$. Thus, the total communication cost of securely evaluating a spline is $2 + 4m$ elements of communication. This can be performed in 2 online rounds where the first round is used for MIC gate evaluation and the second round is used for $\mathcal{F}_{\mathsf{matMult}}$. In Appendix I , we describe an optimized protocol for performing the dot-product (for the specific case of splines) which reduces the overall communication of spline to just 6 elements of communication. Crucially, this optimization makes the online communication cost of spline independent of the number of intervals m.

Note that in our Sigmoid Approximation described in Algorithm 2, we use spline only when the input is between $[0, 1)$. This means that the spline protocol only needs to be executed on the fractional bits of the input. In other words, given a positive fixed point input x, let $y = x \bmod 2^s$. It is easy to see that y represents the fractional bits of x. In the secret-shared setting, parties can locally compute $[\![y]\!] := [\![x]\!] \bmod 2^s$ to derive a sharing of the fractional bits of x in the smaller ring $\mathcal{R} = \mathbb{Z}_{2^s}$. Now parties can use (shares of) y for evaluating the MIC component of the spline protocol, thus reducing the domain size of the MIC from l bits to s bits. This observation will be needed later in Section 6. If we set the output domain of MIC to be $\mathbb{Z}_{2^l}$ and compute the $\vec{Q}$ over $\mathcal{R} = \mathbb{Z}_{2^l}$, we can ensure that the final output of spline protocol is shared in $\mathcal{R} = \mathbb{Z}_{2^l}$ to be compatible for further computations.

5.2. Secure Powers Evaluation

To evaluate a Taylor series approximation inside MPC, we need a procedure to securely compute a d-degree polynomial which, in turn, requires computing the (secret-shares of) consecutive powers $\{x, x^2, \ldots, x^d\}$ for a (secret-shared) input x. Naively, one could invoke $\mathcal{F}_{\mathsf{Mult}}$ repeatedly d times in order to generate these powers. However, this makes the communication-cost proportional to the degree d. In [LYK+19], the authors proposed a novel protocol to generate all d powers using a single element of online communication per party, where the masked value $x_{\mathsf{mask}} = x - r$ is revealed. The protocol leverages a new type of offline pre-processing correlation called "random powers". In such a correlation, parties have a sharing of $\{r, r^2, \ldots, r^d\}$ for a uniformly random $r \in \mathcal{R}$. For a (secret-shared) input x in the online phase, the parties "consume" these special correlations in order to generate a sharing of $\{x, x^2, \ldots, x^d\}$. The main observation in the protocol is the following relationship:

$$[\![x^i r^j]\!] = [\![r^{i+j}]\!] + x_{\mathsf{mask}}\Big(\sum_{l=0}^{i-1}[\![x^{i-1-l}r^{j+l}]\!]\Big) \qquad (1)$$

The aforementioned protocol works only for integer inputs (mapped to ring elements in the natural way) and it is unclear how to directly extend it to inputs represented in fixed-point format. The main challenge is that Equation 1 now needs to be evaluated over real numbers instead of ring elements in order to get the correct result. We observe that emulating the evaluation of Eq. 1 over reals inside a ring requires the following: (i) Performing fixed point multiplications instead of ring multiplication (i.e. we need to perform a truncation operation after every ring multiplication to adjust the scale [1]), (ii) Ensuring that none of the intermediate values in the computation wrap around the ring, since a multiplication wrapping prior to truncation corrupts the share. While incorporating the first condition into Eq. 1 might seem straightforward, it is less obvious how to incorporate the second condition. The reason is that the term r^{i+j} will almost always wrap around the ring when r is sampled from the fixed-point region of the ring. Note that this wraparound is not an issue when we want to evaluate Eq. 1 over integers.

We observe that in our specific use-case of sigmoid evaluation, the input x to the powers protocol is of the form e^{-z}. As we have already discussed that considering only $z \geq 0$ suffices for sigmoid evaluation (due to its symmetric nature), this means that we can assume that x is always a real numbered value between $(0, 1]$.

With this observation in place, we are able to incorporate condition (ii) mentioned earlier in the following way: Instead of sampling r from the entire fixed-point region of the ring, we sample it only from the region representing real numbers between $[0, 1)$. While this ensures that the fixed point representations of powers of r don't wrap around the ring, it creates another issue: Revealing the (fixed-point representation of) masked value x_{mask} is no longer secure. The reason is that the distribution of the fixed-point representation of x_{mask} is no longer uniform over the ring.

To get around the above issue, we make the following observation: Although it is insecure to reveal x_{mask} in its entirety, it is fine to reveal the absolute fractional value of x_{mask}, denoted by x_{fracMask}, because this distribution is still uniform. Then the actual value of x_{mask} is either $+x_{\mathsf{fracMask}}$ if $x \geq r$, and $-x_{\mathsf{fracMask}}$ otherwise. We also observe that parties can locally compute a sharing of bit $t = \mathbf{1}\{x \geq r\}$ as shown in Line 7 in Algorithm 3.

In the actual protocol, we invoke a fixed-point adapted version of the powers protocol from [LYK+19] on both $+x_{\mathsf{fracMask}}$ and $-x_{\mathsf{fracMask}}$. Then parties can select the correct set of powers using a multiplexer where the selection bit is set to t. We describe our complete protocol in Algorithm 3 where we use $\mathcal{F}_{\mathsf{MUX2}}$ as a black-box.

When $\mathcal{F}_{\mathsf{MUX2}}$ is replaced by an actual 2-round OT protocol, the first round of OT can be parallelized with Line 2 by invoking $\mathcal{F}_{\mathsf{MUX2}}$ on $(p_i^c, p_i^{1\oplus c}, f)$ instead, thus making the selection bit of $\mathcal{F}_{\mathsf{MUX2}}$ independent of the result of reconstruction on Line 2. Hence, the overall protocol will require 2 online rounds. The per-party online communication cost is s bits for Line 2 and $1+2kl$ when realizing $\mathcal{F}_{\mathsf{MUX2}}$ using OT as described earlier. Thus the total communication happens to be $2(s + 1 + 2kl)$ bits.

1. A potential option is to perform all multiplications first (without truncations) and only do truncations at the very end, but this approach would require the ring size to be proportional to the degree d (in order to accommodate the intermediate increase in the scale), and hence will be inefficient.

Algorithm 3: Fixed-point powers Protocol

```
$\Pi_{\mathsf{fxpPowers}}$ :

Input : $[\![x]\!]$, where $x \in [0, 2^s)$ and $\widetilde{x} \in [0, 1)$
Output : $[\![\widehat{y}]\!], [\![\widehat{y^2}]\!], \ldots, [\![\widehat{y^k}]\!]$, where $y = \widetilde{x}$
Precomputation: $[\![\widehat{r}]\!]$, $[\![\widehat{r^2}]\!]$, $\ldots$, $[\![\widehat{r^k}]\!]$, where $r \in \mathbb{R}$ and
  $r \leftarrow [0, 1)$

1  $[\![x - r]\!] \leftarrow [\![x]\!] - [\![r]\!]$
2  $x_{\mathsf{fracMask}} := \mathsf{Recon}([\![x - r]\!]^s)$, where $[\![x - r]\!]^s$ is the $s$
     least significant bits of $[\![x - r]\!]$ and Recon happens in
     the ring $Z_{2^s}$.
3  Let $\langle c \rangle$ be a default sharing of bit $c$ denoting the public
     carry bit in the most significant place during the above
     additive reconstruction.
4  $\{p_i^0\}_{i \in [k]} \leftarrow \Pi_{\mathsf{maskPowers}}(x^0_{\mathsf{fracMask}})$, where
     $x^0_{\mathsf{mask}} := 0^{l-s} || x_{\mathsf{mask}}$
5  $\{p_i^1\}_{i \in [k]} \leftarrow \Pi_{\mathsf{maskPowers}}(x^1_{\mathsf{fracMask}})$, where
     $x^1_{\mathsf{mask}} := 1^{l-s} || x_{\mathsf{mask}}$
   // The value of $x - r$ is $x^0_{\mathsf{fracMask}}$ if $x \geq r$, else $x^1_{\mathsf{fracMask}}$.
6  Let $f$ denote the bit of $[\![x - r]\!]$ at location $s + 1$ from
     LSB.
7  $\langle t \rangle := \langle c \rangle \oplus f$
8  // $d = 0$ if $x \geq r$, and 1 otherwise
9  $\forall i \in [k]$ : $\mathsf{res}_i \leftarrow \mathcal{F}_{\mathsf{MUX2}}(p_i^0, p_i^1, \langle t \rangle)$.
   // Parties use the $t$ bit to select the correct set of powers.
10 return $\mathsf{res}_1, \mathsf{res}_2, \ldots, \mathsf{res}_k$

   .....................................................
   // Local subprocedure invoked by each party $\mathsf{P}_i$
   $\Pi_{\mathsf{maskPowers}}$ :

   Input: $x_{\mathsf{mask}}$ where $x_{\mathsf{mask}} \in [0, 2^s)$
   Output: $[\![\widehat{y}]\!], [\![\widehat{y^2}]\!], \ldots, [\![\widehat{y^k}]\!]$, where $y = \widetilde{x_{\mathsf{mask}}}$

1  $A \leftarrow$
     Initialize empty 2D array of dimension $(k + 1) \times (k + 1)$
2  for $i$ = 0 to $k$ :
3      $A_{0,i} \leftarrow [\![\widehat{r^i}]\!]$
4  for $l$ = 1 to $k$ :
       // Compute all $A_{i,j}$ where $l = i + j$
5      $s \leftarrow 0$
6      for $i$ = 1 to $l$ :
7          $j \leftarrow l - i$
8          $s \mathrel{+}= A_{i-1,j}$
           // Invariant : $s = \sum_{k<i} [\![\widehat{y^{i-1-k} r^{j+k}}]\!]$
9          $A_{i,j} \leftarrow [\![\widehat{r^{i+j}}]\!] + \mathcal{F}_{\mathsf{fxpMult}}(x_{\mathsf{mask}}, s)$
           // Invariant: $A_{i,j}$ will store $[\![\widehat{y^i r^j}]\!]$ following Equation 1
10 return $A_{1,0}, A_{2,0}, \ldots, A_{k,0}$
```

5.3. Secure Polynomial Evaluation

Suppose parties hold a secret-sharing of (fixed-point representation of) a real value x and would like to evaluate a polynomial $Q(x) = \sum_{i=1}^{d} a_i x^i$, where the coefficient $a_i \in \mathcal{R}$ are publicly known. A straightforward way to do so is the following: Parties invoke $\Pi_{\mathsf{fxpPowers}}$ to learn sharing of $\{x, x^2, \ldots, x^k\}$, and then perform a local linear sum of the shares of x^i weighted by coefficient a_i. Thus the overall procedure would require the same online communication cost as $\Pi_{\mathsf{fxpPowers}}$. We observe that one could do better by modifying $\Pi_{\mathsf{fxpPowers}}$ as follows: in Line 9, instead of invoking $\mathcal{F}_{\mathsf{MUX2}}$ for all $i \in [k]$, parties can first locally compute a weighted linear sum $P^0 = \sum_{i=0}^{k} a_i p_i^0$ and $P^1 = \sum_{i=0}^{k} a_i p_i^1$, and then use a *single* invocation of $\mathcal{F}_{\mathsf{MUX2}}$ on inputs $(P^0, P^1, \langle t \rangle)$. This reduces the total communication cost of the protocol to only $2(s + 1 + 2l)$ bits, making it *independent* of the degree d of the polynomial Q. We refer to this optimized protocol as Π_{fxpPoly}.

5.4. Theoretical Cost Analysis of Secure Sigmoid

We will now analyze the online communication cost for our sigmoid protocol. Our overall secure sigmoid construction invokes 1 MIC gate requiring $2l$ bits of total communication, 2 secure exponentiations requiring $4l$ bits of total communication, 2 invocations of Π_{fxpPoly} requiring $4 + 12l$ bits of total communication, 2 secure spline invocations on 10 intervals between $[0, 1)$ requiring $12l$ bits of total communication, and one invocation of $\mathcal{F}_{\mathsf{matMult}}$ in the end to combine the results requiring $24l$ bits of total communication. Thus, the total overall communication for the secure protocol is $54l + 4$ bits of communication.

The entire sigmoid computation can be performed in 4 online rounds. In the first round, we evaluate MIC gate, secure exponentiations and round 1 of secure spline in parallel. In the second round, we complete round 2 of secure spline and also invoke round 1 of Π_{fxpPoly} on the result of secure exponentiation. In the third round, we complete round 2 of Π_{fxpPoly}. In the fourth and final round, we perform a single round of $\mathcal{F}_{\mathsf{matMult}}$ between the MIC gate outputs and the results of Π_{fxpPoly} and secure spline.

The offline cost in the trusted offline setup is measured by the amount of storage needed for the preprocessing.

6. Secure Sigmoid with Distributed Offline Setup

In the previous section, we outlined a secure sigmoid construction which is highly communication efficient in the online phase assuming parties have access to a trusted offline setup phase, possibly using a trusted third party. However, in the real world, such a trusted third party might not be always available or, in some cases, even undesirable. In such scenarios, it becomes essential that the two parties be able to securely emulate the trusted offline phase in an *efficient* manner.

Looking back at our construction in the previous section, we observe that the FSS preprocessing forms the bottleneck cost of securely emulating the trusted offline phase in a 2PC setting. This happens because the FSS key generation algorithm involves the usage of a PRG, and naively running the FSS key generation algorithm inside 2PC will involve the cost of computing the PRG circuit (e.g. AES) using 2PC.

This will typically [2] blow up the communication cost of the offline phase by at least linear in the size of PRG circuit.

In this section, we will discuss an alternative approach for computing sigmoid which will enable a communication efficient offline phase while adding a mild communication overhead in the online phase. We do this by simply replacing the offline-expensive MIC gate (which is based on FSS) with a novel communication efficient secure comparison protocol.

In the previous construction, we have used the FSS based $\mathcal{F}_{\mathsf{MIC}}$ functionality at two different places. We use $\mathcal{F}_{\mathsf{MIC}}$ to determine if the shares of input x lie in the range $[0,1), [1, \frac{1}{\log_2(e)})$, $[\frac{1}{\log_2(e)}, \infty)$, or the negative equivalents of these ranges. Besides this, we also use $\mathcal{F}_{\mathsf{MIC}}$ as a sub-protocol inside the secure spline functionality. Our new construction will be nearly a drop-in replacement for the first $\mathcal{F}_{\mathsf{MIC}}$ functionality based on FSS. As mentioned in Section 5.1, the second spline only needs to operate on the fractional bits of the input and thus can be instantiated on a much smaller domain of s bits. Thus, we retain the second $\mathcal{F}_{\mathsf{MIC}}$ as we can use [Ds17] for efficiently generating FSS keys for it in the distributed setting when $s \leq 24$.

However, one difference is that our construction returns XOR Boolean shares rather than arithmetic shares of a Boolean value. This means that rather than a Beaver-multiplication based dot-product, we instead use $\mathcal{F}_{\mathsf{MUX}}$ on the outputs of comparison in order to select the tailored sigmoid evaluation on the interval corresponding to input x. The parties will use the output of our new comparison as $\mathcal{F}_{\mathsf{MUX}}$ input to either retrieve shares of the sigmoid evaluation on the interval, or shares of 0, and then add together these shares across all intervals to select the sigmoid result.

6.1. Secure Comparison

Suppose party P_0 has a private input x while party P_1 has a private input y. The output of a secure comparsion functionality, henceforth denoted as $\mathcal{F}_{\mathsf{CMP}}$, is a Boolean sharing of $\mathbf{1}\{x < y\}$, a bit indicating the result of comparison, where x and y are bitstrings of length l (interpreted as unsigned bit representation of positive integers). More formally,

$\mathcal{F}^l_{\mathsf{CMP}}(x, y) \rightarrow (b_0, b_1)$
where $x, y \in \{0,1\}^l$
and b_0, b_1 is a Boolean sharing of bit $b := \mathbf{1}\{x < y\}$

A common approach to computing secure comparison is divide-and-conquer [Cou18], [GSV07], [RRK+20b], which first splits the larger input strings into smaller strings, performs comparisons on these smaller strings, and then combines the results. However, these protocols have non-constant number of rounds in the online phase due to a logarithmic depth recursion tree. Cheetah [HLHD22] optimizes the offline communication of [RRK+20b] use VOLE-style OT extension which is orthogonal to our focus of online efficiency.

Another line of work based on function secret sharing (FSS) [BCG+21] performs secure comparison using a distributed comparison function (DCF). This technique allows an online optimal protocol for secure comparison having 1 round and 1 element of communication per party. However, the caveat of directly using FSS to perform comparison is the expensive cost of running the FSS offline phase in 2PC. While Doerner and Shelat [Ds17] propose an elegant approach for performing FSS offline phase, their technique is efficient only for small domains (i.e. input bit lengths less than 20). This is because it requires locally computing an exponential (in input bit length) number of PRGs. There is currently no better concretely communication-efficient technique in the literature for conducting the FSS offline phase.

Another line of work originating from ABY [DSZ15] and it successors like ABY 2.0 [PSSY21a] and SyncCirc [PSSY21b] solve the secure comparison problem by running a GMW-style MPC on a (variant of) boolean adder circuit. These techniques have round complexity proportional to the depth of the circuit which is $O(\log l)$ whereas our protocol is constant round (atmost 3). In Table 6, we compare the cost of our protocol with SyncCirc (the most optimized work in this direction). Rabbit [MRVW21] solves the secure comparison problem in the dishonest majority multiparty setting using boolean adder circuit and other techniques, has higher communication cost and $O(\log l)$ rounds as expected.

In our approach, we start by looking at the decomposition of comparison problem for l length bit-strings in terms of comparison and equality operations on smaller sub-strings as described in Garay et. al. [GSV07]. Formally, for $x = x_1 || x_2$ and $y = y_1 || y_2$, where $x, y \in \{0,1\}^l$ are l length bit strings that we want to compare, the following relationship holds:

$$x < y = (x_1 < y_1) \oplus \Big((x_1 = y_1) \wedge (x_2 < y_2)\Big) \quad (2)$$

In general, we can extend the above decomposition to q pieces in the following way. Let $x = x_1 || \dots || x_q$ and $y = y_1 || \dots || y_q$ where x_i, y_i are m length bit strings, $q = \frac{l}{m}$ (for ease of exposition, assume m divides l). Then, the following relationship holds:

$$\begin{aligned} x < y = &(x_1 < y_1) \\ &\oplus \Big((x_1 = y_1) \wedge (x_2 < y_2)\Big) \\ &\oplus \dots \\ &\oplus \Big((x_1 = y_1) \wedge \dots \wedge (x_{q-1} = y_{q-1}) \wedge (x_q < y_q)\Big) \end{aligned} \quad (3)$$

Looking ahead, the reason for splitting l length bit-string into smaller sub-strings of length m bits is to leverage the power of FSS gates for small input domains (e.g. $m = 16$ bits) which have an efficient offline phase. Assum-

2. This is true for approaches like Garbled Circuit or standard GMW-style secret-sharing based MPC. However, by assuming hardness of problems based on algebraic structures with richer homomorphic properties (e.g. LWE, LPN etc), one can reduce the communication below the circuit size. Currently, these approaches however are computationally much more inefficient than standard MPC approaches to be practical.

ing, $l_i = x_i \overset{?}{<} y_i$ and $e_i = x_i \overset{?}{=} y_i$, we can rewrite the above equivalence as:

$$\begin{aligned} x < y &= l_1 \oplus (e_1 \wedge l_2) \oplus \ldots \oplus (e_1 \wedge \ldots \wedge e_{q-1} \wedge l_q) \\ &= l_1 \oplus (e_1 \wedge l_2) \oplus \ldots \oplus (e_1 \wedge \ldots \wedge e_{q-1} \wedge l_q) \\ &= \langle 1 \quad e_1 \quad e_1 \wedge e_2 \quad \ldots \quad e_1 \wedge e_2 \wedge e_{q-1} \rangle \cdot \\ &\quad \langle l_1 \quad \ldots \quad l_q \rangle \end{aligned} \tag{4}$$

At a high-level, our protocol:

1) Uses q independent FSS comparison gates (based on DCF) for m bit input and 1 bit output to compute $l_1, \ldots, l_q$.
2) Uses $q-1$ independent FSS equality gates (based on DPF) for m bit input and 1 bit output to compute $e_1, \ldots, e_{q-1}$.
3) Given $e_1, \ldots, e_{q-1}$, uses a single iDPF for $q-1$ bit input and 1 bit output in order to compute the all-prefix AND of e_i values i.e. $e_1, e_1 \wedge e_2, e_1 \wedge e_2 \wedge e_3$, etc.
4) Finally, computes a dot product between two bit vectors, each of length q, to get the final result.

Step 1 and Step 2 follow directly from FSS gates for comparison and equality constructed in [BCG+21] and described in Appendix B . Step 4 can be performed in a standard way using bit Beaver triples. We elaborate on Step 3, i.e. how to use the iDPF in order to compute the all-prefix AND of e_i values. We first observe: A single DPF can be easily used to compute the Boolean AND of k bits $b_1, \ldots, b_k$. The observation is that the AND of k bits can be represented as a point function in the following way:

$$\mathsf{AND}(b_1, \ldots, b_k) \quad \equiv \quad f(b) = \begin{cases} 1 & ; \quad b = 2^k - 1 \\ 0 & ; \quad \text{otherwise} \end{cases}$$

where $b = b_1 || \ldots || b_k$

Note that in our context, we want to compute the AND on e_i values. A naive solution is to use $q-1$ independent DPFs to compute all the prefix AND values i.e. $e_1, e_1 \wedge e_2, e_1 \wedge e_2 \wedge e_3$ and so on.

However, since the ANDs are correlated and have an incremental pattern, we can instead use a *single iDPF* to perform the above task much more efficiently. Let's consider the following point function:

$$f(e) = \begin{cases} 1 & ; \quad e = 2^{q-1} - 1 \\ 0 & ; \quad otherwise \end{cases}$$

where $e = e_1 || e_2 || \ldots || e_{q-1}$

If we create an iDPF for this point function, and invoke it on the input $x = e_1 || \ldots || e_k$, we will get 1 as the output iff $e_1 || \ldots || e_k$ is a k length substring of $2^{q-1} - 1$ (which is basically the all 1 string). This will happen iff $\mathsf{AND}(e_1, \ldots, e_k) = 1$. Since an iDPF supports incremental evaluation by design, we can evaluate a single iDPF on $e_1, \ldots, e_k$ for all $k \in [q-1]$ and retrieve the all prefix AND evaluation.

As mentioned in Section 2.4, using an FSS scheme for function f in the context of MPC is done via the corresponding offset function which works on the masked input value $\widetilde{x} := x + r_{in}$ instead of the actual private value x. The input mask r_{in} is defined in the offline phase and is used to define the parameters for FSS key generation. In [BGI19], [BCG+21], the authors use this technique to create an equality check gate and comparison gate via FSS schemes such as DPF and DCF respectively(cf. Appendix B). However, leveraging iDPF in order to create useful MPC gates has been unexplored. In the preceding paragraph, we outlined the way an iDPF can be leveraged for computing the all-prefix AND of multiple bits. However, to use this idea in the context of MPC, we need to operate on masked input and somehow encode the mask inside the iDPF without affecting the correctness. Here we observe that the typical way of masking via group addition i.e. $\widetilde{x} := x + r_{in}$ and then trying to set the special point $\alpha = 2^{q-1} - 1 + r_{in}$ doesn't really work. This is because if $x_1 || \ldots || x_k$ is a length k prefix of $2^{q-1} - 1$, then it doesn't imply that $\widetilde{x_1} || \ldots || \widetilde{x_k}$ is also a length k prefix of α. This means that instantiating a iDPF at α and then performing incremental evaluations on the masked input would not lead to the correct prefix AND result. Our solution to this problem is to use XOR masking instead of the usual group addition based masking. Specifically, we define the masked input $\widetilde{x} := x \oplus r_{in}$ and then instantiate an iDPF with the special point $\alpha = (2^{q-1} - 1) \oplus r_{in}$. It is easy to see that with this masking technique, the following equivalence holds: $x_1 || \ldots || x_k$ is a length k prefix of $2^{q-1} - 1$ iff $\widetilde{x_1} || \ldots || \widetilde{x_k}$ is a length k prefix of α. We describe the protocol $\prod_{\mathsf{CMP}}$ formally in Figure 1

Our protocol requires only 3 (simultaneous) rounds of interaction assuming we instantiate $\mathcal{F}_{\mathsf{innerProduct}}$ using the standard beaver multiplication based approach. The first round requires $2q - 1$ elements of communication per party where each element is of size m bits. The second round requires $q - 1$ bits of communication per party. Finally, the third round requires a secure computation of two secret shared q length bit-vectors. Using standard bit-triples, this can be accomplished using $2q$ bits of communication per party. Noting that the first element in the left vector is a default sharing of 1, the last round can be slightly optimized to $2(q-1)$ bits of communication per party. Overall, the total communication cost of our protocol across both parties is $4l - 2m + 6q - 6$ bits. We note that for the special case when $q = 1$, the second and third round are not needed and the communication cost comes down to just one element (of size $m = l$ bits) per party. This special case is equivalent to a pure DCF based comparison. For the special case where $q = 2$, we can skip the second round of protocol and directly perform the third round using the fact that $t_1 = e_1$ which brings down the communication cost to $2l - m + 2$ bits per party. The computation cost is dominated by the PRG evaluations required for the FSS schemes. Each evaluation of FSS gate on n bit input requires n PRG evaluations per party. Therefore, the total number of local PRG calls in our protocol is $2l - m + q - 1$ per party.

In terms of offline costs, our secure comparison protocol requires generation of $q - 1$ pairs of FSS keys for

Private inputs: P_0 has l bit private input x and P_1 has l bit private input y

Output: P_i outputs a share $[z]_i$ such that $z = 1\{x < y\}$

Preprocessing:

- For all $i \in [q-1]$: $\left(k^{\text{eq}}_{0,i}, k^{\text{eq}}_{1,i}\right) \leftarrow \mathsf{Gen}^{\text{eq}}\left(1^\lambda, r^{\text{eq,in}}_i, s^{\text{eq,in}}_i, r^{\text{eq,out}}_i\right)$. P_0 gets $k^{\text{eq}}_{0,i}, r^{\text{eq,in}}_i$ whereas P_1 gets $k^{\text{eq}}_{1,i}, s^{\text{eq,in}}_i$
- For all $i \in [q]$: $\left(k^{\text{cmp}}_{0,i}, k^{\text{cmp}}_{1,i}\right) \leftarrow \mathsf{Gen}^{\text{cmp}}\left(1^\lambda, r^{\text{cmp, in}}_i, s^{\text{cmp, in}}_i, r^{<\text{,out}}_i\right)$. P_0 gets $k^{\text{cmp}}_{0,i}, r^{\text{cmp, in}}_i, \left[r^{\text{cmp, out}}_i\right]_0$ whereas P_1 gets $k^{\text{cmp}}_{1,i}, s^{\text{cmp, in}}_i, \left[r^{\text{cmp, out}}_i\right]_1$
- $\left(k^{\text{iDPF}}_0, k^{\text{iDPF}}_1\right) \leftarrow \mathsf{Gen}^{\text{iDPF}}_q\left(1^\lambda, \alpha \oplus r^{\text{eq,out}}, \{\beta_i\}_{i\in[q]}\right)$ where $\alpha = 2^{q-1} - 1$, $\beta_i = 1$, $r^{\text{eq,out}} = r^{\text{eq,out}}_1 || ... || r^{\text{eq,out}}_q$. P_0 gets k^{iDPF}_0 whereas P_1 gets k^{iDPF}_1

P_0 $\qquad\qquad$ P_1

P_0:

Parse x as $x_1||...||x_q$

For all $i \in [q-1]$, compute $x^{\text{eq}}_i := x_i + r^{\text{eq, in}}_i$

For all $i \in [q]$, compute $x^{\text{cmp}}_i := x_i + r^{\text{cmp, in}}_i$

P_1:

Parse y as $y_1||...||y_q$

For all $i \in [q-1]$, compute $y^{\text{eq}}_i := y_i + s^{\text{eq, in}}_i$

For all $i \in [q]$, compute $y^{\text{cmp}}_i := y_i + s^{\text{cmp, in}}_i$

$$\xrightarrow{\left\{x^{\text{eq}}_i\right\}_{i\in[q-1]}\left\{x^{\text{lt}}_i\right\}_{i\in[q]}} \qquad \xleftarrow{\left\{y^{\text{eq}}_i\right\}_{i\in[q-1]}\left\{y^{\text{lt}}_i\right\}_{i\in[q]}}$$

P_0:

For all $i \in [q-1]$, compute $[\widetilde{e_i}]_0 := \mathsf{Eval}^{\text{eq}}\left(k^{\text{eq}}_{0,i}, x^{\text{eq}}_i, y^{\text{eq}}_i\right)$

For all $i \in [q]$, compute:

- $\left[\widetilde{l_i}\right]_0 := \mathsf{Eval}^{\text{cmp}}\left(k^{\text{cmp}}_{0,i}, x^{\text{cmp}}_i, y^{\text{cmp}}_i\right)$
- $[l_i]_0 := \left[\widetilde{l_i}\right]_0 \oplus \left[r^{\text{cmp,out}}_i\right]_0$

P_1:

For all $i \in [q-1]$, compute $[\widetilde{e_i}]_1 := \mathsf{Eval}^{\text{eq}}\left(k^{\text{eq}}_{1,i}, x^{\text{eq}}_i, y^{\text{eq}}_i\right)$

For all $i \in [q]$, compute:

- $\left[\widetilde{l_i}\right]_1 = \mathsf{Eval}^{\text{cmp}}\left(k^{\text{cmp}}_{1,i}, x^{\text{cmp}}_i, y^{\text{cmp}}_i\right)$
- $[l_i]_1 = \left[\widetilde{l_i}\right]_1 \oplus \left[r^{\text{cmp, out}}_i\right]_1$

$$\xrightarrow{\{[\widetilde{e_i}]_0\}_{i\in[q-1]}} \qquad \xleftarrow{\{[\widetilde{e_i}]_1\}_{i\in[q-1]}}$$

P_0:

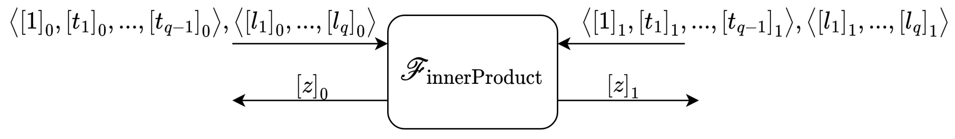
For all $i \in [q-1]$: $\widetilde{e_i} = \mathsf{Reconstruct}([\widetilde{e_i}]_0, [\widetilde{e_i}]_1)$

For all $i \in [q-1]$: $[t_i]_0 = \mathsf{Eval}^{\text{iDPF}}\left(k^{\text{iDPF}}_0, \widetilde{e_1}||...||\widetilde{e_i}\right)$

P_1:

For all $i \in [q-1]$: $\widetilde{e_i} = \mathsf{Reconstruct}([\widetilde{e_i}]_0, [\widetilde{e_i}]_1)$

For all $i \in [q-1]$: $[t_i]_1 = \mathsf{Eval}^{\text{iDPF}}\left(k^{\text{iDPF}}_1, \widetilde{e_1}||...||\widetilde{e_i}\right)$



$\left\langle [1]_0, [t_1]_0, ..., [t_{q-1}]_0 \right\rangle, \left\langle [l_1]_0, ..., [l_q]_0 \right\rangle$
$[z]_0$
$\mathscr{F}_{\text{innerProduct}}$
$\left\langle [1]_1, [t_1]_1, ..., [t_{q-1}]_1 \right\rangle, \left\langle [l_1]_1, ..., [l_q]_1 \right\rangle$
$[z]_1$

Output $[z]_0$ $\qquad\qquad$ Output $[z]_1$

Figure 1: Constant round secure comparison protocol $\prod_{\mathsf{CMP}}$ for l bit inputs.

equality check of m length bit-strings, q pairs of FSS keys for comparison check of m length bit-strings, a single iDPF key pair for $q-1$ length bitstrings and $q-1$ bit Beaver triples for peroming inner product. Leveraging the technique of [Ds17], generating a single DPF key pair will require $2 + m(12\lambda + 8)$ bits of communication, a single DCF key pair will require $m(12\lambda + 10)$ bits of communication and a single iDPF key pair will require $(q-1)(12\lambda + 10)$ bits of communication. Additionally, as mentioned in [RRK+20b], generating a single bit Beaver triple requires an amortized cost of $\lambda + 16$ bits of communication using OT extension techniques. Adding up all the costs, the offline communication of our protocol comes out to be $24\lambda l - 12\lambda m + 13\lambda q - 13\lambda + 18l - 8m + 28q - 28$ bits. The FSS key generation (for DPF, DCF, iDPF) on n bit input domains using [Ds17] requires a local computation of 2^{n+1} PRG calls per party. This leads to a total offline computation cost of $q \cdot 2^{m+2} - 2^{m+1} + 2^q$ AES calls per party. Using an estimate of 360 million AES calls per second on a single core 3.6 GHz machine (10 machine cycles per AES) [TSS+20], the offline computation cost will take around 5 milliseconds for $q = 8, m = 16$.

Before proceeding further, we note that an alternative formulation of secure comparison lets the two parties hold secret shares of x and y as input (instead of private inputs) and learn a secret shared bit b representing the comparison output. We will use $\mathcal{G}_{\mathsf{CMP}}$ to denote this alternative functionality which is more relevant in the context of secret-sharing MPC, for example in our secure logistic regression use-case. As mentioned in [Cou18], this problem can be non-interactively and black-box reduced to $\mathcal{F}_{\mathsf{CMP}}$. The observation is based on the following relationship which reduces the task of comparing two shared values x, y to the task of securely computing the MSB of $x - y$:

$$x < y = MSB(x - y) \tag{5}$$

Now, in order to compute the MSB of some l bit secret value z, where P_0 and P_1 hold share z_0 and z_1 respectively, we can use the following observation:

$$MSB(z) = MSB(z_0) \oplus MSB(z_1) \oplus \mathbf{1}\{2^{l-1} - 1 - z_0 < z_1\} \tag{6}$$

Using $2n$ invocations of $\mathcal{G}_{\mathsf{CMP}}$ and a n invocations of $\mathcal{F}_{\mathsf{AND}}$, one can easily realize the functionality captured by MIC gate for n public intervals. In our sigmoid use-case for distributed offline setup, we will replace the MIC gate with multiple invocation of $\mathcal{G}_{\mathsf{CMP}}$ and $\mathcal{F}_{\mathsf{AND}}$. Instantiating $\mathcal{G}_{\mathsf{CMP}}$ with the 3 round protocol described in Section 6.1 and $\mathcal{F}_{\mathsf{AND}}$ with the usual bit beaver multiplication, the protocol for MIC evaluation will require 4 online rounds and $8nl - 4mn + 12nq - 8n$ bits of online communication (where m and q are parameters in our secure comparison protocol). As an example, for $n = 6, l = 63, m = 16, q = 4$ (which is the value used in our experiments), this leads to 2880 bits of communication.

6.2. Secure FSS Key Generation

As we have seen, our secure comparison protocol invokes FSS primitives such as DPF, DCF and iDPF. Besides this, our secure spline protocol invoke $\mathcal{F}_{\mathsf{MIC}}$ which in-turn relies on a DCF. In order to implement the offline phase of our protocol, we also need to generate keys for these FSS primitives efficiently in a 2PC setting. We discuss our novel approach to this in Appendix E, together with the reasons why the approaches of [BCG+21] and [Ds17] are insufficient.

6.3. Theoretical cost analysis of Secure Sigmoid with Distributed Offline Setup

We will now analyze the online communication cost for our sigmoid protocol for this distributed offline setting. As mentioned before, the difference in this sigmoid compared to the previous section is that the task of MIC gate is performed using secure comparison protocol. In our experiments, we set the parameter $m = 16$ in the secure comparison protocol. For 6 intervals, emulating the MIC gate using secure comparison and AND gates (as described in Section 6.1) requires a total communication of $48l + 72q - 432$ bits (where q is a parameter in secure comparison protocol, $q = \lceil l/m \rceil$). Besides this, like the previous sigmoid construction, we invoke 2 secure exponentiations requiring $4l$ bits of total communication, 2 invocations of Π_{fxpPoly} requiring $4 + 12l$ bits of total communication, 2 secure spline invocations on 10 intervals between $[0, 1)$ requiring $12l$ bits of total communication. One additional change is the following: Instead of using one invocation of $\mathcal{F}_{\mathsf{matMult}}$ in the end to combine the results, we now use 6 invocations of $\mathcal{F}_{\mathsf{MUX}}$ requiring $24l + 12$ bits of total communication. This modification is needed because the outputs of secure comparison are boolean shares and hence not directly compatible for a multiplication with arithmetic shared values [3]. Thus, the total overall communication for the secure protocol is $100l + 72\lceil l/m \rceil - 416$ bits of communication. The entire protocol without the MIC gate emulation (using secure comparison) can be performed in 3 rounds as described in Section 5.4. Our emulation of MIC gate using secure comparison protocol requires 4 online rounds. These rounds can be performed in parallel with the secure exponentiation, Π_{fxpPoly} and secure spline protocols. In the end, the $\mathcal{F}_{\mathsf{MUX}}$ requires an additional 2 rounds of communication. Thus, the total number of rounds required by our protocol is 6.

7. Experimental Evaluation

Implementation Details. We implemented our constructions in C++ and compiled our system with the Bazel build system [Baz22]. We used native C++ `uint64_t` for most operations. For FSS operations, we used `uint_128` from the Abseil library [Goo22a].

3. An alternative solution is to perform share conversion from boolean to arithmetic and then use a beaver multiplication based dot-product to combine the results. This would lead to approximately the same cost

Experimental Setup. We ran our experiments on two compute-optimized c2-standard-8 Google Cloud instances with 32 GB RAM and Intel Xeon CPU at 3.1GHz clock rate. Our implementation runs on a single thread and utilizes a single core of each instance. In the LAN setting, both instances were deployed in the us-central1 region where the mean network latency was 0.15ms and the bandwidth was $\approx 2.1 GB/s$. In the WAN setting, one instance was in us-central1 while the other was in us-west2. The mean network latency was 46.20ms and the bandwidth $\approx 60 MB/s$. All runtimes and communication are end-to-end totals and include both the client and server costs, with all computation sequentialized (i.e., server and client do not compute at the same time).

Cloud costs. We include the monetary cost of running our protocols on the Google Cloud Platform (GCP), using the prices listed on the GCP website. For computational cost, we use the CPU spot price of \$0.02 per-hour for pre-emptible virtual machines, and use network cost of \$0.08 per GB for egress to the internet. All prices are in USD. This reflects batch computation with parties situated in different cloud providers, as has been used in other works [IKN+20].

7.1. Sigmoid Experiments

Approximating the sigmoid function is the most challenging and costly component of gradient descent (see our discussion in Appendix H) . For that reason, we benchmark our sigmoid protocols separately. In this section, we refer to our sigmoid protocol with trusted offline setup (Section 5) as v1 and our sigmoid protocol with distributed offline setup (Section 6) as v2. We show runtime, communication and monetary cost in Table 1, for $10^2, 10^3$, and 10^4 sigmoid inputs and the following parameters: 20 fractional bits, 31-bit width (integer and fractional), 63-bit ring size, 10 spline intervals in $[0, 1)$, and Taylor series of degree 10^4. The sigmoids are executed in a single batch.

Sigmoid Accuracy Figure 2 (also Figure 6 in the appendix) gives visual representations of our sigmoid accuracy as compared to the Python floating point implementation of sigmoid ("true" sigmoid). We note that in Figure 2, our sigmoid nearly exactly overlaps with the true sigmoid. In Figure 6 in appendix, we show that our error remains tiny, on the order of 10^{-4} when using 20 fractional bits.

Benchmark Comparisons. We compare to the most recent secure sigmoid protocols in Table 1. We focus on *accurate* sigmoid approximations as inaccurate approximations often result in worse models than in standard logistic regression (see Section 7.2). Our comparison includes SIRNN [RRG+21], whose sigmoid protocol strictly improves over other state-of-the-art sigmoid approximations such as MiniONN [LJLA17] and DeepSecure [RRK18]. We also compare to the sigmoid approximation presented in MP-SPDZ[5] [Kel20].

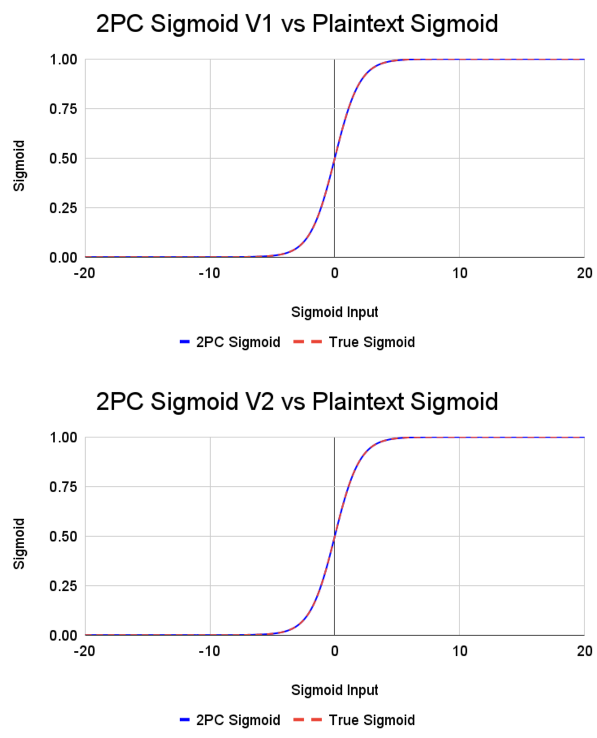


Figure 2: Our sigmoid with trusted offline setup (top) and distributed offline setup (bottom) executed in 2PC vs standard Python sigmoid implementation.

Table 1: Comparison of our sigmoids with trusted ($v1$) and distributed ($v2$) offline setup to SIRNN and MP-SPDZ's accurate sigmoid implementation.

Technique	Time for # Instances (sec) 10^2	10^3	10^4	Comm. per Instance (KB)	# Rounds	USD Cost per 10^6 runs
				LAN		
Sigmoid v1	1.83	18.31	183.08	**0.50**	**4**	**\$0.140**
Sigmoid v2	1.69	17.33	173.84	**1.18**	**6**	**\$0.186**
MP-SPDZ	0.05	0.29	-	15.32	124	\$1.170
EST. SIRNN	0.02	0.04	0.19	4.88	≈100	\$0.372
				WAN		
Sigmoid v1	2.10	19.07	185.20	**0.50**	**4**	**\$0.141**
Sigmoid v2	2.16	18.79	176.80	**1.18**	**6**	**\$0.188**
MP-SPDZ	7.12	9.02	-	15.32	124	\$1.219
EST. SIRNN	4.60	4.62	4.77	4.88	≈100	\$0.375

We retrieved the SIRNN values directly from [RRG+21] and extrapolated the cost for our network settings[6]. For MP-SPDZ, we ran their sigmoid implementation in the trusted dealer mode in our environment.

We observe a gain of $\approx 10\times$ in communication efficiency over SIRNN and $\approx 30\times$ over MP-SPDZ. We also reduce the rounds for sigmoid ($\approx 25\times$ over SIRNN and $\approx 31\times$ over MP-SPDZ): Our v1 sigmoid requires 4 rounds

4. We were not able to compile 10^4 sigmoid executions into the bytecode used by the MP-SPDZ virtual machine as our device ran out of memory.

5. MP-SPDZ presents different sigmoid approximations. We focus our comparison on their accurate sigmoid approximation, which directly computes exponentiation and reciprocal in MPC.

6. Note that SIRNN does not have offline phase. The entire protocol cost is online.

while our v2 sigmoid requires 6 rounds, while we estimate that SiRnn uses ≈ 100 communication rounds (though the number of rounds is not discussed in SiRnn) while MP-SPDZ uses 124 rounds. While our construction has higher computation than SiRnn and MP-SPDZ, we have better monetary costs. We decrease costs by $\approx 2.7\times$ over SiRnn and by $\approx 8.4\times$ over MP-SPDZ on LAN. On a higher latency WAN, we decrease costs by $\approx 2.7\times$ over SiRnn and by $\approx 8.6\times$ over MP-SPDZ.

The improved sigmoid costs also impact inference, where the cost is the sum of a single matrix-vector multiplication and a single sigmoid evaluation. For example, inference on a single example of 10 features requires one-time communication of $1.25KB$ (recall this cost can be paid once as the model does not change) and $1.752KB$ per each example. For 10^6 examples this amounts to $1.67GB$ and costs \$0.13 assuming communication is the only cloud cost. For comparison, using SiRnn's sigmoid would cost \$0.47.

Offline cost estimate. We now provide an analytical estimate of the offline costs involved in our sigmoid protocol. For the v1 setting, we measure offline cost as the storage cost of preprocessing material per party which consists of the following components:

- MIC gate: Requires a key size of $(l \cdot \lambda + l^2 + 3l + \lambda) + 2 \cdot n_I \cdot l$ bits where n_I is the number of MIC intervals [BCG+21]. For $l = 63$ and $n_I = 6$, we get a key size of 1.6 KB.
- Exponentiation: Requires a correlation consisting of 2 field elements of atmost l bits [KLRS22]. For $l = 63$, we get a cost of 0.015 KB.
- Polynomial: Requires a correlation consiting of (shares of) d incremental powers of a random ring element. For $l = 63$, we get a cost of 0.077 KB.
- Spline: Requires a MIC gate key for 10 intervals and 1 Beaver triple. For $l = 63$, we get a cost of 0.56 KB.
- Dot product: For performing dot product of two vectors of length 6, we require 6 Beaver triples. For $l = 63$, we get a cost of 0.046 KB.

In the v1 sigmoid protocol, we use 1 MIC gate, 2 exponentiations, 2 polynomial calls, 2 spline calls and 1 Dot Product. This requires a total storage cost of 2.95 KB per party.

For the v2 setting, the offline cost is measured by the total communication cost needed to generate the required preprocessing material in MPC. In the following, we assume that the amortized cost of generating a Beaver triple for $l = 63$ bit ring is 0.4375 KB [RSS19] and a bit Beaver triple is 0.0175 KB [RRK+20b]. The v2 sigmoid protocol consists of the following components:

- Mux : Requires 2 calls to Ideal OT in the online phase. For this, it suffices to have 2 ROT correlations generated in the offline phase which will require $2\lambda + 4l$ bits of communication. For $l = 63$, we get a communication cost of 0.062 KB.
- Exponentiation: The required correlation can be generated using 2 Beaver multiplications which will cost 0.936 KB.
- Polynomial: The required correlation for $d = 10$ degree polyonimal can be generated using $d = 10$ Beaver multiplications which will cost 4.68 KB.
- Spline: Requires an MIC gate key on domain size $s = 20$ bits for 10 intervals with output size of $l = 63$ bits, and 1 Beaver triple. Using the protocols described in Appendices B E for MIC key generation, the spline offline cost will be 5.81 KB.
- MIC based on $\prod_{\mathsf{CMP}}$: For 6 intervals, we need 12 invocations of $\prod_{\mathsf{CMP}}$ and 6 invocations of $\mathcal{F}_{\mathsf{AND}}$ (perfomed using bit Beaver triples). For $l = 63$, this will cost 260.98 KB.

In the v2 sigmoid protocol, we use invoke 1 MIC gate, 2 exponentiations, 2 polynomial calls, 2 spline calls and 6 Mux calls. This requires a total offline communication cost (across both parties) of 284 KB per party.

7.2. Logistic Regression Experiments

We evaluated our logistic regression experiments on four datasets. We do basic preprocessing on the datasets with the help of Scikit-learn's machine learning library (remove rows with missing features, normalize features with Scikit-learn's `StandardScaler`, shuffle the rows, etc.). To facilitate testing, we split each dataset into a training set (70%) and a testing set (30%). We use the following datasets and summarize their sizes and training parameters in Table 2.

Table 2: Datasets we used for our experiments.

	Titanic	Arcene	Criteo Uplift	Gisette
Training Size	500	70	7000	4200
Testing Size	214	30	3000	1800
Total Size	714	100	10000	6000
# Features	6	10000	15	5000
Learning Rate α	1	0.1	0.1	1
Regularization λ	0.0001	0.0001	0.1	0.1
Prediction Threshold	0.43	0.18	0.59	0.64
# Iterations	6	6	6	6

- *Titanic.* This dataset is used to predict which passengers survived the Titanic shipwreck. Survival is modeled as function of passenger class, ticket cost, and demographics such as sex, age range, etc. Our preprocessed dataset contains data on 714 passengers and 6 features.
- *Arcene.* This dataset contains information from the National Cancer Institute and the Eastern Virginia Medical School. It is used to determine whether a patient has cancer. There are 100 patients and 10^4 features.
- *Criteo Uplift.* This dataset predicts whether a user targeted by advertising purchases a product (i.e. converts). We used a subset of the original dataset with 10^4 data points and 15 features. The features constitute data about the users and the labels signify whether they converted. This dataset is highly imbalanced. From 10000 data points, there were only 47 conversions.
- *Gisette.* This dataset is used to distinguish the digits 4 and 9. The digits were originally encoded in fixed-size 28×28 images and preprocessed to yield 5000 features.

Our preprocessed Gisette dataset contains 6000 digit samples.

Accuracy Evaluation. We compare accuracy of our 2PC protocols against a plaintext floating-point implementation in Table 4. We use 6 iterations of logistic regression, using parameters described above. Our 2PC protocols are close to plaintext logistic regression in all cases except Criteo Uplift, where we observe a moderate loss in accuracy and also F1 score. This dataset is especially challenging for logistic regression since label distribution is heavily imbalanced.

Performance Evaluation. We present our end-to-end runtime and total communication costs in Table 5. All versions use the parameters described above, and run for 6 iterations. Our runtimes and communication are totals for both parties. We observe that our costs grow nearly linearly with the number of examples, but are relatively independent of the number of features. This emphasizes that sigmoid is a significant portion of our protocol costs.

Comparison to previous works. We first note that some key works in the area have relatively coarse sigmoid approximations, and we do not do a detailed comparison with these works. For example SecureML [MZ17](see Figure 5). As a result, SecureML does not closely match plaintext logistic regression: running logistic regression in plaintext with SecureML's sigmoid approximation on the Titanic dataset yields 0.59 accuracy while plaintext (and our approach) yields ≈ 0.79. Similarly, F1 score goes down from our ≈ 0.78 to SecureML's ≈ 0.65. ABY2.0 [PSSY21a] improves on the efficiency of SecureML's logistic regression by reducing the online runtime, but uses the same coarse sigmoid approximation.

If we settled for the sigmoid approximation used in SecureML and ABY2.0, our work would offer an improvement in online communication. This is because their sigmoid approximation essentially reduces to secure comparisons and AND gates. Our improved comparison protocol (see in Table 6) would reduce training costs over ABY2.0 (which is a predecessor to SynCirc [PSSY21b]).

Turning to accurate logistic regression approximations, MP-SPDZ [Kel20] implements logistic regression and has comparable accuracy to our protocol. We ran the online phase of MP-SPDZ logistic regression on a dataset of 1000 examples and 10 features and compared to our logistic regression implementation with v1 sigmoid. One iteration of MP-SPDZ communicates ≈ 18.8 MB while our work requires ≈ 0.5 MB, a $38\times$ improvement. On an ultra low-latency LAN, our runtime is slower (≈ 0.3s for MP-SPDZ and 18.6s for our technique) but we perform better on a higher latency WAN (MP-SPDZ takes ≈ 25.3s while our technique takes 19.6s) due to MP-SPDZ's multi-round nature.

SIRNN [RRG+21] implements sigmoid, but does not implement logistic regression, so we limit ourselves to the sigmoid comparison in the preceding section.

7.3. Secure Comparison Experiments

We compare our secure comparison protocol $\prod_{\mathsf{CMP}}$ against [BCG+21]'s FSS comparison gate. In Appendix C, we provide an analytical cost comparison with latest works.

Performance Comparison [BCG+21]'s FSS comparison gate presents the lowest known online communication for comparison, but the offline phase is computationally infeasible for 64 bit inputs. We show that with a relatively small increase to our costs ($2\times$ online communication and a runtime increase from 0.364ms to 0.532ms on a LAN network and a batch of 1000 comparisons), we can make the offline phase computationally feasible (see discussion in Section 6.1). We present our experiments in Table 3.

Table 3: Comparison of our new $\prod_{\mathsf{CMP}}$ protocol to the FSS protocol [BCG+21] on a batch of 1000 inputs.

	$\prod_{\mathsf{CMP}}$	FSS Comparison [BCG+21]
LAN (sec)	0.532	0.364
WAN (sec)	0.671	0.410
Communication (KB)	29.55	15.63

Table 4: Accuracy comparison of our 2PC algorithms with plaintext algorithm implemented in Python floating point.

	Python Plaintext	2PC Approach V1	2PC Approach V2
		Titanic Dataset	
F1 Score	0.77551	0.77551	0.77551
Accuracy	0.79439	0.79439	0.79439
		Arcene Dataset	
F1 Score	0.76923	0.76923	0.76923
Accuracy	0.8	0.8	0.8
		Criteo Uplift Dataset	
F1 Score	0.47059	0.38462	0.38462
Accuracy	0.994	0.98933	0.98933
		Gisette Dataset	
F1 Score	0.96987	0.96540	0.96540
Accuracy	0.97056	0.96611	0.96611

Table 5: Total costs of running our 2PC gradient descent for 6 iterations on 4 datasets with 20 fractional bits of precision.

	2PC V1	2PC V2	2PC with DP V1	2PC with DP V2
		Titanic Dataset (500×6)		
Comm (MB)	1.49	3.38	1.49	3.38
LAN (sec)	55.34	54.29	55.07	53.46
WAN (sec)	58.7317	58.8215	58.4406	58.5171
LAN Cost (USD)	0.04c	0.06c	0.04c	0.06c
WAN Cost (USD)	0.04c	0.06c	0.04c	0.06c
		Arcene Dataset (70×10000)		
Comm (MB)	1.24	1.50	1.18	1.45
LAN (sec)	9.47	9.06	9.39	9.00
WAN (sec)	12.99	13.02	12.90	13.04
LAN Cost (USD)	0.015c	0.017c	0.014c	0.016c
WAN Cost (USD)	0.017c	0.019c	0.016c	0.019c
		Criteo Uplift Dataset (7000×15)		
Comm (MB)	20.81	47.27	20.81	47.27
LAN (min)	12.94	12.82	12.88	12.77
WAN (min)	13.15	13.04	13.08	12.96
LAN Cost (USD)	0.59c	0.80c	0.59c	0.79c
WAN Cost (USD)	0.60c	0.80c	0.60c	0.80c
		Gisette Dataset (4200×5000)		
Comm (MB)	13.00	28.88	12.97	28.85
LAN (min)	8.67	8.56	8.59	8.49
WAN (min)	9.06	8.96	8.88	8.79
LAN Cost (USD)	0.39c	0.51c	0.39c	0.51c
WAN Cost (USD)	0.40c	0.52c	0.40c	0.52c

8. Conclusion

We show that techniques from FSS can be combined with secret-sharing MPC to get the best of both worlds in terms of online communication cost. Specifically, we can have reduced communication in the online phase while still having an efficient offline phase. We demonstrate this idea by designing a novel secure logistic regression training protocol with the best known online communication ($\approx 38\times$ lower than MP-SPDZ), a secure sigmoid evaluation construction with $\approx 10\times$ online communication reduction over state-of-the-art SiRnn and a secure comparison protocol with *constant* number of communication rounds that reduces online communication over SynCirc by $\approx 1.3 - 2.6\times$.

References

[ACG$^+$16] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[AG21] Apple and Google. Exposure notifications private analytics. https://github.com/google/exposure-notifications-android/blob/master/doc/ENPA.pdf, 2021.

[AKAA07] S. Akinci, E. Kaynak, E. Atilgan, and Ş. Aksoy. Where does the logistic regression analysis stand in marketing literature? a comparison of the market positioning of prominent marketing journals. *European Journal of Marketing*, 2007.

[AS19] Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 509–529. Springer, Heidelberg, June 2019.

[ASSKG19] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J Kusner, and Adrià Gascón. Quotient: two-party secure neural network training and prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1231–1247, 2019.

[Baz22] Bazel. Bazel. https://bazel.build/, 2022.

[BBCG$^+$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.

[BCD$^+$09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security, 13th International Conference*, 2009.

[BCG$^+$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 Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 871–900. Springer, Heidelberg, October 2021.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.

[BEG$^+$19] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, Brendan McMahan, et al. Towards federated learning at scale: System design. *Proceedings of Machine Learning and Systems*, 1:374–388, 2019.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[BGI19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 341–371. Springer, Heidelberg, December 2019.

[BKK$^+$16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. *Proc. Priv. Enhancing Technol.*, 2016.

[BLL21] Junyoung Byun, Woojin Lee, and Jaewook Lee. Parameter-free he-friendly logistic regression. *Advances in Neural Information Processing Systems*, 34:8457–8468, 2021.

[BTC87] Carl Boyd, Mary Ann Tolson, and Wayne S. Copes. Evaluating trauma care. *The Journal of Trauma: Injury, Infection, and Critical Care*, 1987.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation, (NSDI)*, 2017. https://crypto.stanford.edu/prio/ (accessed 2020-12-09).

[CG20] Henry Corrigan-Gibbs. Privacy-preserving firefox telemetry with prio. https://rwc.iacr.org/2020/slides/Gibbs.pdf, 2020.

[CKP22] Jung Hee Cheon, Wootae Kim, and Jai Hyun Park. Efficient homomorphic evaluation on large interval. *Cryptology ePrint Archive*, 2022.

[Cou18] Geoffroy Couteau. New protocols for secure equality test and comparison. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 303–320. Springer, Heidelberg, July 2018.

[CsU19] Jeffrey Champion, abhi shelat, and Jonathan Ullman. Securely sampling biased coins with applications to differential privacy. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[CYS21] Rishav Chourasia, Jiayuan Ye, and Reza Shokri. Differential privacy dynamics of langevin diffusion and noisy gradient descent. In *Advances in Neural Information Processing Systems*, 2021.

[DFK$^+$06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Heidelberg, March 2006.

[DKM$^+$06] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, 2006.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, 2006.

[DN04] Cynthia Dwork and Kobbi Nissim. Privacy-preserving datamining on vertically partitioned databases. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 528–544. Springer, Heidelberg, August 2004.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[EKR18] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Found. Trends Priv. Secur.*, 2018.

[Fre09] David A. Freedman. *Statistical Models: Theory and Practice*. Cambridge University Press, 2 edition, 2009.

[Gil21] Alexandre Gilotte. Results from the criteo-adkdd-2021 challenge. https://medium.com/criteo-engineering/results-from-the-criteo-adkdd-2021-challenge-50abc9fa3a6, 2021.

[GKCG22] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. Llama: A low latency math library for secure inference. *Proceedings on Privacy Enhancing Technologies*, 4:274–294, 2022.

[Gol64] Robert E. Goldschmidt. Applications of division by convergence. *Master's thesis, MIT*, 1964.

[Gol09] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2009.

[Goo22a] Google. Abseil - c++ common libraries. https://github.com/abseil/abseil-cpp, 2022.

[Goo22b] Google. https://developer.chrome.com/docs/privacy-sandbox/attribution-reporting/. https://developer.chrome.com/docs/privacy-sandbox/attribution-reporting/, 2022.

[Goo22c] Google. Privacy sandbox. https://privacysandbox.com/intl/en_us/, 2022.

[GSB+17] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. *PoPETs*, 2017(4):345–364, October 2017.

[GSV07] Juan Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In *International Workshop on Public Key Cryptography*, pages 330–342. Springer, 2007.

[HLHD22] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure two-party deep neural network inference. *IACR Cryptol. ePrint Arch.*, 2022:207, 2022.

[HRK+20] Charlie Harrison, Mariana Raykova, Michael Kleeber, John Delaney, and Andres Munoz Medina. Multi-browser aggregation service explainer. https://github.com/WICG/conversion-measurement-api/blob/main/SERVICE.md, 2020.

[IKN+20] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On deploying secure computing: Private intersection-sum-with-cardinality. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 370–389. IEEE, 2020.

[JM09] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (3rd Edition)*. 2009.

[JWEG18] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. Distributed learning without distress: Privacy-preserving empirical risk minimization. In *Advances in Neural Information Processing Systems*, 2018.

[Kel20] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.

[KLRS22] Mahimna Kelkar, Phi Hung Le, Mariana Raykova, and Karn Seth. Secure poisson regression. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[KVH+21] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems*, 34, 2021.

[LJLA17] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 619–631, 2017.

[LYK+19] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. HoneyBadgerMPC and AsynchroMix: Practical asynchronous MPC and its application to anonymous communication. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 887–903. ACM Press, November 2019.

[MRVW21] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In Nikita Borisov and Claudia Diaz, editors, *Financial Cryptography and Data Security*, pages 249–270, Berlin, Heidelberg, 2021. Springer Berlin Heidelberg.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

[PD09] Sanjay Kumar Palei and Samir Kumar Das. Logistic regression model for prediction of roof fall risks in bord and pillar workings in coal mines: An approach. *Safety Science*, 2009.

[PSSY21a] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2165–2182. USENIX Association, August 2021.

[PSSY21b] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Syncirc: Efficient synthesis of depth-optimized circuits for secure computation. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 147–157. IEEE, 2021.

[RMY18] Deevashwer Rathee, Pradeep Kumar Mishra, and Masaya Yasuda. Faster PCA and linear regression through hypercubes in HElib. Cryptology ePrint Archive, Report 2018/801, 2018. https://eprint.iacr.org/2018/801.

[RRG+21] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. SiRnn: A math library for secure RNN inference. In *2021 IEEE Symposium on Security and Privacy*, pages 1003–1020. IEEE Computer Society Press, May 2021.

[RRK18] Bita Darvish Rouhani, M Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th annual design automation conference*, pages 1–6, 2018.

[RRK+20a] 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*, pages 325–342, 2020.

[RRK+20b] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow2: Practical 2-party secure inference. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 325–342. ACM Press, November 2020.

[RSS19] Deevashwer Rathee, Thomas Schneider, and KK Shukla. Improved multiplication triple generation over rings via rlwe-based ahe. In *International Conference on Cryptology and Network Security*, pages 347–359. Springer, 2019.

[RTPB20] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. Ariann: Low-interaction privacy-preserving deep learning via function secret sharing. *Proceedings on Privacy Enhancing Technologies*, 2022(1):291–316, 2020.

[SGRP19] Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas. Make some ROOM for the zeros: Data sparsity in secure distributed machine learning. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1335–1350. ACM Press, November 2019.

[TSS+20] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *arXiv preprint arXiv:2004.13293*, 2020.

[WTB+20] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *arXiv preprint arXiv:2004.02229*, 2020.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[YLC+19] Qiang Yang, Yang Liu, Yong Cheng, Yan Kang, Tianjian Chen, and Han Yu. Federated learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(3):1–207, 2019.

[YS22] Jiayuan Ye and Reza Shokri. Differentially private learning needs hidden state (or much faster convergence). *CoRR*, abs/2203.05363, 2022.

Appendix

1. Definitions: iDPF, DCF, DDCF

Incremental Distributed Point Functions. Introduced by Boneh et al. [BBCG+21], incremental distributed point functions (iDPF) are a generalization of the standard distributed point function (DPF). At a high level, a DPF is a compressed pseudorandom 2-party secret-sharing of a unit vector of length 2^n. More specifically, DPF allows a compressed 2-party secret-sharing of a point function $f_{\alpha,\beta}$ where $\alpha \in \{0,1\}^n, \beta \in \mathbb{F}$, and:

$$f_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$$

Such a secret sharing is represented by a pair of keys (k_0, k_1) where key k_b is the share held by Party P_b. Incremental DPFs (iDPF) are a generalization of DPF which

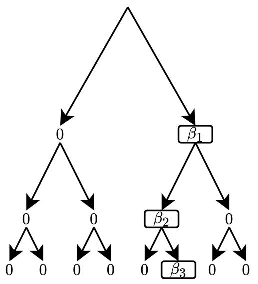


Figure 3: Incremental DPF gives compact secret sharing of values on the nodes of a binary tree with a single non-zero path. In this example, $\alpha = 101$ and the values on the path to the leaf at index α are $\beta_1, \beta_2, \beta_3$. All other nodes are 0. This figure shows the reconstructed secret shares $\mathsf{Eval}(0, k_0, \cdot) \oplus \mathsf{Eval}(1, k_1, \cdot)$. The keys are generated as $(k_0, k_1) \leftarrow \mathsf{Gen}^{\mathsf{idpf}}(\alpha, \beta_1, \beta_2, \beta_3)$.

allow compressed sharing of a binary tree with 2^n leaves and a unique special path from root to leaf. I.e., there is a single non-zero path in the tree, ending at leaf α, whose nodes have non-zero values $\beta_1, \ldots, \beta_n$. More specifically, iDPF allows a 2-party secret-sharing of an *all-prefix point* function $f_{\alpha,\bar{\beta}}$, where $\alpha \in \{0,1\}^n, \bar{\beta} = ((\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n))$, and for each $l \in [n]$:

$$f_{\alpha,\bar{\beta}} : \bigcup_{l \in [n]} \{0,1\}^l \to \bigcup_{l \in [n]} \mathbb{G}_l, \text{ and}$$

$$f_{\alpha,\bar{\beta}}(x_1, \ldots, x_l) = \begin{cases} \beta_l & \text{if } (x_1, \ldots, x_l) = (\alpha_1, \ldots, \alpha_l) \\ 0 & \text{otherwise} \end{cases}$$

We sometimes allow an iDPF to be evaluated over the empty prefix. We now present iDPF formally, see Figure 3 for more intuition. We closely follow the definitions of Boneh et al. [BBCG+21], with on major difference being that we expose the EvalNext function as part of our definition. We will use this in our reduction from distributed comparison functions to iDPFs.

Definition 2. A 2-party iDPF scheme is a tuple of three algorithms $(\mathsf{Gen}^{\mathsf{idpf}}, \mathsf{EvalNext}^{\mathsf{idpf}}, \mathsf{EvalPrefix}^{\mathsf{idpf}})$ such that:

- $\mathsf{Gen}^{\mathsf{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n)))$ is a PPT *key generation* algorithm that given security parameter 1^λ and a function description $(\alpha, (\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n))$, outputs a pair of keys and public parameters $(k_0, k_1, \mathsf{pp} = (\mathsf{pp}_1, \ldots, \mathsf{pp}_n))$.
  Recall that $\alpha \in \{0,1\}^n$ represents the index of the leaf at the bottom of the non-zero path while $\beta_1 \in \mathbb{G}_1, \ldots, \beta_n \in \mathbb{G}_n$ correspond to the values on the nodes of the non-zero path (apart from the root node). pp includes the public values $\lambda, n, (\mathbb{G}_1, \ldots, \mathbb{G}_n)$.
- $\mathsf{EvalNext}^{\mathsf{idpf}}(b, \mathsf{st}_b^{l-1}, \mathsf{pp}_l, x_l)$ is a polynomial time *incremental evaluation* algorithm that given a party id $b \in \{0,1\}$, secret state st_b^{l-1}, public parameters pp_l, and input evaluation bit $x_l \in \{0,1\}$, outputs an updated state and output share (st_b^l, y_b^l).

Intuitively, $\mathsf{EvalNext}$ represents the evaluation on some partial value $x \in \{0,1\}^{l-1}$ and outputs a secret sharing y_b^l of the value on the $x||x_l$th node of the binary tree and an updated state st_b^l.

- $\mathsf{EvalPrefix}^{\mathsf{idpf}}(b, k_b, \mathsf{pp}, (x_1, \ldots, x_l))$ is a polynomial time *prefix evaluation* algorithm that given a party id $b \in \{0,1\}$, iDPF key k_b, public parameters pp, and input prefix $(x_1, \ldots, x_l) \in \{0,1\}^l$, outputs an additive secret sharing of the output value y_b^l.

Next, we present iDPF correctness and security.

Definition 3. $(\mathsf{Gen}, \mathsf{EvalNext}, \mathsf{EvalPrefix})$ from Definition 2 is an iDPF scheme if it satisfies the following requirements:

- **Correctness**. For all $\lambda, n \in \mathbb{N}, \alpha \in \{0,1\}^n$, abelian groups and values $\bar{\beta} = ((\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n))$, level $l \in [n]$, and input prefix $(x_1, \ldots, x_l \in \{0,1\}^l)$, the following requirements hold:
  - $\mathsf{EvalNext}$: $Pr[y_0^l + y_1^l = f_{\alpha,\bar{\beta}}(x_1, \ldots, x_l)] = 1$, where probability is taken over: $(k_0, k_1, \mathsf{pp}) \leftarrow \mathsf{Gen}^{\mathsf{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n)))$, And for each $b \in \{0,1\}$, y_b^l is:
    ```
    1) $\mathsf{st}_b^0 \leftarrow k_b$
    2) for $j = 1$ to $l$:
    3)     $(\mathsf{st}_b^j, y_b^j) \leftarrow \mathsf{EvalNext}^{\mathsf{idpf}}(b, \mathsf{st}_b^{j-1}, \mathsf{pp}_j, x_j)$
    4) return $y_b^l$
    ```
  - $\mathsf{EvalPrefix}$: $Pr[y_0^l + y_1^l = f_{\alpha,\bar{\beta}}(x_1, \ldots, x_l)] = 1$, where probability is taken over: $(k_0, k_1, \mathsf{pp}) \leftarrow \mathsf{Gen}^{\mathsf{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n)))$, And for each $b \in \{0,1\}$: $y_b^l \leftarrow \mathsf{EvalPrefix}^{\mathsf{idpf}}(b, k_b, \mathsf{pp}, (x_1, \ldots, x_l))$
- **Security**. For every $b \in \{0,1\}$, there is a PPT simulator Sim_b, such that for every sequence $((\alpha, \bar{\beta})_\lambda)_{\lambda \in \mathbb{N}}$ of polynomial size all-prefix point functions and polynomial size input sequence x_λ, the outputs of the Real and Ideal experiments are computationally indistinguishable:
  - Real_λ: $(k_0, k_1, \mathsf{pp}) \leftarrow \mathsf{Gen}^{\mathsf{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n)))$,
    Output (k_b, pp)
  - Ideal_λ:
    Output $Sim_b(1^\lambda, (n, \mathbb{G}_1, \ldots, \mathbb{G}_n))$

A naive approach to constructing iDPF would be to generate one DPF key for each prefix length, i.e. a total of n independent keys. Then, evaluate $x \in \{0,1\}^l$ with the lth key. This solution would yield key size *quadratic* in the input length n. [BBCG$^+$21] gives a more direct construction with key size linear in n.

Theorem A.1 (Concrete cost of iDPF [BBCG$^+$21]). *Given a PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$, there exists a iDPF scheme with key-size $\lambda + (\lambda+2)n + \sum_{i\in[n]} m_i$ bits, where n is the bit-length of α and m_i is the bit-length of β_i. For $m'_i = 1 + \lceil m_i/\lambda \rceil$, the key generation algorithm* Gen *invokes G at most $2\sum_{i\in[n]} m'_i$ times and the algorithm* Eval *invokes G at most $\sum_{i\in[|x|]} m'_i$ times.*

Distributed Comparison Function (DCF) A DCF is a central building block of many FSS gates including interval containment, spline, and comparison. It is a FSS scheme for a function $f^<_{\alpha,\beta}$, which outputs β if $x < \alpha$ and 0 otherwise. For a vector of size 2^n, the current most efficient construction has a key size $\approx n(\lambda + n)$ [BCG$^+$21].

In this work, we introduce a new simple DCF construction by black-box reducing it to iDPF. We believe this construction is of independent interest and present it in Appendix D.

Dual Distributed Comparison Function (DDCF) DDCF is a variant of DCF and a class of functions $f_{\alpha,\beta_1,\beta_2} : \{0,1\}^n \to \mathbb{G}$. Parameterized by α, β_1, β_2, DDCF outputs β_1 for $0 \le x < \alpha$ and β_2 for $x \ge \alpha$. DDCF can be constructed from DCF using $f_{\alpha,\beta_1,\beta_2} = \beta_2 + f^<_{\alpha,\beta_1-\beta_2}(x)$.

2. Functionalities based on FSS

We will now describe some of the functionalities which can be realized efficiently using Function Secret Sharing primitive. We note that gates based on FSS operate on masked inputs and produce masked outputs (instead of standard secret-sharing MPC gates which operates on input shares and produce output shares). Specifically, a masked value x_{mask} for a secret input x is computed as $x_{\mathsf{mask}} := x + r$, where r is a uniform random element from the same domain as x. The mask r is sampled during an offline phase and is used in constructing the pre-processing material for FSS based gates. As described in [BCG$^+$21], we can easily convert from a masked value to a secret-shared value by letting parties hold a secret-sharing of the mask from the offline phase.

Equality Gate. Let $x, y \in \mathbb{U}_N$ be inputs to the equality gate. The output is a Boolean sharing $\mathbf{1}\{x = y\}$. More formally:

$$\mathcal{F}_{\mathsf{EQ}}(x, y) \to (b_0, b_1)$$
$$\text{where } x, y \in \mathbb{U}_N$$
$$\text{and } b_0, b_1 \text{ is a Boolean sharing of bit } b := \mathbf{1}\{x = y\}$$

Boyle et. al. [BGI19] constructed an equality gate by making two observations. First, $x = y$ can be evaluated by zero-testing $x - y$, i.e. $\mathbf{1}\{x - y = 0\}$. Second, equality test can be reduced to a single DPF call. Recall that the inputs to FSS gates are masked. I.e., let x, y be the masked inputs and $\mathsf{r}_0^{\mathsf{in}}, \mathsf{r}_1^{\mathsf{in}}$ their masks. Then, equality holds when $x - \mathsf{r}_0^{\mathsf{in}} = y - \mathsf{r}_1^{\mathsf{in}}$, or equivalently, $x - y = \mathsf{r}_0^{\mathsf{in}} - \mathsf{r}_1^{\mathsf{in}}$. In other words, we evaluate a DPF function that evaluates to $\beta = 1$ when $\alpha = \mathsf{r}_0^{\mathsf{in}} - \mathsf{r}_1^{\mathsf{in}}$, 0 otherwise. We present the full construction in Algorithm 4.

Comparison Gate Let $x \in \mathbb{U}_N$, $y \in \mathbb{U}_N$ be inputs to the comparison gate. The output is a Boolean sharing $\mathbf{1}\{x < y\}$.

We present the comparison gate of Boyle et. al. [BCG$^+$21] in Algorithm 5. This comparison gate requires a single invocation of DDCF, and thus a single invocation of DCF. Note that we slightly modify the protocol to make it syntactically compatible with our secure comparison. I.e., we (1) write the comparison for $x < y$ rather than

Algorithm 4: FSS Gate for $\mathcal{F}_{\mathsf{EQ}}$

```
Input: P_0, P_1 hold $x_{\mathsf{mask}} := x + \mathsf{r}_0^{\mathsf{in}}$, where $x_{\mathsf{mask}} \in \mathbb{G}_1$,
  and $y_{\mathsf{mask}} := y + \mathsf{r}_1^{\mathsf{in}}$, where $y \in \mathbb{G}_2$
Output: P_0, P_1 learn a uniform boolean sharing
  $b_{\mathsf{mask}} = b \oplus \mathsf{r}^{\mathsf{out}}$, where $b := \mathbf{1}\{x = y\}$.

// Part I: Offline Phase.
$\mathsf{Gen}_n^{\mathsf{eq}}(1^\lambda, \mathsf{r}_0^{\mathsf{in}}, \mathsf{r}_1^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:
Let $\mathsf{r}_0^{\mathsf{in}} \in \mathbb{G}_1$ and $\mathsf{r}_1^{\mathsf{in}} \in \mathbb{G}_2$.
Let $\alpha \leftarrow \mathsf{r}_0^{\mathsf{in}} - \mathsf{r}_1^{\mathsf{in}}$, $\beta = 1$.
$k_0', k_1' \leftarrow \mathsf{Gen}^{\mathsf{DPF}}(1^\lambda, \alpha, \beta)$
Sample random additive shares $\mathsf{r}_0^{\mathsf{out}}, \mathsf{r}_1^{\mathsf{out}} \leftarrow [\![\mathsf{r}^{\mathsf{out}}]\!]$.
Let $k_b = k_b' || \mathsf{r}_{\mathsf{b}}^{\mathsf{out}}$.
return $(k_0, k_1)$

// Part II: Online Phase.
$\mathsf{Eval}_n^{\mathsf{eq}}(b, k_b, x_{\mathsf{mask}}, y_{\mathsf{mask}})$:
Parse $k_b = k_b' || \mathsf{r}_{\mathsf{b}}^{\mathsf{out}}$.
return $\mathsf{Eval}^{\mathsf{DPF}}(b, k_b', x_{\mathsf{mask}} - y_{\mathsf{mask}}) + \mathsf{r}_{\mathsf{b}}^{\mathsf{out}}$
```

[BCG+21]'s $x > y$ and (2) the output group is $\mathbb{U}_2$ instead of $\mathbb{U}_N$.

Algorithm 5: FSS Gate for $\mathcal{F}_{\mathsf{CMP}}^n$

```
Input: P_0, P_1 hold $x_{\mathsf{mask}} := x + \mathsf{r}_0^{\mathsf{in}}$, where $x_{\mathsf{mask}} \in \mathbb{G}_1$,
  and $y_{\mathsf{mask}} := y + \mathsf{r}_1^{\mathsf{in}}$, where $y \in \mathbb{G}_2$
Output: P_0, P_1 learn a uniform boolean sharing
  $b_{\mathsf{mask}} = b \oplus \mathsf{r}^{\mathsf{out}}$, where $b := \mathbf{1}\{x < y\}$.

// Part I: Offline Phase.
$\mathsf{Gen}_n^{\mathsf{cmp}}(1^\lambda, \mathsf{r}_0^{\mathsf{in}}, \mathsf{r}_1^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:
Let $y = (2^n - (\mathsf{r}_0^{\mathsf{in}} - \mathsf{r}_1^{\mathsf{in}})) \in \mathbb{U}_N$ and $\alpha^{(n-1)} = y_{[0,n-1)}$.
$(k_0^{(n-1)}, k_1^{(n-1)}) \leftarrow \mathsf{Gen}_{n-1}^{\mathsf{DDCF}}(1^\lambda, \alpha^{(n-1)}, \beta_1, \beta_2, \mathbb{U}_2)$,
  where $\beta_1 = 1 \oplus y_{[n-1]}, \beta_2 = y_{[n-1]} \in \mathbb{U}_2$.
Sample random $\mathsf{r}_0^{\mathsf{out}}, \mathsf{r}_1^{\mathsf{out}} \leftarrow \mathbb{U}_N$ s.t. $\mathsf{r}_0^{\mathsf{out}} \oplus \mathsf{r}_1^{\mathsf{out}} = \mathsf{r}^{\mathsf{out}}$.
For $b \in \{0,1\}$, let $k_b = k_b^{(n-1)} || \mathsf{r}_{\mathsf{b}}^{\mathsf{out}}$.
return $(k_0, k_1)$

// Part II: Online Phase.
$\mathsf{Eval}_n^{\mathsf{cmp}}(b, k_b, x_{\mathsf{mask}}, y_{\mathsf{mask}})$:
Parse $k_b = k_b^{(n-1)} || \mathsf{r}_{\mathsf{b}}^{\mathsf{out}}$.
Set $z = (x_{\mathsf{mask}} - y_{\mathsf{mask}}) \in \mathbb{U}_N$.
Set $m_b^{(n-1)} \leftarrow \mathsf{Eval}_{n-1}^{\mathsf{DDCF}}(b, k_b^{(n-1)}, z^{(n-1)})$, where
  $z^{(n-1)} = 2^{n-1} - z_{[0,n-1)} - 1$.
return $b \cdot z_{[n-1]} + m_b^{(n-1)} - 2 \cdot z_{[n-1]} \cdot m_b^{(n-1)} + \mathsf{r}_{\mathsf{b}}^{\mathsf{out}}$
```

Multiple Interval Containment (MIC) gate. Boyle et. al. [BCG+21] presented an FSS gate for the $\mathcal{F}_{\mathsf{MIC}}$ functionality. Such a functionality is parameterized by a set of m intervals $\{p_i, q_i\}_{i \in [m]}$ where $p_i, q_i \in \mathbb{U}_N$. It takes as input a masked value x_{mask}, and outputs a sequence of bits $\{b_i\}$ where $b_i = \mathbf{1}\{p_i \leq x \leq q_i\}$.

Algorithm 6: FSS Gate for $\mathcal{F}_{\mathsf{MIC}}$

```
Input: P_0, P_1 hold $x_{\mathsf{mask}} := x + \mathsf{r}_0^{\mathsf{in}}$, where $x_{\mathsf{mask}} \in \mathbb{G}_1$,
  and $y_{\mathsf{mask}} := y + \mathsf{r}_1^{\mathsf{in}}$, where $y \in \mathbb{G}_2$
Output: P_0, P_1 learn a uniform arithmetic sharing of
  $b_{i\mathsf{mask}} = b_i + \mathsf{r}_{\mathsf{i}}^{\mathsf{out}}$, where $b_i := \mathbf{1}\{p_i \leq x \leq q_i\}$.

// Part I: Offline Phase.
$\mathsf{Gen}_{n,m,\{p_i,q_i\}_i}^{\mathsf{mic}}(1^\lambda, \mathsf{r}^{\mathsf{in}}, \{\mathsf{r}_{\mathsf{i}}^{\mathsf{out}}\}_{i \in [m]})$:
Let $\gamma = (N-1) + \mathsf{r}^{\mathsf{in}}$
$(k_0^{(N-1)}, k_1^{(N-1)}) \leftarrow \mathsf{Gen}_n^{\mathsf{DCF}}(1^\lambda, \gamma, 1, \mathbb{U}_N)$
for $i = 1$ to $m$ :
    Set $q_i' = q_i + 1$, $\alpha_i^{(p)} = p_i + \mathsf{r}^{\mathsf{in}}$, $\alpha_i^{(q)} = q_i + \mathsf{r}^{\mathsf{in}}$,
      $\alpha_i^{(q')} = q_i + 1 + \mathsf{r}^{\mathsf{in}}$.
    Sample random $z_{i,0}, z_{i,1} \leftarrow \mathbb{U}_N$ such that:
      $z_{i,0} + z_{i,1} = \mathsf{r}^{\mathsf{out}} + \mathbf{1}\{\alpha_i^{(p)} > \alpha_i^{(q)}\} - \mathbf{1}\{\alpha_i^{(p)} > p_i\} +$
      $\mathbf{1}\{\alpha_i^{(q')} > q_i'\} + \mathbf{1}\{\alpha_i^{(q)} = N-1\}$
For $b \in \{0,1\}$, let $k_b = k_b^{(N-1)} || \{z_{i,b}\}_i$
return $(k_0, k_1)$

// Part II: Online Phase.
$\mathsf{Eval}_{n,m,,\{p_i,q_i\}_i}^{\mathsf{mic}}(b, k_b, x_{\mathsf{mask}})$:
Parse $k_b = k_b^{(N-1)} || \{z_{i,b}\}_i$.
for $i = 1$ to $m$ :
    Set $q_i' = q_i + 1 \bmod N$.
    Set $x_i^{(p)} = x + (N - 1 - p_i)$ and
      $x_i^{(q')} = x + (N - 1 - q_i')$.
    Set $s_{i,b}^{(p)} \leftarrow \mathsf{Eval}_n^{\mathsf{DCF}}(b, k_b^{(N-1)}, x_i^{(p)})$.
    Set $s_{i,b}^{(q')} \leftarrow \mathsf{Eval}_n^{\mathsf{DCF}}(b, k_b^{(N-1)}, x_i^{(q')})$.
    $y_{i,b} = b \cdot (\mathbf{1}\{x_{\mathsf{mask}} > p_i\} - \mathbf{1}\{x_{\mathsf{mask}} >$
      $q_i'\} - s_{i,b}^{(p)} + s_{i,b}^{(q')} + z_{i,b})$.
    return $\{y_{i,b}\}_i$
```

3. Secure Comparison protocol benchmarks

In Table 6, we benchmark the offline and online communication costs of our new comparison protocol $\prod_{\mathsf{CMP}}$ for different values of l (the bit length of the comparison inputs). We set $m = 16$ as the parameter in our secure comparison protocol. We compare it against CrypTFlow2 [RRK+20b] (where we set $m = 4$ as used by the authors[7]), SynCirc [PSSY21b] and Couteau's protocol [Cou18]. Note that if $l \leq m$, then we set $m = l$. Our offline costs exclude the cost of base oblivious transfers. For SynCirc [PSSY21b], we could not obtain values for $l = 4, 8, 128$ bits as the paper doesn't report costs for these cases.

4. Black-Box Reduction from DCF to iDPF

We now describe our reduction from DCFs to iDPFs. Our construction is based on the following intuition. Suppose the two parties have shares $[\![v_{n-1}]\!]$ of an $(n-1)$-bit DCF $f^{<}_{\alpha_1 \dots \alpha_{n-1}, \beta}$ evaluated at the $n-1$-bit prefix $x_1, ..., x_{n-1}$ of x. They now want to get $[\![v_n]\!]$, i.e., shares of the output of the n-bit DCF $f^{<}_{\alpha,\beta}$ on input x. There are four cases.

1) $x_1, \dots, x_{n-1} \neq \alpha_1, \dots, \alpha_{n-1}$. Then no matter what α_n and x_n are, $v_n = v_{n-1}$.
2) $x_1, \dots, x_{n-1} = \alpha_1, \dots, \alpha_{n-1}$, and $\alpha_n = 0$. Then no matter what x_n is, $x \geq \alpha$, and so $v_n = v_{n-1} = 0$.
3) $x_1, \dots, x_{n-1} = \alpha_1, \dots, \alpha_{n-1}$, and $\alpha_n = 1, x_n = 1$. Then $x = \alpha$ and therefore $v_n = v_{n-1} = 0$.

7. Higher values of m in CrypTFlow2 lead to an exponential increase in the online communication cost.

Table 6: Concrete communication and round costs of our comparison protocol vs. prior works as functions of l. The offline costs exclude the costs of base OTs.

	Our Approach (m = 16)		CrypTFlow2 ($m = 4$) [RRK+20b]		SynCirc [PSSY21b]		Couteau16 [Cou18]	
l	comm.	rounds	comm.	rounds	comm.	rounds	comm.	rounds
				Offline Phase				
4	0.75 KB	80 rounds	**0.03 KB**	**1 round**	-	-	0.19 KB	2 rounds
8	1.51 KB	80 rounds	**0.08 KB**	**1 round**	-	-	0.44 KB	2 rounds
16	3.02 KB	80 rounds	**0.19 KB**	**1 round**	3.43 KB	1 round	1.02 KB	2 rounds
32	9.07 KB	80 rounds	**0.43 KB**	**1 round**	8.82 KB	1 round	1.85 KB	3 rounds
64	21.74 KB	80 rounds	**0.93 KB**	**1 round**	16.07 KB	1 round	3.83 KB	3 rounds
128	46.71 KB	80 rounds	**1.95 KB**	**1 round**	-	-	6.36 KB	3 rounds
				Online Phase				
4	**8 bits**	**1 round**	20 bits	2 rounds	-	-	30 bits	2 rounds
8	**16 bits**	**1 round**	60 bits	3 rounds	-	-	162 bits	6 rounds
16	**32 bits**	**1 round**	142 bits	4 rounds	84 bits	3 rounds	308 bits	6 rounds
32	**100 bits**	**2 rounds**	308 bits	5 rounds	178 bits	3 rounds	530 bits	12 rounds
64	**242 bits**	**3 rounds**	642 bits	6 rounds	316 bits	4 rounds	1120 bits	12 rounds
128	**522 bits**	**3 rounds**	1312 bits	7 rounds	-	-	2101 bits	12 rounds

4) $x_1, \dots, x_{n-1} = \alpha_1, \dots, \alpha_{n-1}$, and $\alpha_n = 1, x_n = 0$. Then $v_{n-1} = 0$, but $v_n = \beta$.

Observe that only in the last case, $v_n \neq v_{n-1}$, and more precisely, $v_n = v_{n-1} + \beta$. Now if we can construct shares of a value δ, such that $\delta = 0$ in cases (1)–(3), and $\delta = \beta$ in case (4), then $v_n = v_{n-1} + \delta$, which allows us to recursively build a DCF for arbitrary n.

Our main observation is that we can use a $n-1$-bit DPF, evaluated on $x_1, \dots, x_{n-1}$, to obtain shares of δ. Observe that in case (1), any DPF will satisfy $\delta = 0$. To distinguish between case (2) on one side, and (3) and (4) on the other, we only need to look at α_n, and set the DPF value to be 0 when $\alpha_n = 0$, and β otherwise. Finally, observe that the distinction between (3) and (4) can be made at evaluation time, since it only depends on x. That is, we only use the DPF result at all if $x_n = 0$, and set $\delta = 0$ otherwise.

Algorithm 7 shows our construction in detail. In addition to the two DPF keys, the two parties obtain an additional secret-shared value, which can be interpreted as the iDPF evaluation at the empty prefix. It is used to initialize v_1. For $i = 2, \dots, n$, v_i is then constructed from $v_i - 1$ and $\delta = (1 - x) \cdot y_i$, where y_i is the iDPF evaluation at level i. Correctness follows by the above recursion argument.

Theorem A.2 (Concrete cost of DCF using iDPF). *Given a PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$, there exists a DCF scheme with key-size $n(\lambda+m+2)-2$ bits, where n is the bit-length of α and m is the bit-length of β. For $m' = 1 + \lceil m/\lambda \rceil$, the key generation algorithm* Gen *invokes G at most $2(n-1)m'$ times and the algorithm* Eval *invokes G at most $(n-1)m'$ times.*

Efficiency. Note that in our reduction, β_i at each level of iDPF is either set to β or 0. Therefore, for all $i \in [n]$, $|\beta_i| = |\beta| = m$.

Following from Theorem A.1 and the fact that we can set the iDPF domain size to be $n - 1$ (instead of n), the key-size turns out to be $\lambda + (\lambda + 2)(n - 1) + (n - 1)m$ bits. Since we require an additional sharing of β_1, the total DCF key size becomes $\lambda + (\lambda + 2)(n - 1) + (n - 1)m + m$ bits which simplifies to $n(m + \lambda + 2) - 2$ bits.

The cost of $\mathsf{Gen}_{\mathsf{DCF}}$ and $\mathsf{Eval}_{\mathsf{DCF}}$ algorithms can be computed based on the underlying cost of $\mathsf{Gen}_{\mathsf{iDPF}}$ and $\mathsf{Eval}_{\mathsf{iDPF}}$ algorithms. Following from the Theorem A.1 and the fact that we can set the domain size of iDPF to be $n-1$, the total PRG invocations in $\mathsf{Gen}_{\mathsf{iDPF}}$ (and hence in $\mathsf{Gen}_{\mathsf{DCF}}$) turns out to be $2(n - 1)m'$ where $m' = 1 + \lceil m/\lambda \rceil$. In $\mathsf{Eval}_{\mathsf{DCF}}$, we perform an $\mathsf{EvalNext}_{\mathsf{iDPF}}$ at each of the $n - 1$ prefixes of the input x which will cost $\sum_{j \in [2,n]} m' = (n - 1)m'$ PRG evaluations. $\square$

Algorithm 7: DCF to iDPF reduction

```
$\mathsf{Gen}^{\mathsf{DCF}}_n(1^\lambda, \alpha, \beta)$ :
1 Let $\alpha = \alpha_1, \dots, \alpha_n \in \{0,1\}^n$ be the bit decomposition
    of $\alpha$
2 Let $\{\beta_1, \dots, \beta_n\}$ be a sequence of values such that:
    $\beta_i := \beta$ if $\alpha_i = 1$, and 0 otherwise.
3 $(k_0, k_1, \mathsf{pp}) \leftarrow \mathsf{Gen}^{\mathsf{iDPF}}_{n-1}(\alpha, \beta_2, \dots, \beta_n)$
4 Choose random $[\![\beta_1]\!]^0$, $[\![\beta_1]\!]^1$ such that
    $[\![\beta_1]\!]^0 + [\![\beta_1]\!]^1 = \beta_1$.
5 return $\big((k_0, [\![\beta_1]\!]^0), (k_1, [\![\beta_1]\!]^1), \mathsf{pp}\big)$

$\mathsf{Eval}^{\mathsf{DCF}}_n\big(b, (k_b, [\![\beta_1]\!]^b), \mathsf{pp}, x\big)$ :
1 Let $x = x_1, \dots, x_n \in \{0,1\}^n$ be the bit decomposition
    of $x$
2 Let $v_1 = (1 - x_1) \cdot [\![\beta_1]\!]^b, \mathsf{st}_1 = k_b$
3 for $i = 2$ to $n$ :
4     $(\mathsf{st}_i, y_i) \leftarrow \mathsf{EvalNext}^{\mathsf{iDPF}}_{n-1}(b, \mathsf{st}_{i-1}, k_b, x_1 \dots x_{i-1})$
5     $v_i \leftarrow v_{i-1} + (1 - x_i) \cdot y_i$
6 return $v_{n-1}$
```

Comparison with original DCF construction. Boyle et al. [BCG+21] presented a direct construction of DCF by carefully modifying and making non black-box changes to a prior DPF construction [BGI16]. We provide a conceptually simpler DCF construction by making black-box use of iDPFs (which have a richer structure than DPF). As an added benefit, the key size of our DCF construction is smaller than Boyle et al. [BCG+21] by $\lambda + m + 2$ bits. In terms of computation, our construction doesn't require any PRG evaluations at the first bits, and so it saves $m' = \lfloor m/\lambda \rfloor$ PRG evaluations.

5. Efficient 2PC generation of FSS keys

As we have seen, our secure comparison protocol invokes FSS primitives such as DPF, DCF and iDPF. Besides this, our secure spline protocol invokes $\mathcal{F}_{\mathsf{MIC}}$ which in-turn relies on a DCF. In order to implement the offline phase of our protocol, we also need to generate keys for these FSS primitives efficiently in a 2PC setting. Note that as described in Appendix D, DCF can be black-box reduced to iDPF. Furthermore, DPF is just a special case of a iDPF. So it suffices to design an efficient 2PC offline phase for generating iDPF keys.

A straight-forward way to generate these keys in MPC is to implement $\mathsf{Gen}^{\mathsf{iDPF}}$ using a generic MPC compiler. This, however, has the drawback of requiring PRG calls inside the MPC, making this approach inefficient in practice. [Ds17] present a construction that does not require secure PRG evaluations. While, it comes at a computation cost that is linear in the domain size (i.e., exponential in the input size), and its round complexity is linear in the input size, it is still efficient enough in our case, where the domain for any single DCF is small.

However, the original Doerner-shelat construction is not sufficient to obtain FSS keys that generate arithmetic shares for domains larger than one bit. This is often the format required to compose with other secret-sharing-based MPC protocols, which is also the case for our construction. Specifically, this is needed when we invoke the MIC gate as part of our secure spline protocol.

While one option is to convert from Boolean to arithmetic shares after the DPF evaluation in the online computation, this would require additional rounds of interaction and communication. In the spirit of reducing online communication as far as possible without sacrificing offline performance, we instead develop a new construction for generating DPF keys with arithmetic output shares directly.

Also note that while previous work [BCG+21] claims a construction of Doerner-shelat for DCFs with arbitrary output groups, their construction is missing a crucial step, namely the computation of t^* in Step 10 of Fig. 9 of [BCG+21]. The main challenge for this construction is the fact that in order to compute the value correction words included in the DPF keys, the parties need to identify which one of them holds share 1 and which one holds share 0 of the control bit corresponding to the node on the evaluation path at every level. There are 2^l nodes at level l, and each party can locally evaluate its shares for all nodes, but the parties do not know which node is on the evaluation path.

So we need to implement this oblivious selection of the shares of appropriate node whose index is shared between the two parties. We leverage the following observation. The value of the control bit is one only for nodes that lie on the evaluation path and is zero for all other nodes. Since we have binary shares, this means that for all nodes not on the evaluation path, the shares of the two parties are equal. This means that if each party sums up its shares for the control bits of all nodes in the last level, the resulting values will differ by one and the party who has the larger value holds a share 1 of the control bit of the evaluation path node in the last level, while the other party has share 0.

We can solve the problem by comparing the two sums of shares of control bits at the last level, but in as we are trying to generate these DPF keys in order to solve a comparison problem more efficiently, so this is less satisfying. Our second observation is that since these values differ just by one, it is sufficient to consider only their last two bits to compute the comparison bit. This allows us to compute t^* using a single AND-Gate.

We present the details our Doerner-shelat construction for iDPFs with arbitrary output groups in Algorithm 8. The two parties hold secret shares of α and $\{\beta_i\}_{i\in[n]}$, and would like to generate the iDPF keys for $f_{\alpha,\{\beta_i\}_{i\in[n]}}$. In order to get a protocol for distributed DCF key generation, observe that we only need to compute shares $\beta_1, \dots \beta_n$ in Algorithm 7 given $\alpha_1, \dots, \alpha_n$ and β. As $\beta_i = \alpha_i \cdot \beta$, this reduces to n parallel calls to $\mathcal{F}_{\mathsf{MUX}}$. Finally, observe that in groups where $-x = x$ (such as boolean sharing), $[\![W^0_{CW}]\!] = [\![W^1_{CW}]\!]$ in Step 11, and so the last $\mathcal{F}_{\mathsf{MUX2}}$ call can be saved in that case, making the entire second MPC linear.

6. Adding Differential Privacy

In this section, we discuss how our solution can also provide differential privacy for its output, which limits the leakage from the final model about individual training samples. As we mentioned in the introduction, our approach allows that the two computation parties obtain cryptographic shares of the logistic regression parameters which they use to jointly answer inference queries. So one option for enabling differential privacy will be at that query level.

However, we consider here the case where the trained regression model is released to a single party and the goal is to guarantee DP for the model parameters. Since our training construction used SGD, we will also use the DP-SGD approach introduced by Abadi et al. [ACG+16] for general SGD ML training and the instantiation of Jayaraman et. al. [JWEG18] for the setting of logistic regression presented in Algorithm 9. Jayaraman et. al. [JWEG18] provides a two party computation protocol for secure training of logistic regression when the input data is horizontally partitioned between the two parties. We adapt their framework to the setting where the input is fully secret-shared between the two parties.

In Algorithm 10 we give the pseudocode for implementing the DP-SGD algorithm in MPC. The MPC protocol is similar to the non-DP algorithm in Algorithm 1, except in

Algorithm 8: Secure Distributed $\mathsf{Gen}^{\mathsf{iDPF}}$

```
Inputs: Each party holds additive shares of $\alpha \in \{0,1\}^n$ (bitwise) and $\{\beta_i\}_{i\in[n]}$ where $\beta_i \in \mathbb{G}_i$
Output: iDPF keys for $f_{\alpha,\{\beta_i\}_{i\in[n]}}$
Parameters: Let $G : \{0,1\}^\lambda \to \{0,1\}^{2(\lambda+1)}$ and $\mathsf{Convert} : \{0,1\}^\lambda \to \{0,1\}^{\lambda+1}$ be PRGs.

Each party $\mathsf{P}_b$ performs the following:
1  Sample $s_b^\emptyset \in \{0,1\}^\lambda$, set $t_b^\emptyset = b$.
2  for $l$ = 1 to $n$ :
3      For all $w \in \{0,1\}^{l-1}$, compute
         $s_b^{w,L}||t_b^{w,L}||s_b^{w,R}||t_b^{w,R} = G(s_b^w)$.
4      Compute $s_b^L||t_b^L||s_b^R||t_b^R =$
         $\bigoplus_{w\in\{0,1\}^{l-1}} s_b^{w,L}||t_b^{w,L}||s_b^{w,R}||t_b^{w,R}$.
5      Secure Computation:
       - Inputs: Boolean sharing of $\alpha_l$, arithmetic sharing of
         $\{s_b^L, s_b^R, t_b^L, t_b^R\}_{b\in\{0,1\}}$.
       - Compute:
           $[\![s^R]\!] \leftarrow [\![s_0^R]\!] \oplus [\![s_1^R]\!]$
           $[\![s^L]\!] \leftarrow [\![s_0^L]\!] \oplus [\![s_1^L]\!]$
           $[\![s_{CW}]\!] \leftarrow \mathcal{F}_{\mathsf{MUX2}}\Big([\![s^R]\!], [\![s^L]\!], [\![\alpha_l]\!]\Big)$
           $[\![t_{CW}^L]\!] \leftarrow [\![t_0^L]\!] \oplus [\![t_1^L]\!] \oplus [\![\alpha_l]\!] \oplus [\![1]\!]$
           $[\![t_{CW}^R]\!] \leftarrow [\![t_0^R]\!] \oplus [\![t_1^R]\!] \oplus [\![\alpha_l]\!]$
       - Output $s_{CW}, t_{CW}^L, t_{CW}^R$ to both
6      For all $w \in \{0,1\}^{l-1}$, set
         $\tilde{s}_b^{w||0}||\tilde{s}_b^{w||1} \leftarrow (s_b^{w,L}||s_b^{w,R}) \oplus t_b^w \cdot (s^{CW}||s^{CW})$
7      For all $w \in \{0,1\}^{l-1}$, set
         $t_b^{w||0}||t_b^{w||1} \leftarrow (t_b^{w,L}||t_b^{w,R}) \oplus t_b^w \cdot (t_{CW}^L||t_{CW}^R)$
8      For all $w \in \{0,1\}^l$, set $s_b^w||W_b^w \leftarrow \mathsf{Convert}(\tilde{s}_b^w)$
9      Compute $W_b^l \leftarrow \sum_{w\in\{0,1\}^l} W_b^w$.
10     Compute $T_b^l \leftarrow b + (-1)^b \cdot \sum_{w\in\{0,1\}^l} t_b^w$.
         Let $\tau_b^0$ and $\tau_b^1$ denote the two least significant bits of
         $T_b$.
11     Secure Computation:
       - Inputs: Arithmetic sharing of $\beta_l$, private inputs
         $W_b^l, \tau_b^0, \tau_b^1$ for Party $\mathsf{P}_b$.
       - Compute:
           $[\![t^*]\!] \leftarrow 1 \oplus \tau_0^1 \oplus \tau_1^1 \oplus (\tau_0^0 \cdot \tau_1^0)$
           $[\![W_{CW}^0]\!] \leftarrow [\![\beta_l]\!] - W_0^l + W_1^l$
           $[\![W_{CW}^1]\!] \leftarrow -[\![\beta_l]\!] + W_0^l - W_1^l$
           $[\![W_{CW}]\!] \leftarrow \mathcal{F}_{\mathsf{MUX2}}\Big([\![W_{CW}^0]\!], [\![W_{CW}^1]\!], [\![t^*]\!]\Big)$.
       - Output $W_{CW}$ to both
12     Set $CW^l \leftarrow s_{CW}||t_{CW}^L||t_{CW}^R||W_{CW}$
13 Output $k_b \leftarrow s_b^\emptyset||CW^1||\ldots||CW^n$
```

each iteration, the computation parties make the gradient differentially private using noise perturbation. We assume that this noise is generated in an offline phase where computation parties get secret shares for noise vectors. In the online phase, they add these shares of noise to the gradient update. Techniques for two party generation of DP noise were presented by Dwork at al. [DKM+06] and Champion et al. [CsU19].

If we only want to guarantee DP from the output of the secure logistic regression training, then we can reveal the DP gradient update to the two computation parties as shown in Algorithm 10. This would enable some efficiency optimization replacing a secure matrix multiplication with a plaintext matrix multiplication. While this approach still provides DP for the output, it is not known what is the exact privacy comparison between revealing only the final DP output model and all intermediate DP gradient updates. However, recent works [CYS21], [YS22] show that keeping the DP-SGD intermediate states hidden allows for faster convergence and spending less privacy budget for strongly convex loss functions for noisy stochastic gradient descent. Our DP secure computation training algorithm supports hiding these intermediate states at the same online communication cost.

Algorithm 9: DP SGD

```
Public inputs: Number of iterations $T$, Dataset size $n$, Batch size $B$, Lipschitz value $G = 1$, Smoothness value $L = 0.25$, Learning rate $\alpha = 1/L$, DP parameters $\epsilon$ and $\delta$
Private inputs: Dataset $\mathbf{X}, \mathbf{y}$ having $k$ features
1  Let $\mathbf{w_0}$ be the initial model with arbitrary weights
2  for $t$ = 1 to $T$ :
3      Compute gradient
         $\mathbf{g_t} \leftarrow \frac{1}{B}\mathbf{X}_B^T \times (\mathsf{Sigmoid}(\mathbf{X}_B \times \mathbf{w_{t-1}}) - \mathbf{Y_B})$
4      Perturb gradient $\widetilde{\mathbf{g_t}} \leftarrow \mathbf{g_t} + \mathcal{N}(0, \sigma^2 I_p)$ where
         $\sigma^2 = \frac{8G^2T\log(1/\delta)}{n^2\epsilon^2}$
5      Update model $\mathbf{w_t} \leftarrow \mathbf{w_{t-1}} - \alpha \cdot \widetilde{\mathbf{g_t}}$
6  return $\mathbf{w}_T$
```

Jayaraman et. al. [JWEG18] also present an output-perturbation DP technique for logistic regression, which adds noise only to the final model, rather than at each level of gradient descent. We note that our original protocol in Algorithm 1 can easily be modified to use the output perturbation technique, by having both parties collaboratively generate shares of the output perturbation noise and add it to their respective shares of the output before revealing them.

As noted in [JWEG18], adding the noise iteratively to the gradient or directly to the output may have different impact on the accuracy of the final model depending on the setting, though adding noise iteratively generally results in more accurate models. We are able to support both options between Algorithms 1 (with output-perturbation at the end of training) and Algorithm 10.

Algorithm 10: DP-SGD Logistic Regression Protocol

```
Public inputs: Number of iterations $T$, dataset dimensions $n, k$, batch size $B$, Lipschitz value $G = 1$, smoothness value $L = 0.25$, learning rate $\alpha = 1/L$, DP parameters $\epsilon$ and $\delta$, regularization parameter $\lambda$.
Private inputs: Secret-shared dataset $[\![X]\!] \in R^{n\times k}$ and labels $[\![\mathbf{y}]\!] \in R^n$. Secret shares $[\![r_t]\!] \in R^k$ of noise drawn from $\mathcal{N}(0, \sigma^2 I_p)$, for each $t \in [T]$.
1  Let $\mathbf{w_0}$ be the initial model with arbitrary weights.
2  for $t = 1$ to $T$ :
3      for $b = 1$ to $\lfloor n/B \rfloor$ :
4          $i \leftarrow (b-1)\cdot B + 1$
5          $j \leftarrow \min(n, b\cdot B)$
6          $[\![X_B]\!] \leftarrow [\![X_{i...j}]\!]$
7          $[\![\mathbf{u}]\!] \leftarrow [\![X_B]\!] \cdot \mathbf{w_{t-1}}$
8          $[\![\mathbf{s}]\!] \leftarrow \mathcal{F}_{\mathsf{Sigmoid}}(\mathbf{u})$
9          $[\![\mathbf{d}]\!] \leftarrow [\![\mathbf{s}]\!] - [\![\mathbf{y}_{i...j}]\!]$
10         $[\![\mathbf{g}]\!] \leftarrow \mathcal{F}_{\mathsf{matMult}}([\![{X_B}^\top]\!], [\![\mathbf{d}]\!])$
11         $[\![\mathbf{w_t}]\!] \leftarrow [\![\mathbf{w_{t-1}}]\!] - (\alpha/B)\cdot([\![\mathbf{g}]\!] + \lambda\cdot[\![\mathbf{w_{t-1}}]\!]) + [\![r_t]\!]$
12         $\mathbf{w_t} \leftarrow \mathsf{Reconstruct}([\![\mathbf{w_t}]\!])$
13 return $\mathbf{w_T}$.
```

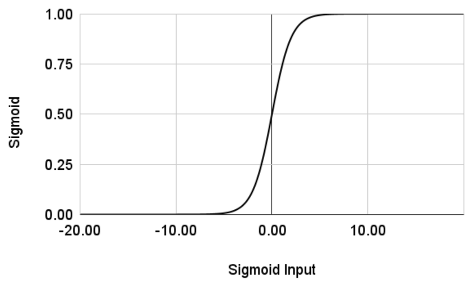


Figure 4: Plaintext Sigmoid Function

7. Additional Figures

In this section, we show plaintext sigmoid in Figure 4 and demonstrate how our plaintext sigmoid approximation compares to the plaintext approximation in SecureML in Figure 5. In Figure 6, we measure the absolute error (defined w.r.t. plaintext python implementation shown in Figure 4) for three different implementations: plaintext sigmoid approximation in fixed point (top), 2PC sigmoid with trusted offline setup (middle), and 2PC sigmoid with distributed offline setup (bottom). We do this experiment for input values in the range $[-20, 20]$ at increments of 0.1.

8. Bottleneck cost of Secure Logistic Regression

In each iteration of logistic regression, we perform sigmoid evaluations proportional to the batch size along with 2 correlated matrix multiplications (Line 7 and Line 10 in Algorithm 1). Assuming n training examples, batch size B, number of epochs T, we first discuss the cost associated with the correlated matrix multiplications.

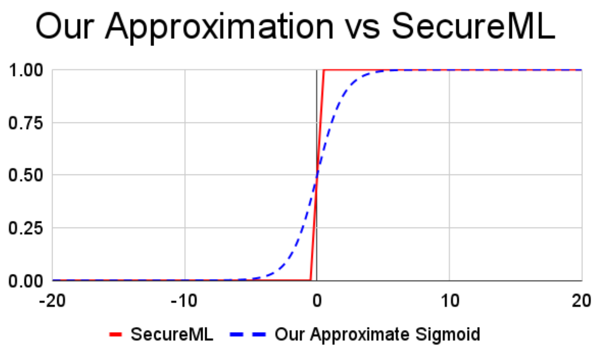


Figure 5: Plaintext Comparison of V1 sigmoid to SecureML's approximation.

In the online phase, there is a one-time cost of $2nk$ elements of communication (associated with the dataset X). Additionally, the per *iteration* (inner loop) communication cost of multiplying X_B with w_{t-1} (Line 7 in Algorithm 1) is $2k$ elements, and the cost of multiplying ${X_B}^\top$ with d (Line 10 in Algorithm 1) is $2B$ elements. Hence, we have a communication of $2k + 2B$ elements per iteration. Since there are $T \cdot \lfloor n/B \rfloor$ iterations, the total communication cost of matrix multiplications for the entire logisitic regression training comes out to be $2nk + T \cdot \lfloor n/B \rfloor\,(2k + 2B)$ elements.

Note that the sigmoid is invoked on B inputs per iteration (and n per epoch). Therefore, the total online cost of sigmoid across T epochs is $T \cdot n \cdot s$, where s is the number of elements communicated per sigmoid. Hence, sigmoid becomes a bottleneck whenever the following condition is satisfied:

$$T \cdot n \cdot s > 2nk + T \cdot (\frac{2kn}{B} + 2n)$$

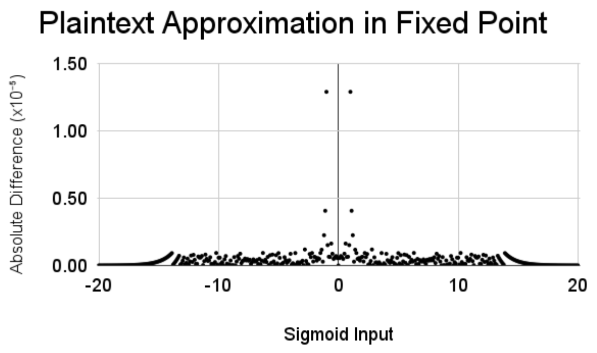
$$s > 2(\frac{k}{T} + \frac{k}{B} + 1)$$

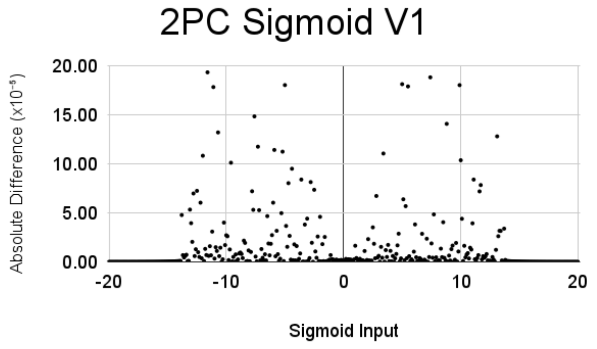
The above condition is often true for large datasets and/or when per sigmoid communication cost is high (which is true because of its nonlinear nature).

Note that in terms of latency (round complexity), the sigmoid computation dominates the matrix multiplication. This is because each matrix multiplication only requires 1 round of communication whereas accurate sigmoid approximation typically requires more rounds (in our case it requires 4 rounds for trusted offline (dealer) setting and 6 rounds for distributed (2PC) offline setting).

9. Optimized Dot Product

We compute sigmoid on the $[0, 1)$ interval by evaluating a spline of one degree polynomials of the form $a_i x + b_i$, where a_i and b_i are public coefficients. At a high level, we evaluate $[\![x]\!]$ on each interval i and then select only the interval output where x actually belongs. More specifically, each party can evaluate the spline on each interval with the





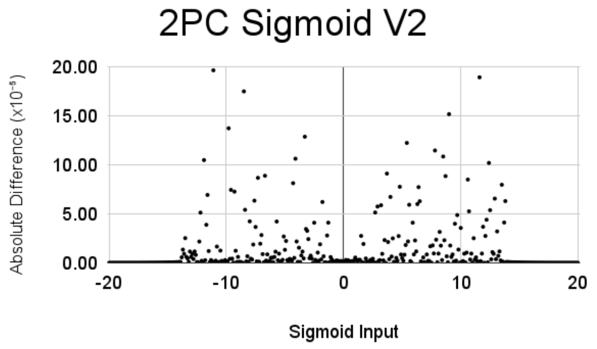


Figure 6: Absolute difference error ($\times 10^{-5}$) of our sigmoid approximation in plaintext fixed point representation (top), with trusted offline setup run in 2PC (middle), and with distributed offline setup run in 2PC (bottom).

same input $[\![x]\!]$ to get $[\![a_i x + b_i]\!]$ using local operations. For n intervals, P_0, P_1 hold:

$$[\![a_1 x + b_1]\!], \ldots, [\![a_n x + b_n]\!]$$

We then use a FSS multi-interval containment gate to get a sharing of one-hot encoded vector d, with 1 only at the interval t where the input belongs, 0 elsewhere. E.g., if x belongs to interval $t = 3$, P_0 and P_1 hold:

$$[\![d]\!] = [\![0, 0, 1, 0, \ldots, 0]\!]$$

Now we want to compute the dot product of these two vectors to get a sharing of evaluating x on the proper interval. Naively multiplying the two vectors pairwise requires communicating $4n$ ring elements. We now show how to reduce the communication to just 4 elements (i.e. independent of the number of intervals).

Note that a_i and b_i are public. Hence, P_0 and P_1 can locally compute:

$$[\![a_t]\!] \leftarrow [\![d_1]\!]a_1 + \ldots + [\![d_n]\!]a_n$$
$$[\![b_t]\!] \leftarrow [\![d_1]\!]b_1 + \ldots + [\![d_n]\!]b_n$$

Now P_0, P_1 do a single Beaver triple multiplication and compute:

$$[\![a_t x + b_t]\!]$$

Importantly, this single product requires communicating a total of only 4 ring elements.

10. Failure probability

The non-interactive fixed point truncation protocol from [MZ17] and the single round exponentiation protocol from [KLRS22] are probabilistic i.e. with some probability, that can be made arbitrary low by increasing the ring size, the output of these protocols can be incorrect. Since we use these two primitives as sub-protocols in our logistic regression protocol, it also induces an error probability on the overall training algorithm.

Each invocation of the non-interactive fixed point truncation protocol [MZ17] has an error probability of $p_{\mathsf{trunc}} = \frac{2^{w+1}}{2^l}$. We use this as a subprotocol in every instance of fixed point multiplication to adjust the scale. In each matrix multiplication, we truncate once after the accumulation (i.e. for multiplying matrix M_1 with matrix M_2, we do the usual Beaver multiplication without truncation to get a matrix M_3, and then truncate each element of M_3 by appropriate scale). This ensures (as pointed out in [MZ17]) that the probability of errors introduced due to truncation is low and the error union bound scales proportional to $|M_3|$ (instead of being proportional to $|M_1| \cdot |M_2|$ which would have been the case if we truncate before accumulating).

With that, we first computing the number of truncations in the logistic regression training due to matrix multiplications and sigmoid-specific operations. The number of truncations performed during the two matrix multiplications in logistic regression (line 7 and 10 in Algorithm 1) per iteration is $B + k$, i.e. depends on the size of the multiplication output. Recall that we evaluate sigmoid on 6 intervals. On two of these intervals, we perform independent Taylor approximations computed using the secure polynomial protocol. In each invocation of the polynomial protocol, we perform $\frac{d^2+d}{2}$ truncations per input in the batch, where d is the degree of the Taylor approximation. In our experiments, we set $d = 10$ which results in a total of $110 \cdot B$ truncations. On two other intervals, we invoke an independent instance of Secure Spline. In each spline invocation, we have one truncation per spline interval. In our experiments, we set the number of intervals to 10, which results in a total of $20 \cdot B$ truncations. Adding up the truncations from matrix multiplication, Taylor series approximation and spline invocation, we get a total of $131 \cdot B + k$ truncations per iteration.

Additionally, each exponentiation protocol from [KLRS22] has a failure probability $p_{\mathsf{exp}} \approx \frac{2^{w+1}}{2^l}$. We invoke the exponentiation protocol twice per input in the batch, hence a total of $2B$ exponentiations per iteration.

Now we can bound the total failure probability of one iteration of training by $2Bp_{\mathsf{exp}} + (131B + k)p_{\mathsf{trunc}}$ using union bound. Assuming t is the total number of iterations in the training and plugging the values of p_{exp} and p_{trunc}, we get a total failure probability bound across all iterations as $(133B + k) \cdot \frac{2^{w+1}}{2^l} \cdot t$. Compared to SecureML [MZ17], our failure probability only reduces the security by ≈ 7 bits while providing a much more accurate training (due to our better sigmoid approximation).