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 novel construction that enables two parties to securely 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 that results in a secure protocol with better communication, protocols for secure powers evaluation and secure spline computation on 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 batch of 10^3 sigmoids with ≈ 0.5 MB of online communication, 4 online rounds, and ≈ 1.6 seconds of online time over WAN. This is $\approx 30\times$ less in online communication, $\approx 31\times$ fewer online rounds, and $\approx 5.5\times$ less online time than the well-known MP-SPDZ's protocol. Our system can train a logistic regression model over 6 epochs and a database containing $70,000$ samples and 15 features with 208.09 MB of online communication and 9.68 minutes of online time. We compare our logistic regression training against MP-SPDZ over a synthetic dataset of 1000 samples and 10 features and show an improvement of $\approx 130\times$ in online communication and $\approx 4.75\times$ in online time over WAN. We converge to virtually the same model as plaintext in all cases. We open-source our system and include extensive tests.

1. Introduction

One of the most ubiquitous ways to understand and use large amounts of data is to train models that capture their most significant general properties. 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) [45], [82] 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 [11], 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 [9]. 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 [29] 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 [28] where the two aggregators

were run by Mozilla and the Internet Security Research Group (ISRG). The same design underlies the Private Analytics system implemented by Google and Apple in their Exposure Notifications system [5], 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 [48], 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 [47], considers a similar measurement goal, which is to compute aggregate measurements across attributed conversions from all users. Again, MPC with distributed data stewardship can be used for this kind of measurement [52].

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 [5]), 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 [15], [40], engineering [66], and finance [3] applications. It was the functionality of choice in Criteo's challenge for effective use of some of the privacy preserving APIs proposed by Chrome [44] and also in iDASH secure genome analysis competition [80]. While not as powerful as Deep Neural Networks (DNNs), it turns out to still be broadly useful for important applications so we focus on it in this work.

Online-offline Computation Model. Our constructions consider the online-offline computation model [39] 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. Another possible realization of this setting is in the well-known *three* party honest majority setting where one of the parties acts as the dealer for generating and distributing the offline preprocessing material across the remaining two parties which then carry out the online part of the secure computation.

The second setting that we address 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 [45], [82], 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 [36] is one approach to guarantee that it cannot be used to extract individual records. Thus, we also consider differentially private logistic regression training in our distributed protocol.

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 ($\approx 130\times$ reduction over MP-SPDZ [58]) 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 $\approx 30\times$ and $\approx 38\times$ smaller than the communication of the state-of-the-art sigmoid construction of MP-SPDZ [58] and SIRNN [72], respectively. Secure evaluation of a batch of 10^3 sigmoids requires 1-1.51 seconds over LAN and includes 0.5-1.18 MB 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, a novel all-prefix AND sub-protocol 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. The online communication for 128-bit numbers comparison is only 522 bits and has an improvement of $\approx 1.3 - 2.6\times$ over SynCirc. Secure comparison is a core building block in a broad range of functionalities far beyond the scope of this paper, such as auctions, database search, biometric authentication, combinatorial problems. This construction may therefore be of independent interest. As a concrete estimate of our improvement, our new secure comparison protocol can be used to execute Batcher's widely used sorting network [6] in a semi-honest 2-party setting on a 10^4 sized array of 64 bit integers using just 13.69 MB of communication whereas prior works such as SynCirc [68], CrypTFlow2 [73], Couteau16 [30] will require 17.88 MB, 36.32 MB and 63.37 MB respectively.

New Techniques for (i)DPFs. The communication and round efficiency of our constructions leverage (incremental) distributed point functions ((i)DPFs) [13], [19] techniques inside our MPC protocols in new ways. We present a new technique for computing the all-prefix Boolean AND of n secret shared bits using iDPFs, which achieves the round efficiency of our new secure 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. The latter is used for secure Taylor approximation by adapting a prior work [62] which works only for integers.

Implementation and Evaluation. We present end-to-end implementation of our protocols, which we open-source to the community[1]. Our implementation combines techniques from FSS and secret-sharing-based MPC, and includes extensive unit and end-to-end tests. We evaluate the costs of our logistic regression training algorithm and its building blocks such as our novel secure comparison and sigmoid approximation. We can train a model over cryptographically shared data of $70,000$ samples with 15 features in less than 10min with $\approx$ 200MB of communication, which amounts to 1.95c cost over WAN. We achieve accuracy close to the plaintext trained model (less than 1% difference). We reduce online communication over state of the art by $\approx 130\times$ for logistic regression training.

1.1. Our Approach

We outline the main ideas of our approach in this section and present detailed related work discussion towards the end in Section 8.

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 (Section 4 and Section 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 from function secret sharing [19] which enable non-interactive computation. In particular we use the multiple interval containment (MIC) gate [16] 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 E). MIC gates leverage distributed comparison functions (DCFs) [16] which rely on function secret sharing [20]. We introduce a reduction from DCFs to incremental distributed point functions (iDPFs) [13], which is conceptually simpler than the previous construction by [16].

Secure Powers Computation with Fixed-Point Representation (Appendix L.4). 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 secure exponentiation of Kelkar et al. [57]. For the second part we present a new construction inspired by the HoneyBadgerMPC secure powers protocol [62], which we extend to work with fractional values in fixed-point representation.

Online-Offline Balanced Protocol (Section 6). The most costly part of our offline computation is the generation of FSS keys, which are needed for MIC gates. In the setting without trusted preprocessing these keys need to be generated using two-party computation, presenting significant costs challenging the offline phase feasibility. Existing approaches either rely on general-purpose MPC, which is expensive because of the need to securely evaluate a PRG, or they use the Doerner-Shelat technique [34], which requires computation exponential in the number of input bits. When applying the MIC gate to spline approximation, this is not an issue because the inputs can be made short by truncation, leveraging the fact that the input is a fixed point number with absolute value ≤ 1. Nevertheless, in the interval containment functionality, which identifies which type of sigmoid approximation needs to be used, this is no longer the case. This is because we do not have any simple way to reduce the input size. Hence, we would need an FSS gate with a large input domain, which would have extremely high offline computation.

1. https://github.com/google/fss_machine_learning

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 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 smaller bit numbers combined with a functionality that computes the AND over the bits in all bit prefixes of a number. We present a new single online round construction for the latter functionality, where the input is split among two parties, which uses iDPFs. The resulting protocol improves over the online communication of Rathee et al.'s CrypTFlow2 [74] by $\approx 2.6\times$, SynCirc [68] by $\approx 1.3\times$ and Couteau [30] by $\approx 4.6\times$ for 64-bit inputs and appropriate parameters. We also reduce the number of online communication rounds by similar factors, i.e., from 6, 4 and 12 rounds respectively to 3.

Secure comparison is a fundamental building block for higher-level privacy-preserving applications. Couteau [30] presents 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 F). In Appendix F, we give a new 2-party protocol for generating DPF keys for arithmetic-shared outputs using MPC. This is an important extension to [34], who only handle Boolean-shared outputs. Our construction only requires a single additional oblivious transfer in the offline phase, independent of the size of the output shares.

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 $L = 2^\ell$ and $s \leq w < \ell$. Similar to previous works [24], 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 at most 2^{w-s} have a corresponding fixed-point representation in the ring. Specifically, non-negative real numbers are mapped to $[0, 2^w)$ whereas negative real numbers are mapped to $(L - 2^w, L)$ 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* [45], i.e., the parties are assumed to follow the prescribed protocol. We denote the two parties by P_0 and P_1. The security in such a model is captured by the standard real/ideal paradigm whereby the view of an adversary in the real-world, that corrupts one of the parties (either P_0 or P_1), can be efficiently simulated in the ideal world where parties interact with an ideal functionality. A detailed description of the security model can be found in Appendix K.

The protocol Π may be divided into an offline preprocessing phase Π_{offline} (independent of parties' inputs) and an online phase Π_{online} that depends on parties' inputs. In practical settings, Π_{offline} may be performed by a trusted third party, or by the parties executing an MPC protocol. However, Π_{online} is always performed by parties using MPC. Due to space constraints, we defer the formal proofs of security of our protocols to Appendix K. The costs of a protocol Π are captured by three standard metrics: computation, communication and rounds of interaction. Unless otherwise specified, we measure computation cost per party, communication cost across both parties, and rounds of interaction in a simultaneous synchronous message model (where both parties proceed synchronously and are allowed to send each other a message in the same round). When the protocol is divided into an offline and online phase, these cost metrics are independently calculated for each phase.

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 [64]: Suppose x_0 and x_1 are the shares of $\widehat{x}$ held by party P_0 and P_1 respectively. Then, to perform the truncate operation, both parties can 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 [64] 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 is tolerable as the FXP representation itself admits an error in the least significant bit of the fractional part compared to the actual real value. Apart from this, there is also a failure probability of $\frac{2^{w+1}}{2^{\ell}}$ associated with the protocol, which can be made arbitrary small by choosing w and ℓ appropriately. Appendix J explains how this failure probability affects our protocols.

2.1. Logistic Regression

Logistic regression is a probabilistic classifier and a supervised learning algorithm [56]. 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 is one such technique that computes the optimal weights $\mathbf{w}$ by minimizing the average loss $\mathcal{L}_{\mathsf{CE}}$ 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 $\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{B} \sum_{i \in [B]} \mathbf{g}_i$. This procedure is repeated until $\mathcal{L}_{\mathsf{CE}}$ is minimized (or sufficiently small).

In the context of secure computation protocols we will run a fixed number of epochs to avoid leakage about the private samples based on the time for convergence. We will use a minibatch and a fullbatch technique. The minibatch technique, in each step of gradient descent, uses a fixed subset of B samples. This is unlike the fullbatch technique that uses all n samples. In minibatch, each step of gradient descent is called an iteration. The set of iterations that process a full batch is called an epoch. In other words, the gradient descent is run over multiple epochs each consisting of a fixed number of iterations.

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

Let parties hold additive shares of $x, y \in \mathcal{R}$. Then they can use 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 online phase. This incurs 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 [57] 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 truncation protocol at the end, where s least significant bits are truncated from the result in order to adjust the fractional scale. We use the non-interactive SecureML truncation [64] described in Section 2.

2.3. Two-Party Computation Functionalities

Multiplexer. Following [73], 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 (fresh) 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 simultaneous OTs [73]. 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) 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 with the sharing of 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) [19]. 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 $f_b(x) \in \mathbb{G}^{out}$.

Distributed point function (DPF) [19] is an FSS for a point function that evaluates non-zero on a single point. A DPF allows a compressed 2-party secret-sharing of a point function. Incremental distributed point functions (iDPFs) [13] are a generalization of DPFs which 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 $\ell \in [n]$:

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

$$f_{\alpha,\bar{\beta}}(x_1, \ldots, x_\ell) = \begin{cases} \beta_\ell & \text{if } (x_1, \ldots, x_\ell) = (\alpha_1, \ldots, \alpha_\ell) \\ 0 & \text{otherwise} \end{cases}$$

Distributed Comparison Function (DCF) is an FSS scheme for a function $f^{<}_{\alpha,\beta}$, which outputs β if $x < \alpha$ and 0 otherwise.

Secure Computation via FSS. Boyle et. al. [16], [21] 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. Functions can be computed on secret-shared inputs and outputs using FSS gates together with a single round of communication. Functions supported include $\mathcal{F}_{\mathsf{EQ}}, \mathcal{F}_{\mathsf{CMP}}$ and $\mathcal{F}_{\mathsf{MIC}}$ for securely computing equality, comparison and multiple-interval-containment respectively. The equality and comparison functionalities are self-explanatory. For $\mathcal{F}_{\mathsf{MIC}}$, parties know a series of k public intervals, and the output of $\mathcal{F}_{\mathsf{MIC}}(x)$ is a k-element secret-shared vector which is 1 in a position when x lies in the interval, and 0 otherwise. We note that we can accommodate either Boolean-shared or arithmetic-shared outputs. Due to lack of space, we present formal definitions and details for these functionalities in Appendix A and Appendix B.

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 of interaction. The setting is as follows: There are two parties (servers), each holding a share of private dataset X, consisting of n examples (rows) and k features (columns), and a sharing of the vector of n labels $\mathbf{y}$. They wish to train a logistic regression model $\mathbf{w}$ and end up with a sharing of $\mathbf{w}$. The two parties are non-colluding and semi-honest. In this setting, the security of a protocol Π is formally captured via the real-ideal paradigm as mentioned in Section 2 and detailed in Appendix K.

Like previous works [64], [78], we leverage arithmetic secret-sharing (see Section 2) and train the model $\mathbf{w}$ 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 efficient secure exponentiation.

Algorithm 1: Logistic Regression Protocol

Public inputs: Number of epochs T, dataset dimensions n, k, batch size B, learning rate α, regularization parameter λ, fixed-point parameters $(\mathcal{R}, w, s, \mathsf{Fix})$.
Private inputs: Secret-shared dataset $[\![X]\!] \in \mathcal{R}^{n \times k}$ and labels $[\![\mathbf{y}]\!] \in \mathcal{R}^n$ where X and $\mathbf{y}$ are in fixed-point representation.
Private outputs: Secret-shared trained model after T epochs $[\![\mathbf{w_T}]\!]$ where $\mathbf{w_T}$ is in fixed-point representation.

Let $[\![\mathbf{w_0}]\!]$ be the initial secret-shared model with arbitrary weights.
for $t = 1$ *to* T **:**
  for $b = 1$ *to* $\lfloor n/B \rfloor$ **:**
    $i \leftarrow (b-1) \cdot B + 1$
    $j \leftarrow \min(n, b \cdot B)$
    $[\![X_B]\!] \leftarrow [\![X_{i \ldots j}]\!]$
    $[\![\mathbf{u}]\!] \leftarrow \mathcal{F}_{\mathsf{corrMatMult}}([\![X_B]\!], [\![\mathbf{w_{t-1}}]\!])$
    $[\![\mathbf{s}]\!] \leftarrow \mathcal{F}_{\mathsf{Sigmoid}}([\![\mathbf{u}]\!])$
    $[\![\mathbf{d}]\!] \leftarrow [\![\mathbf{s}]\!] - [\![\mathbf{y}_{i \ldots j}]\!]$
    $[\![\mathbf{g}]\!] \leftarrow \mathcal{F}_{\mathsf{corrMatMult}}([\![{X_B}^\top]\!], [\![\mathbf{d}]\!])$
    $[\![\mathbf{w_t}]\!] \leftarrow [\![\mathbf{w_{t-1}}]\!] - (\alpha/B) \cdot ([\![\mathbf{g}]\!] + \lambda \cdot [\![\mathbf{w_{t-1}}]\!])$
return $[\![\mathbf{w_T}]\!]$.

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:

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

Division is sometimes approximated via Goldschmidt's [46] method, which is expensive. Alternatively, exponentiation can be approximated by decomposing the exponent into bits [31], which is costly, or via low-degree polynomials [4], 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 Appendix L.1 and Appendix L.4 for detail on how we implement the more complex components of our functionality.

4.1. Sigmoid Approximation

The sigmoid function is 'symmetric' around the y-axis. More specifically, $S(x) + S(-x) = 1$ for all $x \in \mathbb{R}$. 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 show 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 Appendix L.4 based on the protocol of [62]. Our protocol computes *all* powers of r (irrespective of the degree) in only 2 communication rounds.

However, this approximation works well only when $r \ll 1$. 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 [57]. We note that the exponentiation protocol from [57] 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 $s/\log_2(e)$, i.e. whenever $x \geq s/\log_2(e)$, we set the sigmoid output to 1. This bound can be justified by observing that for any $x \geq s/\log_2(e)$, $e^{-x} < 2^{-s}$. 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 Appendix L.1 [2]. For security, 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 secret-shared between the parties.

Algorithm 2: Approximate Sigmoid

Parameters:
Let m be the number of lines defining a spline.
Let s 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.

$\mathsf{Sigmoid}(x)$:

if $x < 0$ **then**
  $S \leftarrow 1 - \mathsf{Sigmoid}(-x)$.
else
  if $x < 1$ **then**
    $S \leftarrow \mathsf{Spline}(m, [0, 1))$
  else
    if $x \log_2(e) \geq s$ **then**
      $S \leftarrow 1$
    else
      $r \leftarrow e^{-x}$
      $S = \frac{1}{1+r} \leftarrow 1 - r + r^2 - \ldots (-1)^d r^d$
return S

5. Secure Sigmoid with Trusted Setup

In this section, we describe the details of our approach for *securely* computing the sigmoid approximation described in Algorithm 2 with a focus on minimizing the online communication cost. The setting is as follows: there are two parties (servers), each holding a share of private input x, and they wish to end up with a share of $u := \mathsf{Sigmoid}(x)$ where Sigmoid is computed as defined in Algorithm 2. The two parties are non-colluding and semi-honest. In this setting, the security of a protocol Π is formally captured via the real-ideal paradigm as mentioned in Section 2 and detailed in Appendix K.

In designing our protocol, we assume that the offline phase of our protocol is part of a trusted setup phase [3]. 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 9 in Appendix G), 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 $[0, 1), [1, \frac{s}{\log_2(e)}), [\frac{s}{\log_2(e)}, \infty)$, 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 [8]. In Appendix L we discuss how to build the tailored sigmoid implementations for each interval.

6. Secure Sigmoid with Distributed 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.

2. One could imagine using only splines to approximate sigmoid on the entire $(-\infty, \infty)$ interval as has been done in prior works [64] However, this would require a large number of spline intervals and potentially higher degree splines for the approximation to work well, thus increasing the cost of protocol. Therefore, in our heterogeneous approach, we use the spline based approximation only for the smaller interval $[-1, 1]$. For the remaining interval, we use the new exponentiation combined with Taylor series based approximation.

3. Regardless of how the trusted setup phase is actually realized, it is assumed that this setup phase is performed by an entity which is not colluding with any party.

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 [4] blow up the communication cost of the offline phase by at least linear in the size of PRG circuit. An alternative approach by Doerner et. al. [34] ingeniously avoids executing the PRG circuit inside 2PC at the cost of exponential (in the bit length of FSS inputs) computational cost making it feasible only for small input sizes e.g. less than 20 bits.

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 $[0,1), [1, \frac{s}{\log_2(e)}), [\frac{s}{\log_2(e)}, \infty)$, or the negative equivalents of these ranges. Besides this, we also use $\mathcal{F}_{\mathsf{MIC}}$ as a subprotocol 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 Appendix L.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 [34] to efficiently generate FSS keys in the distributed setting when $s \leq 20$.

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 and P_1 have a private input x and y respectively. The output of a secure comparison 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). Formally, $\mathcal{F}^{\ell}_{\mathsf{CMP}}(x, y) \to (b_0, b_1)$, where $x, y \in \{0,1\}^{\ell}$ and b_0, b_1 is a Boolean sharing of bit $b := \mathbf{1}\{x < y\}$.[5]

A common approach to computing secure comparison is divide-and-conquer [30], [42], [73], 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 [53] optimizes the offline communication of [73] using VOLE-style OT extension which is orthogonal to our focus on online efficiency.

Another line of work based on function secret sharing (FSS) [16] 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 [34] 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 at this moment no better concretely communication-efficient technique in the literature for conducting the FSS offline phase.

Another line of work originating from ABY [32] and its successors ABY 2.0 [67] and SynCirc [68] solve secure comparison 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 $O(\log \ell)$ whereas our protocol is constant round (at most 3). In Table 4, we compare the cost of our protocol with SynCirc (the most optimized work in this direction). Rabbit [63] solves the secure comparison problem in the dishonest majority multiparty setting using Boolean adder circuit and other techniques, has higher communication cost and $O(\log \ell)$ rounds.

In our approach, we start by looking at the decomposition of comparison problem for ℓ-bit strings in terms of comparison and equality operations on smaller sub-strings as described in Garay et al. [42]. Formally, for $x = x_1||x_2$ and $y = y_1||y_2$, where $x, y \in \{0,1\}^{\ell}$ are ℓ-bit strings, the following relationship holds:

$$x < y = (x_1 < y_1) \oplus \Big((x_1 = y_1) \wedge (x_2 < y_2)\Big) \quad (1)$$

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

$$\begin{aligned} x < y = {} & (x_1 < y_1) \\ & \oplus \Big((x_1 = y_1) \wedge (x_2 < y_2)\Big) \oplus \ldots \\ & \oplus \Big((x_1 = y_1) \wedge \ldots \wedge (x_{q-1} = y_{q-1}) \wedge (x_q < y_q)\Big) \end{aligned} \quad (2)$$

4. 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.

5. An alternative formulation of secure comparison lets the two parties hold secret shares of x and y as input and learn a secret shared bit b representing the comparison output. However, as mentioned in [30], this problem can be non-interactively and black-box reduced to $\mathcal{F}_{\mathsf{CMP}}$ without any overhead. In our actual implementation within sigmoid use-case, we will use this alternative formulation.

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

$$\begin{aligned} x < y = \ell_1 \ \oplus \ (e_1 \wedge \ell_2) \ \oplus \ \ldots \ \oplus \ (e_1 \wedge \ldots \wedge e_{q-1} \wedge \ell_q) \\ = \langle 1 \quad e_1 \quad e_1 \wedge e_2 \quad \ldots \quad e_1 \wedge e_2 \wedge e_{q-1} \rangle \cdot \\ \langle \ell_1 \quad \ldots \quad \ell_q \rangle \end{aligned} \tag{3}$$

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 $\ell_1, \ldots, \ell_q$.
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 [16] 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 $2^{q-1}-1$. 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 [16], [21], the authors use this technique to create an equality check gate and comparison gate via FSS schemes such as DPF and DCF respectively. 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 $\alpha = 2^{q-1} - 1 + r_{in}$ does not 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 2 in Appendix C. For binary outputs, the protocol requires $4\ell - 2m + 6q - 6$ bits of online communication and 3 rounds, where $m \in [1, \ell]$ is a parameter for the sub-string/block length [6].

7. Experimental Evaluation

Implementation Details. We implemented our constructions in C++ with the Bazel build system [7]. Our implementation is end-to-end, including server and client code, as well as network communication implemented using gRPC's asynchronous APIs. In our codebase we include an efficient implementation of distributed point and comparison functions, which makes use of hardware-backed AES-NI instructions and optimizes CPU pipelining by batching multiple DPF/DCF evaluations together. It also features an implementation of the interactive DPF key generation protocol described in Appendix F. Finally, our implementation has extensive tests, written in the GoogleTest library [49], for all building blocks alongside end-to-end tests.

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 3.1 GHz clock rate (except for experiments run on the Criteo datasets, where we used compute-optimized c2-standard-60 instances with 240 GB RAM and same CPU). Our implementation runs on a single thread and uses 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 0.12ms 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

6. The online communication can be reduced to $2\ell - 2m + 6q - 6$ by sharing the masks for FSS comparison and equality gates used in the first round.

44.70ms and the bandwidth $\approx 65MB/s$. All runtimes and communication are end-to-end totals and include both the client and server costs. The client and the server execute their computation simultaneously in each round. They proceed to next round once both parties completed computation and transferred messages to one another.

Cloud costs. We include the monetary cost of running our protocols on the Google Cloud Platform (GCP), using the prices listed on 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. This reflects batch computation with parties situated in different cloud providers, as has been used in other works [54].

7.1. Sigmoid Experiments

Approximating the sigmoid function is the most challenging and costly component of gradient descent. 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[7], for $10^2, 10^3$, and 10^4 sigmoid inputs and the following parameters: 20 fractional bits, 31-bit width (integer plus fractional), 63-bit ring size, 10 spline intervals in $[0, 1)$, and Taylor series of degree 10. The sigmoids are executed in a single batch.

TABLE 1: Comparison of the online cost of our sigmoids with trusted ($v1$) and distributed ($v2$) offline setup to SIRNN and MP-SPDZ's accurate sigmoid implementations in the latest MP-SPDZ version 0.3.7.

Technique	Time for # Instances (sec)			Comm. per	# Rounds	USD Cost
	10^2	10^3	10^4	Instance (KB)		per 10^3 runs
				LAN		
Sigmoid v1	0.08	1.00	10.80	**0.50**	**4**	**0.004c**
Sigmoid v2	0.13	1.51	16.46	**1.18**	**6**	**0.009c**
MP-SPDZ	0.05	0.29	-	15.32	124	0.117c
Est. SIRNN	0.01	0.03	0.18	19.22	≈100	0.146c
				WAN		
Sigmoid v1	**0.40**	**1.62**	11.79	**0.50**	**4**	**0.004c**
Sigmoid v2	**0.61**	**2.32**	17.88	**1.18**	**6**	**0.010c**
MP-SPDZ	7.12	9.02	-	15.32	124	0.121c
Est. SIRNN	4.50	4.52	4.67	19.22	≈100	0.149c

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 [72], whose sigmoid protocol strictly improves over other state-of-the-art sigmoid approximations such as MiniONN [61] and DeepSecure [76]. We also compare to the sigmoid approximation presented in MP-SPDZ[8] [58]. We retrieved the SIRNN values directly from [72] and extrapolated the cost for our network settings[9]. For MP-SPDZ, we ran their sigmoid implementation in the trusted dealer mode.

We observe a gain of $\approx 38/16\times$ (v1/v2) in communication efficiency over SIRNN and $\approx 30/13\times$ over MP-SPDZ. We also reduce the rounds for sigmoid ($\approx 25/16\times$ over SIRNN and $\approx 31/21\times$ over MP-SPDZ). Our v1 sigmoid requires 4 rounds and our v2 sigmoid requires 6 rounds; we estimate that SIRNN uses ≈ 100 communication rounds (although round complexity is not discussed in SIRNN) and 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 $\approx 36.5\times$ over SIRNN and by $\approx 29.3\times$ over MP-SPDZ on LAN. On a higher latency WAN, we decrease costs by $\approx 37.3\times$ over SIRNN and by $\approx 30.3\times$ over MP-SPDZ. SecFloat [70], a concurrent work, does not explicitly provide a sigmoid protocol but our estimate is that it will require 271 rounds and 47.5 KB of communication (as it uses 187 rounds and 37.23 KB per exponentiation and 84 rounds and 10.27 KB per division) whereas we use only 4/6 rounds 0.5KB/1.18KB of communication for v1/v2, respectively.

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. We note that the increased running time of our protocol, as indicated in Table 1, is due to the local computations involving FSS which consists of AES calls. This gap can be reduced by possibly using multi-threading across batches (which prior works appear to already do). In this work, our focus was not solely on optimizing computation but primarily on reducing communication and the round complexity.

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 key size $(\ell \cdot \lambda + \ell^2 + 3\ell + \lambda) + 2 \cdot n_I \cdot \ell$ bits where n_I is the number of MIC intervals [16]. For $\ell = 63$ and $n_I = 6$, we get a key size of 1.6 KB.
- Exponentiation: Requires a correlation consisting of 2 field elements of at most ℓ bits [57]. For $\ell = 63$, the cost is 0.015 KB.
- Polynomial: Requires a correlation consiting of (shares of) d incremental powers of a random ring element. For $\ell = 63$, we get a cost of 0.077 KB.

7. We were unable to compile 10^4 sigmoid executions into the bytecode used by the MP-SPDZ virtual machine as our device ran out of memory.

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

9. Note that SIRNN does not have offline phase. The entire protocol cost is online. We extrapolated the SIRNN costs in the following way: SIRNN takes 0.08sec on 0.8ms RTT network, so the total rounds will be ≈ 100. In our setting, the LAN and WAN latencies are 0.12ms and 44.7ms, respectively. Therefore, the estimated LAN and WAN SIRNN runtime would be 0.01s (0.12ms * 100) and 4.5s (44.7ms * 100) for 100 instances. Furthermore, SIRNN has a communication cost of 4.88KB assuming 16 bit ring. Since our experiments were performed over 63 bit ring, we scaled the SIRNN cost by $4\times$ for fair comparison.

- Spline: Requires a MIC gate key for 10 intervals and 1 Beaver triple. For $\ell = 63$, 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 $\ell = 63$, we get a cost of 0.046 KB.

In the v1 sigmoid, 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 and an estimated running time of 4.2 μs (based on the bottleneck computation cost of generating FSS keys for MIC gate and Spline)[10]. In comparison, MP-SPDZ requires a storage cost of 80 KB per party and running time of 949 μs when executed in trusted dealer model.

For the v2 setting, the offline cost is measured by the total communication cost and running time needed to generate the required preprocessing material in MPC. In the following, we assume that the amortized cost of generating a Beaver triple for $\ell = 63$ bit ring is 0.4375 KB [75] and a bit Beaver triple is 0.0175 KB [73]. The v2 sigmoid protocol consists of the following components:

- Mux : Requires 2 calls to Ideal OT in online phase. For this, it suffices to have 2 ROT correlations generated in the offline phase, requiring $2\lambda + 4\ell$ bits of communication. For $\ell = 63$, the communication is 0.062 KB.
- Exponentiation: The required correlation can be generated using 2 Beaver multiplications for 0.936 KB.
- Polynomial: The required correlation for $d = 10$ degree polynomial can be generated using d fixed point multiplication. Each fixed point multiplication would require a Beaver multiplication followed by truncation. We estimated the cost of each truncation from [73] for $l = 63$ and $s = 20$ to be 1.5 KB. Given each Beaver triple requires around 0.5 KB to generate and consume, the estimated cost of each fixed point multiplication is approximately 2 KB. For 10 fixed point multiplications, we get a cost of ≈ 20 KB.
- Spline: Requires an MIC gate key on domain size $s = 20$ bits for 10 intervals with output size of $\ell = 63$ bits, and 1 Beaver triple. Using the protocols described in Appendix B and Appendix F for MIC key generation, the offline cost is 5.81 KB.
- MIC based on $\prod_{\mathsf{CMP}}$: For 6 intervals, we need 7 $\prod_{\mathsf{CMP}}$ invocations and 6 $\mathcal{F}_{\mathsf{AND}}$ invocations (performed with bit Beaver triples). The estimated theoretical offline cost for our $\prod_{\mathsf{CMP}}$ is 21.74 KB based on Table 4 in Appendix C. Adding up the costs, we will require ≈ 152 KB to generate preprocessing needed for MIC based on $\prod_{\mathsf{CMP}}$.

In the v2 sigmoid protocol, we invoke 1 MIC based on $\prod_{\mathsf{CMP}}$, 2 exponentiations, 2 polynomial calls, 2 spline calls and 6 MUX calls. This requires an offline communication cost of 205 KB.

We now turn towards the *running time* of the offline phase which includes the cost of secret-sharing based primitives and the cost of FSS-based primitives. The secret-shared based primitives rely on OT which can be cheaply generated using OT extension. From libOT [69] benchmarks, we observed a running time of 75 ns for each OT after excluding the cost of base OTs (which takes less than a second). For FSS based primitives, the offline computation cost is dominated by FSS key generation which involves approximately 2^n calls to AES for n-bit FSS inputs using Doerner Shelat technique. Based on the estimate of 360 million AES calls per second (as reported in [16]), we decided what value of n would be reasonable/feasible. In the v2 sigmoid protocol, we invoke FSS when computing the Spline MIC gate on $s = 20$-bit inputs. We also invoke FSS gates as part of $\prod_{\mathsf{CMP}}$ protocol. For our parameter setting, each invocation of $\prod_{\mathsf{CMP}}$ requires 4 DCF and 4 DPF keys on 16 bit inputs and also an iDPF key on 4 bit input. In total, the theoretical estimated running time comes out to be approximately 16 ms for generating FSS correlations needed for a single sigmoid. Note that our $v1$ sigmoid protocol requires FSS keys on 64 bit domains whose offline phase, if executed securely in two party setting, would require running time of more than 10^{10} seconds using Doerner Shelat technique which is infeasible. This is the reason why we proposed a new protocol for $v2$ setting which can support feasible offline costs.

We also implemented a baseline version (without any optimizations) of the arithmetic extension of Doerner-Shelat's protocol [34], explained in Appendix F, and benchmarked it for 64-bit output groups. Assuming OT correlations have been pre-generated, we get the following costs. In the LAN setting, we observed a running time of 0.29 seconds and 4.17 KB communication for 16-bit input domain. For 20-bit input domain, we get a running time of 3.44 seconds and 5.19 KB communication. In the WAN setting, the communication cost remains same but the running time becomes 6.12 seconds and 10.79 seconds for 16 bit and 20 bit inputs respectively.

7.2. Logistic Regression Experiments

We evaluated our logistic regression experiments on five datasets. We do basic preprocessing on the datasets with the help of Scikit-learn's ML 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 present the training parameters used in our experiments in Table 2.

TABLE 2: Datasets and their corresponding training parameters used in our experiments. We fixed the number of epochs to 6 and ran grid search to determine parameters such as the prediction threshold and the learning rate.

	Titanic	Arcene	Gisette	Criteo Uplift 1	Criteo Uplift 2
Training Size	500	70	4200	70000	70000
Testing Size	214	30	1800	30000	30000
Total Size	714	100	6000	100000	100000
# Features	6	10000	5000	15	15
Learning Rate α	1	0.1	1	1	1
Regularization λ	0.0001	0.0001	0.1	0.0001	0.0001
Prediction Threshold	0.43	0.18	0.64	0.67	0.86
# Epochs	6	6	6	6	6

We used the following datasets: *Titanic [38]* predicts if a passenger would survive the Titanic shipwreck, *Arcene [51]* predicts if a patient has cancer, *Gisette [51]* distinguishes the digits 4 and 9, *Criteo Uplift [33]* predicts whether a user targeted by advertising purchases a product (i.e. converts). We used two versions of the Criteo dataset. In *Criteo Uplift* 1, we used a random subset of the original dataset. This dataset was highly

10. Here we assume an estimate of 360 million AES calls per second on a single-core 3.6 GHz machine (10 machine cycles per AES) [16].

imbalanced. From 10^5 data points, there were only 470 conversions. In *Criteo Uplift* 2, we sampled the examples such that 10% of the labels were positive. I.e., there were 10^4 positive labels.

Accuracy Evaluation. We compare accuracy of our 2PC protocols against a plaintext Python floating-point implementation in Table 6. We use parameters from Table 2 and train for 6 epochs, which is enough for the plaintext algorithm to converge. Our 2PC protocols are close to plaintext logistic regression in all cases.

Performance Evaluation. We present our end-to-end runtime and communication costs for the online phase in Table 7. All versions use the parameters from Table 2, and run for 6 epochs. 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. These works include the piecewise approximations of SecureML (3 pieces) [64] and MP-SPDZ (5 pieces) [58]. As a result of these coarse approximations, SecureML and MP-SPDZ do not closely match plaintext logistic regression: Running logistic regression in plaintext with SecureML's and MP-SPDZ's sigmoid approximations on a subset of the Criteo uplift dataset with $100,000$ examples and 470 positive examples (trained on 6 epochs with minibatches of 100 examples) yields 0 and ≈ 0.42 F1 score, respectively, while Python plaintext (and training in plaintext with our approximation) yields ≈ 0.48 (0.47 when run in MPC). ABY2.0 [67] improves on the efficiency of SecureML's logistic regression by reducing the online runtime, but uses the same coarse sigmoid approximation.

Note that even if we settled for the sigmoid approximation used in SecureML and ABY2.0 (and similarly MP-SPDZ), our work would still 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 4) would reduce their training costs.

Turning to accurate logistic regression approximations, MP-SPDZ [58] has one sigmoid approximation that results in comparable accuracy to our protocol. We ran the offline and online phase of MP-SPDZ's logistic regression training for one fullbatch epoch on a dataset of 1000 examples and 10 features and compared to our logistic regression implementation. Our offline phase costs use our earlier sigmoid offline estimates and [57]'s implementation of correlated Beaver triples. We compare MP-SPDZ's Semi-2k (semi-honest 2-party) protocol with our v2 protocol and MP-SPDZ's Dealer (semi-honest 2-party with a dealer) protocol with our v1 protocol. We present the results in Table 3. SIRNN [72] implements sigmoid, but not logistic regression, so we only compare with their sigmoid in Section 7.1.

TABLE 3: Comparison of 1 epoch of our secure logistic regression with v1 and v2 sigmoid against MP-SPDZ v0.3.7 with their accurate sigmoid version run in the Dealer (c.f. v1) and Semi2k (c.f. v2) setting. The offline phase costs of our protocol are estimates (see text).

	2PC v1	MP-SPDZ Dealer	2PC v2	MP-SPDZ Semi2k
	Dataset (1000×10)			
Offline Comm (MB)	2.96	218.59	226.66	4159.12
Online Comm (MB)	0.50	65.00	1.12	65.00
Offline LAN (sec)	0.16	0.41	19.75	16.00
Online LAN (sec)	2.16	0.42	4.73	0.38
Offline WAN (sec)	0.24	3.25	27.21	740.67
Online WAN (sec)	2.85	13.55	6.21	12.70

7.3. Secure Comparison Experiments

Analytical Comparison. In Table 4, we benchmark the offline and online analytical communication costs of our new comparison protocol $\prod_{\mathsf{CMP}}$ for different values of ℓ (bit length of the inputs). We set $m = 16$ as the parameter in our protocol and estimated the offline cost based on the cost of generating DPF, DCF and iDPF keys for single bit output via the Doerner Shelat technique described in [16], [34]. We compare it against CrypTFlow2 [73] (where we set $m = 4$ as used by the authors[11]), SynCirc [68] and Couteau's protocol [30]. Note that if $\ell \leq m$, then we set $m = \ell$. Our offline costs exclude the cost of base OT. For SynCirc [68], we could not obtain values for $\ell = 4, 8, 128$ bits as the paper doesn't report costs for these cases.

Performance Comparison. We compare our secure comparison protocol $\prod_{\mathsf{CMP}}$ against [16]'s FSS comparison gate which presents the lowest known online communication, 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 364ms to 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). If we use our DCF batching optimization, we reduce the FSS comparison runtime on LAN to 18.72ms (i.e. $\approx 19.4\times$ improvement) and $\prod_{\mathsf{CMP}}$'s runtime to 205.26ms (i.e. $\approx 2.6\times$ improvement). We present our experiments in Table 5.

8. 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) [2], [12], [22], [26], [37], [43], [57]–[59], [64], [64], [67], [71], [72], [76], [79], [81]. 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 computation

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

TABLE 4: Concrete analytical communication and round costs of our comparison protocol vs. prior works as functions of bit length ℓ. The offline costs exclude the costs of base OTs. CryptFlow2 [73] and our approach have an additional parameter m denoting the block size. We fix $m = 4$ for CryptFlow2 (as suggested by the authors) and $m = 16$ for our approach.

	Our Approach (m = 16)		CrypTFlow2 ($m = 4$) [73]		SynCirc [68]		Couteau16 [30]	
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

TABLE 5: Comparison of our new $\prod_{\mathsf{CMP}}$ protocol to the FSS protocol [16] on a batch of 1000 inputs.

	$\prod_{\mathsf{CMP}}$	FSS Comparison [16]
LAN (ms)	532	364
WAN (ms)	671	410
Communication (KB)	29.55	15.63

TABLE 6: Accuracy comparison of our secure 2PC algorithms (executed interactively on 2 machines) with the insecure non-interactive algorithm implementing the standard Section 2.1 gradient descent in Python floating point.

	Insecure Python	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
		Gisette Dataset	
F1 Score	0.96987	0.96540	0.96540
Accuracy	0.97056	0.96611	0.96611
		Criteo Uplift 1 Dataset	
F1 Score	0.47910	0.46336	0.46336
Accuracy	0.993	0.992	0.992
		Criteo Uplift 2 Dataset	
F1 Score	0.86367	0.86155	0.86155
Accuracy	0.971	0.970	0.970

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 [58] and SIRNN [72], 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

TABLE 7: *Online* costs of running our secure 2PC gradient descent (executed interactively on 2 machines) for 6 epochs 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)	2.74	4.54	2.73	4.39
WAN (sec)	5.06	6.77	4.83	6.61
LAN Cost (USD)	0.01c	0.03c	0.01c	0.03c
WAN Cost (USD)	0.01c	0.03c	0.01c	0.03c
		Arcene Dataset (70×10000)		
Comm (MB)	1.24	1.50	1.18	1.45
LAN (sec)	1.71	1.84	1.39	1.53
WAN (sec)	4.60	5.24	4.04	4.79
LAN Cost (USD)	0.01c	0.01c	0.00c	0.01c
WAN Cost (USD)	0.01c	0.01c	0.01c	0.01c
		Gisette Dataset (4200×5000)		
Comm (MB)	13.00	28.88	12.97	28.85
LAN (sec)	66.59	78.64	56.30	69.94
WAN (sec)	84.90	97.67	68.23	81.12
LAN Cost (USD)	0.14c	0.27c	0.13c	0.26c
WAN Cost (USD)	0.15c	0.28c	0.14c	0.27c
		Criteo Uplift 1/2 Datasets (70000×15)		
Comm (MB)	208.09	472.61	208.09	472.61
LAN (min)	9.44	17.64	9.40	17.61
WAN (min)	9.68	17.97	9.69	17.78
LAN Cost (USD)	1.94c	4.28c	1.94c	4.28c
WAN Cost (USD)	1.95c	4.29c	1.95c	4.28c

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) because of the 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 Section 7.1 and Section 7.2.

There are also works such as SecureML [64] and ABY2.0 [67] 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 these works rely mainly on AND gates and secure comparisons. Since we propose an improved secure comparison, replac-

ing the secure comparisons in [64], [67] with those in our work is likely to offer an improvement. This is discussed further with concrete accuracy numbers in Section 7.2. We discuss some other works that fall into this category of "inaccurate sigmoid" in Appendix D.

There are several works which leverage homomorphic encryption in order to compute Logistic Regression, for example [22] and [26]. 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. Froelicher et. al. [41] use multi-party FHE [65] for data analysis but they do not consider secure training of a logistic regression model. Separately from logistic regression and sigmoid computation, there are several related works focused on developing Function-Secret-Sharing or adjacent primitives [16], [21], [67], and using them for machine learning [77]. 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 [16], [30], [32], [42], [67], [68], [73], 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 paper which became public concurrently with our work, LLAMA [50], which builds on SIRNN [72] 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 ensure that their chosen accuracy metric (ULP error) is small (≤ 4). They use the spline gate from [16], 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 d untruncated multiplications, where d is the polynomial degree, 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 (for a given d), 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. Another difference, from a performance standpoint, is that the proposed sigmoid protocol in LLAMA requires large amount of preprocessing material - around 132 KB[12] whereas ours is only 2.95 KB ($44\times$ less) as mentioned in Section 7.1. One of the key factors behind this reduction is our novel sigmoid approximation which departs from "spline-only" based approximation (that LLAMA and other referred papers use). Another major difference is that LLAMA doesn't consider the v2 setting at all in their sigmoid design. If one were to directly port their protocol to run in the v2 setting for $\ell = 63$ bit ring, the estimated offline computation cost to generate the FSS keys using Doerner-Shelat 2PC [34] would require more than 10^{10} seconds, making it practically infeasible.

9. Conclusion

We show that techniques from FSS can be combined with secret-sharing MPC to build a novel secure logistic regression training protocol with better accuracy and performance tradeoffs. Our key contribution is a 2PC-friendly sigmoid approximation. An interesting open problem is to further optimize the offline costs of the sigmoid protocol by developing silent preprocessing inspired techniques [17], [18] targeted specifically towards sigmoid friendly correlations.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

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

[8] 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.

[9] 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.

12. The authors report 33 KB for 16 bit ring. Since we use 63 bit ring in this work, we scale their reported value by $4\times$.

[10] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From input private to universally composable secure multi-party computation primitives. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 184–198. IEEE, 2014.

[11] 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.

[12] 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.

[13] 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.

[14] Christina Boura, Ilaria Chillotti, Nicolas Gama, Dimitar Jetchev, Stanislav Peceny, and Alexander Petric. High-precision privacy-preserving real-valued function evaluation. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*, pages 183–202. Springer, 2018.

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

[16] 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.

[17] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 291–308, 2019.

[18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 489–518. Springer, 2019.

[19] 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.

[20] 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.

[21] 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.

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

[23] Sergiu Carpov, Kevin Deforth, Nicolas Gama, Mariya Georgieva, Dimitar Jetchev, Jonathan Katz, Iraklis Leontiadis, M Mohammadi, Abson Sae-Tang, and Marius Vuille. Manticore: Efficient framework for scalable secure multiparty computation protocols. *Cryptology ePrint Archive*, 2021.

[24] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *Financial Cryptography and Data Security: 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers 14*, pages 35–50. Springer, 2010.

[25] 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.

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

[27] 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.

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

[29] 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).

[30] 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.

[31] 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.

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

[33] Diemert Eustache, Betlei Artem, Christophe Renaudin, and Amini Massih-Reza. A large scale benchmark for uplift modeling. In *Proceedings of the AdKDD and TargetAd Workshop, KDD, London,United Kingdom, August, 20, 2018*. ACM, 2018.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Haas Charles Eaton John. Titanic: Triumph and tragedy. In *W. W. Norton & Company*, 1994.

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

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

[41] David Froelicher, Juan R Troncoso-Pastoriza, Jean Louis Raisaro, Michel A Cuendet, Joao Sa Sousa, Hyunghoon Cho, Bonnie Berger, Jacques Fellay, and Jean-Pierre Hubaux. Truly privacy-preserving federated analytics for precision medicine with multiparty homomorphic encryption. *Nature communications*, 12(1):5910, 2021.

[42] 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.

[43] 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.

[44] Alexandre Gilotte. Results from the criteo-adkdd-2021 challenge, 2021.

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

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

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

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

[49] Google. Googletest – google testing and mocking framework. https://github.com/google/googletest, 2023.

[50] 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.

[51] Guyon Isabelle, Gunn Steve, Ben-Hur Asa, and Dror Gideon. Result analysis of the nips 2003 feature selection challenge. In *NIPS, 2004*, 2004.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

[60] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography*, pages 277–346, 2017.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies*, 2021(CONF):291–311, 2021.

[66] 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.

[67] 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.

[68] 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.

[69] Lance Roy Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[70] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. Secfloat: Accurate floating-point meets secure 2-party computation. *Cryptology ePrint Archive*, 2022.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] XiaoFeng Wang, Haixu Tang, Shuang Wang, Xiaoqian Jiang, Wenhao Wang, Diyue Bu, Lei Wang, Yicheng Jiang, and Chenghong Wang. idash secure genome analysis competition 2017, 2018.

[81] 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.

[82] 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.

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

A. Definitions: iDPF, DCF, DDCF

Incremental Distributed Point Functions. Introduced by Boneh et al. [13], 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 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 $\ell \in [n]$:

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

$$f_{\alpha,\bar{\beta}}(x_1, \ldots, x_\ell) = \begin{cases} \beta_\ell & \text{if } (x_1, \ldots, x_\ell) = (\alpha_1, \ldots, \alpha_\ell) \\ 0 & \text{otherwise} \end{cases}$$

We sometimes allow an iDPF to be evaluated over the empty prefix. We now present iDPF formally, see Figure 1 for more intuition. We closely follow the definitions of Boneh et al. [13], with one major difference being that we expose the $\mathsf{EvalNext}$ 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 $\lambda, n, (\mathbb{G}_1, \ldots, \mathbb{G}_n)$.
- $\mathsf{EvalNext}^{\mathsf{idpf}}(b, \mathsf{st}_b^{\ell-1}, \mathsf{pp}_\ell, x_\ell)$ is a polynomial time *incremental evaluation* algorithm that given a party id $b \in \{0,1\}$, secret state $\mathsf{st}_b^{\ell-1}$, public parameters pp_ℓ, and input evaluation bit $x_\ell \in \{0,1\}$, outputs an updated state and output share $(\mathsf{st}_b^\ell, y_b^\ell)$.
  Intuitively, $\mathsf{EvalNext}$ represents the evaluation on some partial value $x \in \{0,1\}^{\ell-1}$ and outputs a secret sharing y_b^ℓ of the value on the $x||x_\ell$th node of the binary tree and an updated state st_b^ℓ.
- $\mathsf{EvalPrefix}^{\mathsf{idpf}}(b, k_b, \mathsf{pp}, (x_1, \ldots, x_\ell))$ 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 $(x_1, \ldots, x_\ell) \in \{0,1\}^\ell$, outputs an additive secret sharing of the output value y_b^ℓ.

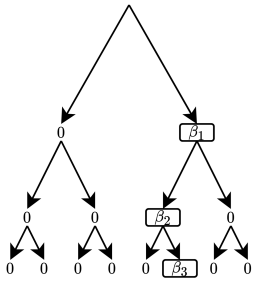

Figure 1: Incremental DPF gives compact secret sharing of values on the nodes of a binary tree with a single non-zero path. In this example, $\alpha = 101$ and the values on the path to the leaf at index α are $\beta_1, \beta_2, \beta_3$. All other nodes are 0. This figure shows the reconstructed secret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

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 $\ell \in [n]$, and input prefix $(x_1, \ldots, x_\ell \in \{0,1\}^\ell)$, the following requirements hold:
  - $\mathsf{EvalNext}$: $Pr[y_0^\ell + y_1^\ell = f_{\alpha,\bar{\beta}}(x_1, \ldots, x_\ell)] = 1$, 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) $\quad (\mathsf{st}_b^j, y_b^j) \leftarrow \mathsf{EvalNext}^{\mathsf{idpf}}(b, \mathsf{st}_b^{j-1}, \mathsf{pp}_j, x_j)$
    4) **return** y_b^ℓ
  - $\mathsf{EvalPrefix}$: $Pr[y_0^\ell + y_1^\ell = f_{\alpha,\bar{\beta}}(x_1, \ldots, x_\ell)] = 1$, 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 $b \in \{0,1\}$:
    $y_b^\ell \leftarrow \mathsf{EvalPrefix}^{\mathsf{idpf}}(b, k_b, \mathsf{pp}, (x_1, \ldots, x_\ell))$
- **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 $x \in \{0,1\}^\ell$ with the ℓth key. This solution would yield key size *quadratic* in the input length n. [13] gives a more direct construction with key size linear in n.

Theorem A.1 (Concrete cost of iDPF [13])**.** *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 $\approx n(\lambda + n)$ [16].

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 E.

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 $f_{\alpha,\beta_1,\beta_2} = \beta_2 + f^{<}_{\alpha,\beta_1-\beta_2}(x)$.

B. Functionalities Based on FSS

We will now describe some of the functionalities which can be realized efficiently with Function Secret Sharing. We note that gates based on FSS operate on masked inputs and produce masked outputs (instead of standard secret-sharing MPC gates which operate 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 [16], 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:

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

Boyle et. al. [21] 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 $\mathsf{r}^{\mathsf{in}}_0, \mathsf{r}^{\mathsf{in}}_1$ their masks. Then, equality holds when $x - \mathsf{r}^{\mathsf{in}}_0 = y - \mathsf{r}^{\mathsf{in}}_1$, or equivalently, $x - y = \mathsf{r}^{\mathsf{in}}_0 - \mathsf{r}^{\mathsf{in}}_1$. In other words, we evaluate a DPF function that evaluates to $\beta = 1$ when $\alpha = \mathsf{r}^{\mathsf{in}}_0 - \mathsf{r}^{\mathsf{in}}_1$, 0 otherwise. We present the full construction in Algorithm 3.

Algorithm 3: FSS Gate for $\mathcal{F}_{\mathsf{EQ}}$

Input: P_0, P_1 hold $x_{\mathsf{mask}} := x + \mathsf{r}^{\mathsf{in}}_0$, where $x_{\mathsf{mask}} \in \mathbb{G}_1$, and $y_{\mathsf{mask}} := y + \mathsf{r}^{\mathsf{in}}_1$, 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.
$\mathsf{Gen}^{\mathsf{eq}}_n(1^\lambda, \mathsf{r}^{\mathsf{in}}_0, \mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{out}})$:
Let $\mathsf{r}^{\mathsf{in}}_0 \in \mathbb{G}_1$ and $\mathsf{r}^{\mathsf{in}}_1 \in \mathbb{G}_2$.
Let $\alpha \leftarrow \mathsf{r}^{\mathsf{in}}_0 - \mathsf{r}^{\mathsf{in}}_1$, $\beta = 1$.
$k'_0, k'_1 \leftarrow \mathsf{Gen}^{\mathsf{DPF}}(1^\lambda, \alpha, \beta)$
Sample random additive shares $\mathsf{r}^{\mathsf{out}}_0, \mathsf{r}^{\mathsf{out}}_1 \leftarrow [\![\mathsf{r}^{\mathsf{out}}]\!]$.
Let $k_b = k'_b || \mathsf{r}^{\mathsf{out}}_b$.
return (k_0, k_1)

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

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. [16] in Algorithm 4. 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 [16]'s $x > y$ and (2) the output group is $\mathbb{U}_2$ instead of $\mathbb{U}_N$.

Algorithm 4: FSS Gate for $\mathcal{F}^n_{\mathsf{CMP}}$

Input: P_0, P_1 hold $x_{\mathsf{mask}} := x + \mathsf{r}^{\mathsf{in}}_0$, where $x_{\mathsf{mask}} \in \mathbb{G}_1$, and $y_{\mathsf{mask}} := y + \mathsf{r}^{\mathsf{in}}_1$, 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.
$\mathsf{Gen}^{\mathsf{cmp}}_n(1^\lambda, \mathsf{r}^{\mathsf{in}}_0, \mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{out}})$:
Let $y = (2^n - (\mathsf{r}^{\mathsf{in}}_0 - \mathsf{r}^{\mathsf{in}}_1)) \in \mathbb{U}_N$ and $\alpha^{(n-1)} = y_{[0,n-1)}$.
$(k^{(n-1)}_0, k^{(n-1)}_1) \leftarrow \mathsf{Gen}^{\mathsf{DDCF}}_{n-1}\big(1^\lambda, \alpha^{(n-1)}, \beta_1, \beta_2, \mathbb{U}_2\big)$, where $\beta_1 = 1 \oplus y_{[n-1]}, \beta_2 = y_{[n-1]} \in \mathbb{U}_2$.
Sample random $\mathsf{r}^{\mathsf{out}}_0, \mathsf{r}^{\mathsf{out}}_1 \leftarrow \mathbb{U}_N$ s.t. $\mathsf{r}^{\mathsf{out}}_0 \oplus \mathsf{r}^{\mathsf{out}}_1 = \mathsf{r}^{\mathsf{out}}$.
For $b \in \{0,1\}$, let $k_b = k^{(n-1)}_b || \mathsf{r}^{\mathsf{out}}_b$.
return (k_0, k_1)

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

Multiple Interval Containment (MIC) Gate. Boyle et. al. [16] 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 $b_i = \mathbf{1}\{p_i \le x \le q_i\}$.

Algorithm 5: 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 $b_{i\mathsf{mask}} = b_i + \mathsf{r}_i^{\mathsf{out}}$, where $b_i := \mathbf{1}\{p_i \le x \le q_i\}$.

// Part I: Offline Phase.
$\mathsf{Gen}^{\mathsf{mic}}_{n,m,\{p_i,q_i\}_i}(1^\lambda, \mathsf{r}^{\mathsf{in}}, \{\mathsf{r}_i^{\mathsf{out}}\}_{i\in[m]})$:
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.
$\mathsf{Eval}^{\mathsf{mic}}_{n,m,,\{p_i,q_i\}_i}(b, k_b, x_{\mathsf{mask}})$:
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 $\{y_{i,b}\}_i$

C. Secure Comparison Protocol

In Figure 2, we describe our formal secure comparison protocol explained in Section 6.1.

D. Related Works

Boura et. al. [14] introduce a novel and exciting sigmoid approximation, which works well in most settings, but is inaccurate unlike our work on the full interval. I.e., their approximation with trigonometric polynomials suffers from Gibbs phenomenon resulting in inaccurate results for some inputs. We note that the follow-up work [23] uses a direct computation of sigmoid for higher precision, and uses iterative techniques to do so. This paper does not discuss communication rounds, and we note that their experiments are run on ultra low-latency 0.3ms network, and hence interaction cost is not reflected.

E. 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, $x \ge \alpha$, 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 $v_n = v_{n-1} = 0$.
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 6 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 E.1 (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. □

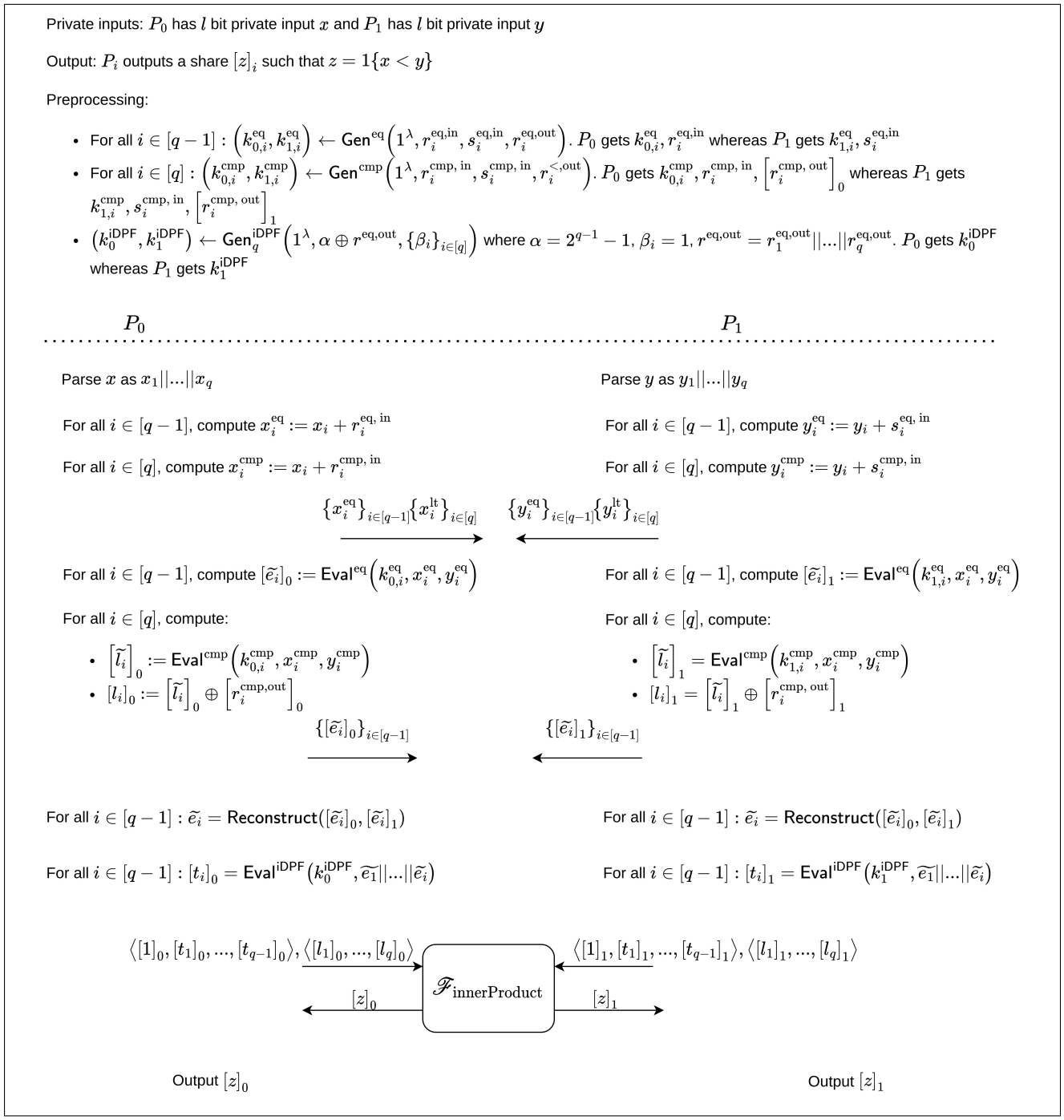

Figure 2: Constant round secure comparison protocol $\prod_{\mathsf{CMP}}$ for ℓ-bit inputs.

Comparison with original DCF construction. Boyle et al. [16] presented a direct construction of DCF by carefully modifying and making non black-box changes to a prior DPF construction [20]. 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. [16] 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.

F. 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 E, 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

Algorithm 6: DCF to iDPF Reduction

$\mathsf{Gen}_n^{\mathsf{DCF}}(1^\lambda, \alpha, \beta)$:

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

$\mathsf{Eval}_n^{\mathsf{DCF}}\big(b, (k_b, [\![\beta_1]\!]^b), \mathsf{pp}, x\big)$:

1 Let $x = x_1, \ldots, x_n \in \{0,1\}^n$ 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 $\quad (\mathsf{st}_i, y_i) \leftarrow \mathsf{EvalNext}_{n-1}^{\mathsf{iDPF}}(b, \mathsf{st}_{i-1}, k_b, x_1 \ldots x_{i-1})$
5 $\quad v_i \leftarrow v_{i-1} + (1 - x_i) \cdot y_i$
6 **return** v_{n-1}

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. [34] presents 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 [16] 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 [16]. 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 7. 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 $\beta_1, \ldots \beta_n$ in Algorithm 6 given $\alpha_1, \ldots, \alpha_n$ 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), $[\![W_{CW}^0]\!] = [\![W_{CW}^1]\!]$ in Step 11, and so the last $\mathcal{F}_{\mathsf{MUX2}}$ call can be saved in that case, making the entire second MPC linear.

G. 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. [1] for general SGD ML training and the instantiation of Jayaraman et. al. [55] for the setting of logistic regression presented in Algorithm 8. Jayaraman et. al. [55] 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 9 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 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

Algorithm 7: 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 P_b performs the following:

Sample $s_b^\emptyset \in \{0,1\}^\lambda$, set $t_b^\emptyset = b$.

for $\ell = 1$ *to* n **:**

For all $w \in \{0,1\}^{\ell-1}$, compute $s_b^{w,L}||t_b^{w,L}||s_b^{w,R}||t_b^{w,R} = G(s_b^w)$.

Compute $s_b^L||t_b^L||s_b^R||t_b^R = \bigoplus_{w\in\{0,1\}^{l-1}} s_b^{w,L}||t_b^{w,L}||s_b^{w,R}||t_b^{w,R}$.

Secure Computation:

- Inputs: Boolean sharing of α_ℓ, arithmetic sharing of $\{s_b^L, s_b^R, t_b^L, t_b^R\}_{b\in\{0,1\}}$.
- Compute :

$$\begin{aligned}
[\![s^R]\!] &\leftarrow [\![s_0^R]\!] \oplus [\![s_1^R]\!] \\
[\![s^L]\!] &\leftarrow [\![s_0^L]\!] \oplus [\![s_1^L]\!] \\
[\![s_{CW}]\!] &\leftarrow \mathcal{F}_{\mathsf{MUX2}}\Big([\![s^R]\!], [\![s^L]\!], [\![\alpha_\ell]\!]\Big) \\
[\![t_{CW}^L]\!] &\leftarrow [\![t_0^L]\!] \oplus [\![t_1^L]\!] \oplus [\![\alpha_\ell]\!] \oplus [\![1]\!] \\
[\![t_{CW}^R]\!] &\leftarrow [\![t_0^R]\!] \oplus [\![t_1^R]\!] \oplus [\![\alpha_\ell]\!]
\end{aligned}$$

- Output $s_{CW}, t_{CW}^L, t_{CW}^R$ to both

For all $w \in \{0,1\}^{\ell-1}$, set $\tilde{s}_b^{w||0}||\tilde{s}_b^{w||1} \leftarrow (s_b^{w,L}||s_b^{w,R}) \oplus t_b^w \cdot (s^{CW}||s^{CW})$

For all $w \in \{0,1\}^{\ell-1}$, set $t_b^{w||0}||t_b^{w||1} \leftarrow (t_b^{w,L}||t_b^{w,R}) \oplus t_b^w \cdot (t_{CW}^L||t_{CW}^R)$

For all $w \in \{0,1\}^\ell$, set $s_b^w||W_b^w \leftarrow \mathsf{Convert}(\tilde{s}_b^w)$

Compute $W_b^\ell \leftarrow \sum_{w\in\{0,1\}^l} W_b^w$.

Compute $T_b^\ell \leftarrow b + (-1)^b \cdot \sum_{w\in\{0,1\}^\ell} t_b^w$.

Let τ_b^0 and τ_b^1 denote the two least significant bits of T_b.

Secure Computation:

- Inputs: Arithmetic sharing of β_ℓ, private inputs $W_b^\ell, \tau_b^0, \tau_b^1$ for Party P_b.
- Compute:

$$\begin{aligned}
[\![t^*]\!] &\leftarrow 1 \oplus \tau_0^1 \oplus \tau_1^1 \oplus (\tau_0^0 \cdot \tau_1^0) \\
[\![W_{CW}^0]\!] &\leftarrow [\![\beta_l]\!] - W_0^l + W_1^l \\
[\![W_{CW}^1]\!] &\leftarrow -[\![\beta_l]\!] + W_0^l - W_1^l \\
[\![W_{CW}]\!] &\leftarrow \mathcal{F}_{\mathsf{MUX2}}\Big([\![W_{CW}^0]\!], [\![W_{CW}^1]\!], [\![t^*]\!]\Big).
\end{aligned}$$

- Output W_{CW} to both

Set $CW^\ell \leftarrow s_{CW}||t_{CW}^L||t_{CW}^R||W_{CW}$

Output $k_b \leftarrow s_b^\emptyset||CW^1||\ldots||CW^n$

of DP noise were presented by Dwork at al. [35] and Champion et al. [25].

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 9. 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 [27], [83] 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 8: DP-SGD

Public inputs: Number of epochs T, Dataset size n, Batch size B, Lipschitz value $G = 1$, Smoothness value $L = 0.25$, Learning rate $\alpha = 1/L$, DP parameters ϵ and δ

Private inputs: Dataset $\mathbf{X}, \mathbf{y}$ having k features

Let $\mathbf{w_0}$ be the initial model with arbitrary weights

for $t = 1$ *to* T **:**

Compute gradient $\mathbf{g_t} \leftarrow \frac{1}{B}\mathbf{X}_B^T \times (\mathsf{Sigmoid}(\mathbf{X}_B \times \mathbf{w_{t-1}}) - \mathbf{Y_B})$

Perturb gradient $\widetilde{\mathbf{g_t}} \leftarrow \mathbf{g_t} + \mathcal{N}(0, \sigma^2 I_p)$ where $\sigma^2 = \frac{8G^2T\log(1/\delta)}{n^2\epsilon^2}$

Update model $\mathbf{w_t} \leftarrow \mathbf{w_{t-1}} - \alpha \cdot \widetilde{\mathbf{g_t}}$

return $\mathbf{w}_T$

Algorithm 9: DP-SGD Logistic Regression Protocol

Public inputs: Number of epochs T, dataset dimensions n, k, batch size B, Lipschitz value $G = 1$, smoothness value $L = 0.25$, learning rate $\alpha = 1/L$, DP parameters ϵ and δ, regularization parameter λ, fixed-point parameters $(\mathcal{R}, w, s, \mathsf{Fix})$.

Private inputs: Secret-shared dataset $[\![X]\!] \in \mathcal{R}^{n\times k}$ and labels $[\![\mathbf{y}]\!] \in \mathcal{R}^n$ where X and $\mathbf{y}$ are in fixed-point representation. Secret shares $[\![r_t]\!] \in \mathcal{R}^k$ of noise drawn from $\mathcal{N}(0, \sigma^2 I_p)$, for each $t \in [T]$ where r_t is in fixed-point representation.

Private outputs: Secret-shared differentially private trained model after T epochs $[\![\mathbf{w_T}]\!]$ where $\mathbf{w_T}$ is in fixed-point representation.

Let $\mathbf{w_0}$ be the initial model with arbitrary weights.

for $t = 1$ *to* T **:**

for $b = 1$ *to* $\lfloor n/B \rfloor$ **:**

$i \leftarrow (b-1) \cdot B + 1$

$j \leftarrow \min(n, b \cdot B)$

$[\![X_B]\!] \leftarrow [\![X_{i\ldots j}]\!]$

$[\![\mathbf{u}]\!] \leftarrow [\![X_B]\!] \cdot \mathbf{w_{t-1}}$

$[\![\mathbf{s}]\!] \leftarrow \mathcal{F}_{\mathsf{Sigmoid}}(\mathbf{u})$

$[\![\mathbf{d}]\!] \leftarrow [\![\mathbf{s}]\!] - [\![\mathbf{y}_{i\ldots j}]\!]$

$[\![\mathbf{g}]\!] \leftarrow \mathcal{F}_{\mathsf{matMult}}\big([\![X_B{}^\top]\!], [\![\mathbf{d}]\!]\big)$

$[\![\mathbf{w_t}]\!] \leftarrow [\![\mathbf{w_{t-1}}]\!] - (\alpha/B)\cdot([\![\mathbf{g}]\!] + \lambda \cdot [\![\mathbf{w_{t-1}}]\!]) + [\![r_t]\!]$

$\mathbf{w_t} \leftarrow \mathsf{Reconstruct}([\![\mathbf{w_t}]\!])$

return $\mathbf{w_T}$.

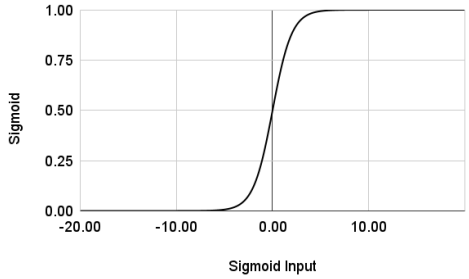

Figure 3: The sigmoid function computed insecurely using the direct formula $\frac{1}{1+e^{-x}}$ over floating-point numbers.

Jayaraman et. al. [55] 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 [55], 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 9.

H. Additional Figures

In this section, we show the sigmoid function computed insecurely (non-interactively outside of secure computation) using the direct formula $\frac{1}{1+e^{-x}}$ in Figure 3. We further demonstrate how our v1 approximation of sigmoid, also computed insecurely, compares to the insecure approximations in SecureML (piecewise approximation of 3 pieces) and MP-SPDZ (also piecewise approximation but of 5 pieces) in Figure 4. In Figure 5, we measure the absolute error of our secure sigmoid implementation (both v1 and v2). The absolute error is defined as the difference between the output of our secure sigmoid implementation (over fixed point inputs with 20 fractional bits) and the baseline implementation of insecure sigmoid using the direct formula $\frac{1}{1+e^{-x}}$ (over floating point inputs). We do this experiment for input values in range $[-20, 20]$ at increments of 0.1.

I. Online Communication Bottleneck 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.

In the online phase, there is a one-time cost of $2nk$ ring elements of communication (associated with the dataset X). Additionally, the per *iteration* (inner loop)

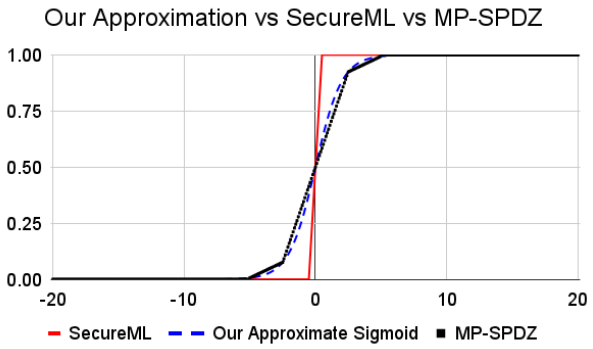

Figure 4: Comparison of our v1 sigmoid approximation to SecureML's and MP-SPDZ's piecewise approximations, executed non-interactively outside of secure computation.

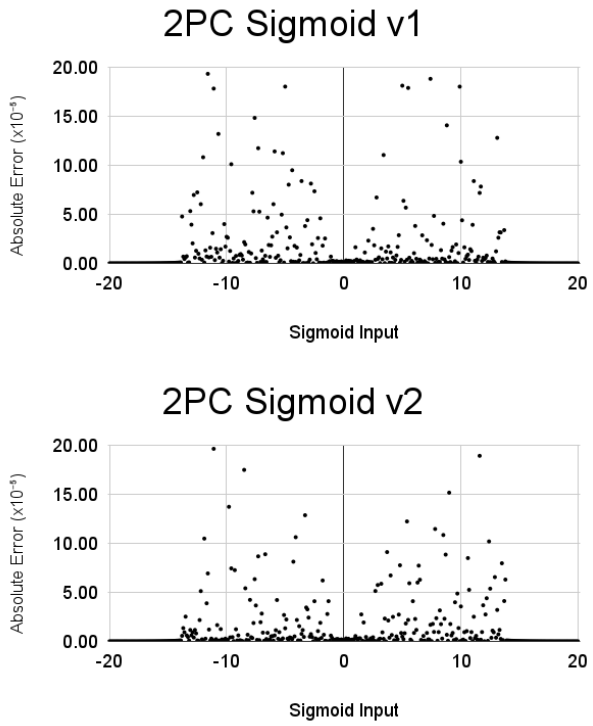

Figure 5: The y-axis shows the absolute error (scaled by 10^{-5}) of our secure sigmoid implementation (sigmoid v1 and sigmoid v2) for different values of inputs on the x-axis ranging from $[-20, 20]$. Recall the absolute error is defined as the difference between the output of our secure sigmoid implementation (over fixed point inputs with 20 fractional bits) and the baseline implementation of insecure sigmoid using the direct formula $\frac{1}{1+e^{-x}}$ (over floating point inputs). For all input values outside the range $[-20, 20]$, the absolute error is less than 10^{-8}.

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 ring elements communicated per sigmoid. Hence, sigmoid becomes a bottleneck, in terms of online communication cost, whenever the following condition is satisfied:

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

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

The above condition is often true for large datasets (i.e. where $B \gg k$) and/or when per sigmoid communication cost is high (which is true for prior accurate secure sigmoid implementations 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).

J. Failure Probability

The non-interactive fixed point truncation protocol from [64] and the single round exponentiation protocol from [57] 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 [64] has an error probability of $p_{\mathsf{trunc}} = \frac{2^{w+1}}{2^\ell}$. 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 [64]) 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 [57] has a failure probability $p_{\mathsf{exp}} \approx \frac{2^{w+1}}{2^\ell}$. 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 $(133B + k) \cdot \frac{2^{w+1}}{2^\ell} \cdot t$. Compared to SecureML [64], our failure probability only reduces the security by ≈ 7 bits while providing a much more accurate training (due to our better sigmoid approximation).

K. Semi-Honest Secure Two-Party Computation

The following description of semi-honest two-party computation is standard in the literature and has been taken from [60].

Semi-Honest Adversary. The model that we consider in this work is that of two-party computation in the presence of *static semi-honest* adversaries. Such an adversary controls one of the parties (statically, and so at the onset of the computation) and follows the protocol specification exactly. However, it may try to learn more information than allowed by looking at the transcript of messages that it received and its internal state. A protocol that is secure in the presence of semi-honest adversaries guarantees that there is no inadvertent leakage of information. Semi-honest secure protocols are often designed as the first step towards achieving the stronger notion of malicious security.

Two-Party Computation (2PC). A two-party protocol problem is cast by specifying a possibly random process that maps pairs of inputs to pairs of outputs (one for each party). We refer to such a process as a *functionality* and denote it $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, where $f = (f_1, f_2)$. That is, for every pair of inputs $x, y \in \{0,1\}^*$, the output-pair is a random variable $(f_1(x,y), f_2(x,y))$ ranging over pairs of strings. The first party (with input x) wishes to obtain $f_1(x,y)$ and the second party (with input y) wishes to obtain $f_2(x,y)$.

Privacy by Simulation. As expected, we wish to formalize the idea that a protocol is secure if whatever can be computed by a party participating in the protocol can be computed based on its input and output only. This is formalized according to the simulation paradigm by requiring the existence of a simulator who generates the view of a party in the execution. However, since the parties here have input and output, the simulator must be given a party's input and output in order to generate the view. Thus, security here is formalized by saying that a party's view in a protocol execution be simulatable given its input and output. This formulation implies that the parties learn

nothing from the protocol execution beyond what they can derive from their input and prescribed output.

One important point to note is that since the parties are semi-honest, it is guaranteed that they use the actual inputs written on their input tapes. This is important since it means that the output is well defined, and not dependent on the adversary. Specifically, for inputs x, y, the output is defined to be $f(x, y)$, and so the simulator can be given this value.

Definition of Security. We begin with the following notation:

- Let $f = (f_1, f_2)$ be a probabilistic polynomial-time functionality and let π be a two-party protocol for computing f. (Throughout, whenever we consider a functionality, we always assume that it is polynomially-time computable.)
- The view of the i^{th} party ($i \in \{1, 2\}$) during an execution of π on (x, y) and security parameter λ is denoted by $\mathsf{view}_i^{\pi}(x, y, \lambda)$ and equals $(w, r_i; m_1^i, \ldots, m_t^i)$, where $w \in \{x, y\}$ (its input depending on the value of i), r_i equals the contents of the i^{th} party's internal random tape, and m_j^i represents the j^{th} message that it received.
- The output of the i^{th} party during an execution of π on (x, y) and security parameter λ is denoted by $\mathsf{output}_i^{\pi}(x, y, \lambda)$ and can be computed from its own view of the execution. We denote the joint output of both parties by $\mathsf{output}^{\pi}(x, y, \lambda) = (\mathsf{output}_1^{\pi}(x, y, \lambda), \mathsf{output}_2^{\pi}(x, y, \lambda))$.

Definition 4. Let $f = (f_1, f_2)$ be a functionality. We say that a protocol π securely computes f in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time algorithms $\mathcal{S}_1$ and $\mathcal{S}_2$ such that

$$\left\{(\mathcal{S}_1(1^{\lambda}, x, f_1(x, y)), f(x, y))\right\}_{x,y,\lambda} \stackrel{c}{\equiv} \left\{(\mathsf{view}_1^{\pi}(x, y, \lambda), \mathsf{output}^{\pi}(x, y, \lambda))\right\}_{x,y,\lambda}$$

, and

$$\left\{(\mathcal{S}_2(1^{\lambda}, x, f_2(x, y)), f(x, y))\right\}_{x,y,\lambda} \stackrel{c}{\equiv} \left\{(\mathsf{view}_2^{\pi}(x, y, \lambda), \mathsf{output}^{\pi}(x, y, \lambda))\right\}_{x,y,\lambda}$$

where $x, y \in \{0, 1\}^*$ such that $|x| = |y|$, $\lambda \in \mathbf{N}$, and $\stackrel{c}{\equiv}$ denotes computational indistinguishability of the ensembles for all large enough values of λ.

Secure Fixed-Point Computation. Let $f : \mathbb{R} \to \mathbb{R}$ be a real-valued function. Examples of such f include multiplication of two real-valued numbers, computing a polynomial with real coefficients on a real-valued number, sigmoid etc. Since it is not possible to compute f exactly with infinite precision, we instead compute it over a fixed-point domain $\mathbb{F}$. Specifically, we compute a different function $\widehat{f} : \mathbb{F} \to \mathbb{F}$ which computes an approximation of f. Our specific choice of $\mathbb{F}$ is described in Section 2 which is standard.

In the context of 2PC, we are interested in computing a functionality $\mathcal{F}_{\widehat{f}}$ which takes in secret shares of some fixed-point input $x \in \mathbb{F}$ and outputs a secret-sharing of $\widehat{f}(x)$. In our paper, we describe protocols $\Pi_{\widehat{f}}$ for securely computing such fixed-point functionalities where the description of $\widehat{f}$ is sometimes implicit in the protocol description. The approximate nature of $\widehat{f}$ w.r.t. to f comes from the fact that: 1) we are only allowed to perform operations on fixed point values, 2) the intermediate computations in $\Pi_{\widehat{f}}$ might have some error. The latter case is true in our protocols (such as Secure Powers, Secure Polynomial, Secure Sigmoid) as it relies on the non-interactive approximate truncation protocol from prior work [64][13] as a subprotocol.

Security Proof (Sketch). Our final protocol for secure logistic regression invokes subprotocols for secure matrix multiplication and secure sigmoid. The protocol for secure sigmoid in turn invokes additional subprotocols such as secure spline, secure exponentiation, secure polynomial evaluation and secure MIC (Multiple Interval Containment). Let $\Pi_{\widehat{f}}$ be a particular subprotocol where $\widehat{f}$ is the approximate function that it is computing. For example, $\Pi_{\widehat{f}}$ can be our secure polynomial protocol from Appendix L.5 where $\widehat{f}$ would be the (implicit) approximate polynomial evaluation function. All of our protocols are designed in the preprocessing model where the protocol Π depends on some correlated randomness available from the offline phase. Let $\mathcal{F}_{\widehat{f}}$ denote a functionality which takes as input secret shares of some fixed-point value $x \in \mathbb{F}$ and outputs a secret-sharing of $\widehat{f}(x)$. Let $\mathcal{F}_{\widehat{f}}^{\mathsf{offline}}$ denote the setup functionality which generates the preprocessing material required for securely computing $\widehat{f}$ and sends it to the parties.

We can argue that $\Pi_{\widehat{f}}$ securely computes $\mathcal{F}_{\widehat{f}}$ in the $\mathcal{F}_{\widehat{f}}^{\mathsf{offline}}$ hybrid model. We note that all of our subprotocols in this paper, except for secure comparison, make use of the non-interactive secure truncation protocol from SecureML [64] for performing fixed point multiplication. Therefore, for proving security of such subprotocols, we will abstract away the truncation protocol as an ideal functionality $\mathcal{F}_{\mathsf{truncate}}$ and prove security in a hybrid model where $\mathcal{F}_{\mathsf{truncate}}$ is available in addition to $\mathcal{F}_{\widehat{f}}^{\mathsf{offline}}$. The exact function implemented by $\mathcal{F}_{\mathsf{truncate}}$ is implicit in the protocol description of SecureML [64].

For proving security, we need to construct a simulator $\mathcal{S}_{\widehat{f}}$ which simulates the protocol in the ideal world. This needs to be done in two steps - first, we need to simulate the preprocessing material, and secondly, we need to simulate the online protocol interaction. For subprotocols which are not based on FSS (such as secure matrix multiplication, secure polynomial evaluation and secure exponentiation), the preprocessing material can be simulated honestly by having the $\mathcal{S}_{\widehat{f}}$ internally execute $\mathcal{F}_{\widehat{f}}^{\mathsf{offline}}$ and sending the adversary its part of the preprocessing material. For simulating the preprocessing material of FSS-based protocols (such as secure spline, secure comparison, secure MIC), we can execute a simualtor $\mathcal{S}_{\mathsf{FSS}}$

13. Although it is possible to use exact truncation protocols from the literature at the cost of increased rounds of interaction, we made a choice to stick with the non-interactive approximate truncation protocol [64] for efficiency reasons and the fact that this approximation only introduces atmost 1 bit of error in the least significant bit of the fractional part which can be offset by increasing the precision scale.

for generating fake FSS keys. Such a simulator is already provided in prior works for DPF [20], DCF [16] and iDPF [13] and its security relies on the assumption of One Way Function (OWF). We note that in all of our subprotocols, simulating the view of the adversary in the online protocol is straightforward as parties only exchange uniformly random elements which can be perfectly simulated. In the overall protocol for secure logistic regression, we can invoke the simulator for all of the underlying subprotocols in order to simulate the view of adversary. For technical reasons, as pointed out in [10], a resharing step needs to be added in the end of secure logistic regression protocol to upgrade from view-indistinguishability to indistinguishability of the joint distribution of adversary's view and honest party's output. Practically, this can be cheaply done by having P_0 and P_1 locally sample a random value r_0 and r_1 respectively from the domain of the weight vector, sending it across to the other party, and then adding $r_0 + r_1$ locally to their output shares.

L. Secure Sigmoid with Trusted Setup

In this section we provide the details of the sigmoid construction in the trusted setup model from Section 5.

L.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 L.2 , 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 $\mathbb{Z}_{2^\ell}$ and compute the $\vec{Q}$ over $\mathcal{R} = \mathbb{Z}_{2^\ell}$, we can ensure that the final output of spline protocol is shared in $\mathcal{R} = \mathbb{Z}_{2^\ell}$ to be compatible for further computations.

L.2. 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 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.

L.3. Spline Concrete Instantiation Details

We approximate sigmoid on $[0, 1)$ by splitting the interval into m equally sized intervals. To do so, we define a series of $m + 1$ points $\{\alpha_i\}_{i\in[m+1]}$ where $\alpha_1 = 0 < \alpha_2 < \ldots < \alpha_m < \alpha_{m+1} = 1$ and $\alpha_{i+1} - \alpha_i = \frac{1}{m}$ for all $i \in [m]$. Then we define m linear univariate polynomials

whose coefficients w denote as $\{a_i, b_i\}_{i\in[m]}$. The sigmoid approximation is then computed as follows:

$$\text{Sigmoid}(x) = \begin{cases} a_1 x + b_1 & \alpha_1 \le x < \alpha_2 \\ a_2 x + b_2 & \alpha_2 \le x < \alpha_3 \\ \quad \dots & \\ a_{m-1} x + b_{m-1} & \alpha_{m-1} \le x < \alpha_m \\ a_m x + b_m & \alpha_m \le x < \alpha_{m+1} \end{cases}$$

The coefficient values for the i^{th} interval are computed by interpolating a line between the coordinates $(\alpha_i, \sigma(\alpha_i))$ and $(\alpha_{i+1}, \sigma(\alpha_{i+1}))$, where $\sigma(\cdot)$ denotes the exact sigmoid function. Table 8 describes the specific values for the coefficients that were used in our experiments for $m = 10$ intervals. Note that the values of the coefficients in Table 8 are the exact real number values which are converted into a fixed point representation when performing the secure computation.

i	a_i	b_i
1	0.24979187478940013	0.5
2	0.24854809833537939	0.5001243776454021
3	0.24608519499181072	0.5006169583141158
4	0.24245143300792976	0.5017070869092801
5	0.23771671089402596	0.5036009757548416
6	0.23196975023940808	0.5064744560821506
7	0.2253146594237077	0.5104675105715708
8	0.2178670895944635	0.5156808094520418
9	0.20975021497391394	0.5221743091484814
10	0.2010907600500101	0.5299678185799949

TABLE 8: Spline parameters for instantiating our sigmoid approximation. We use the fixed-point representation of these values with $s = 20$ fractional bits.

The value $m = 10$ was decided by evaluating the spline approximation for different values of m and checking the average ULP error (defined w.r.t. a fixed fractional scale of $s = 20$ bits), as used in [70], for sigmoid inputs drawn uniformly at random in $[0, 1)$. For $m = 10$, we obtained an average ULP error of 46 which corresponds to an absolute error of less than 0.00005. This seemed to be a reasonable cutoff for the accuracy of logistic regression hence we made the choice of $m = 10$. Note that the online communication cost and rounds of our secure sigmoid protocol is independent of m.

As mentioned in Section 4.1, we use the spline-based approximation only on the interval $[0, 1)$ (which automatically also provides an approximation on $(-1, 0]$ due to the symmetric nature of sigmoid curve) and use exponentiation combined with Taylor-series based approximation on the interval $[1, \infty)$ (and symmetrically on $(-\infty, -1]$). We chose this approach rather than using only splines to approximate sigmoid on the entire $(-\infty, \infty)$ interval, as has been done in prior works [64], as it would require a large number of spline intervals[14] and potentially higher degree splines for the approximation to work well, thus increasing the cost of protocol.

In the v1 setting (i.e. secure sigmoid with trusted offline setup), this increase in cost would be reflected in the amount of FSS preprocessing material that parties need to store. Specifically, for m intervals and d degree splines, the cost would be proportional to md without accounting for fixed-point related issues [16].

In the v2 setting (i.e. secure sigmoid with distributed offline setup), we would have to replace the DCF underlying FSS based spline gate with m instances of a secure comparison protocol supporting efficient offline phase for large bit-lengths (e.g. our protocol in Section 6.1). This is because the FSS based spline gate would require a DCF key for ℓ bit inputs where ℓ is the bit-length of the ring. However, generating such a key using 2PC is currently practical for only small values of ℓ (such as $\ell \le 20$). Note that we do not run into this issue when doing the spline-based approximation on $[0, 1)$ as the underlying FSS is only invoked on the fractional part of the input which can be cast into a smaller ring of s bits where s represents the scale of fixed-point representation.

L.4. 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 $\{x, x^2, \dots, x^d\}$ for a (secret-shared) input x. Naively, one could invoke $\mathcal{F}_{\mathsf{Mult}}$ repeatedly d times to generate these powers. However, this makes the communication-cost proportional to the degree d. In [62], 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 $\{r, r^2, \dots, r^d\}$ 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 $\{x, x^2, \dots, x^d\}$. The main observation in the protocol is the following relationship:

$$[\![x^i r^j]\!] = [\![r^{i+j}]\!] + x_{\mathsf{mask}}\Big(\sum_{\ell=0}^{i-1} [\![x^{i-1-\ell} r^{j+\ell}]\!]\Big) \quad (4)$$

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 4 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 Equation 4 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[15]), (ii) Ensuring that none of the intermediate values in the computation wrap around the ring, since a multiplication wrapping prior to

14. As an estimate, our experiments indicated that performing a degree 1 spline-based approximation of sigmoid on $[-10, 10]$ will yield an average ULP error of 479434 (defined w.r.t. a fixed fractional scale of $s = 20$ bits) even after using an enormous $m = 10^7$ intervals. This corresponds to an average absolute error of 0.46, which is quite large.

15. 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.

truncation corrupts the share. While incorporating the first condition into Equation 4 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 wrap-around is not an issue when evaluating Equation 4 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 10.

In the actual protocol, we invoke a fixed-point adapted version of the powers protocol from [62] 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 10 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 $1+2k\ell$ when realizing $\mathcal{F}_{\mathsf{MUX2}}$ using OT as described earlier. Thus the total communication happens to be $2(s+1+2k\ell)$ bits.

L.5. Secure Polynomial Evaluation

Suppose parties hold a secret-sharing (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 coefficients $a_i \in \mathcal{R}$ 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 the coefficients a_i. Thus, the overall procedure would require the same online communication 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 of the protocol to only $2(s+1+2\ell)$ bits, making it *independent* of the degree d of the polynomial Q. We refer to this optimized protocol as Π_{fxpPoly}.

Algorithm 10: 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: $[\![\widehat{r}]\!]$, $[\![\widehat{r^2}]\!]$, …, $[\![\widehat{r^k}]\!]$, 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 $x^0_{\mathsf{mask}} := 0^{\ell-s}||x_{\mathsf{mask}}$
5 $\{p_i^1\}_{i\in[k]} \leftarrow \Pi_{\mathsf{maskPowers}}(x^1_{\mathsf{fracMask}})$, where $x^1_{\mathsf{mask}} := 1^{\ell-s}||x_{\mathsf{mask}}$
// The value of $x - r$ is x^0_{fracMask} if $x \geq r$, else x^1_{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 P_i
$\Pi_{\mathsf{maskPowers}}$:

Input: x_{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 $\quad A_{0,i} \leftarrow [\![\widehat{r^i}]\!]$
4 **for** ℓ = 1 *to* k **:**
$\quad$ // Compute all $A_{i,j}$ where $\ell = i + j$
5 $\quad sum \leftarrow 0$
6 $\quad$ **for** i = 1 *to* ℓ **:**
7 $\quad\quad j \leftarrow \ell - i$
8 $\quad\quad sum \mathrel{+}= A_{i-1,j}$
$\quad\quad$ // Invariant : $sum = \sum_{k<i} [\![\widehat{y^{i-1-k} r^{j+k}}]\!]$
9 $\quad\quad A_{i,j} \leftarrow [\![\widehat{r^{i+j}}]\!] + \mathcal{F}_{\mathsf{fxpMult}}(x_{\mathsf{mask}}, sum)$
$\quad\quad$ // Invariant: $A_{i,j}$ will store $[\![\widehat{y^i r^j}]\!]$ following Equation 4
10 **return** $A_{1,0}, A_{2,0}, \ldots, A_{k,0}$