Communication Efficient Secure Logistic Regression

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

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

Mariana Raykova
Google, New York
marianar@google.com

Phillipp Schoppmann
Google, Berlin
schoppmann@google.com

Karn Seth
Google, New York
karn@google.com

ABSTRACT

We present a new two-party construction for secure logistic regression training, which enables two parties to train a logistic regression model on private secret shared data. Our goal is to minimize online communication and round complexity, while still allowing for an efficient offline phase. As part of our construction we develop many building blocks of independent interest. These include a new approximation technique for the sigmoid function, which results in a secure evaluation protocol with better communication; secure spline evaluation and secure powers computation protocols for fixed-point values; and a new comparison protocol that optimizes online communication. We also present a new two-party protocol for generating keys for distributed point functions (DPFs) over arithmetic sharing, where previous constructions do this only for Boolean outputs. We implement our protocol in an end-to-end system and benchmark its efficiency. We can securely evaluate a sigmoid in 20 ms online time and 1.12 KB of online communication. Our system can train a model over a database with 6000 samples and 5000 features with online communication of 40 MB and online time of 9 minutes.

1 INTRODUCTION

One of the most ubiquitous ways to understand and use large amounts of data is to train models which capture the most significant general properties of the underlying data. In many settings the dataset used for the model training is owned by different parties that have agreed to cooperate and create a common model across their datasets but do not want to share record level data. Secure multiparty computation (MPC) [25, 49] 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 [7], 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 [6]. 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 [16] 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 [15] where the two aggregators were run by Mozilla and the Internet Security Research Group (ISRG). The same design underlies the Exposure Notifications Private Analytics (ENPA) system implemented by Google and Apple in their Exposure Notifications system [4], 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 [28], 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 [27], considers a similar measurement goal, which is to compute aggregate measurement across attributed conversions from all users. Again, an MPC system with distributed data stewardship can be used to perform this kind of measurement [29].

While the previous two examples show that privately aggregating user data into histograms is useful by itself, the 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[4]), this incurs

egress charges for all traffic between the two servers, which can be significantly higher than intra-cloud traffic costs. 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 [9, 22], engineering [40], 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 [24].

Online-offline Computation Model. Our constructions consider the online-offline computation model [21] which aims to minimize the complexity of the protocol that is on the critical path of processing inputs when they become available, by outsourcing some of the computation into an input-independent offline phase which can be executed at any time prior to the online stage. The main metric that we optimize for in our constructions is communication complexity which as we discussed above could be a major cost in many cross platform two party computation settings.

We consider two settings. The first one assumes a trusted offline preprocessing that can be executed centrally. This is relevant in scenarios where there is a party which can be trusted to honestly compute the different types of correlated randomness such as multiplication triples, function secret sharing (FSS) keys, and others. For example, in some scenarios regulator parties might be considered trusted for the purposes of this preprocessing. Another way to think about trusted preprocessing is measurement settings over large numbers of clients, where the offline phase is distributed across the clients each of which evaluates a small amount of the required preprocessing and submits the output together with its data shares to the two computation servers. The second setting that we address in our protocols does not assume a trusted party for the offline stage and proposes that it is also executed using secure computation between two computation parties. While it is well-known that any computation that a trusted party could perform, can also be distributed using MPC [25, 49], efficiency is a concern here as well. We therefore also investigate how to efficiently perform the offline phase of our protocols using MPC, while still 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 [20] is one approach to guarantee that it cannot be used to extract individual records. Thus, we also consider the questions of constructing a distributed protocol for differentially private logistic regression training.

Our Contributions. We present a new construction for two party secure logistic regression training over a database that is cryptographically shared between the two parties and improves the online communication cost of existing approaches. We present two different protocols: the first one optimized 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 differentially private output model.

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 and the final protocol offers improved communication cost for its online phase which is 3-4x smaller than the communication of the state of the art sigmoid construction of SiRnn [41].

As building blocks in our main protocol we introduce several constructions for functionalities that are of independent interest:

- A new construction for spline evaluation, which supports fixed-point representation of the input.
- A new construction for secure computation of powers of a value with fixed-point representation. We use this construction for secure evaluation of Taylor series approximations.
- A new construction for secure comparison with improved online communication.
- A new construction for two-party generation of distributed point function (DPF) keys with arithmetically shared output values.

We present an end-to-end implementation of our protocols, which is the first implementation that combines FSS-based and secret-sharing-based techniques. We evaluate the costs of our protocols and present a variety of benchmarks including microbenchmarks for a new secure comparison which allows to trade-off online and offline communication and cuts in half the online communication needed by the most recent construction in CrypTFlow2 [42]. Our sigmoid construction also improves 3-4x the online communication of the state of the art solution of SiRnn [41]. A secure sigmoid evaluation runs between 16 and 19ms in different network setting and includes 1-1.5 KB of communication. We can compare 128 bits with 512-688 bits exchanged online. Our final secure logistic regression securely training trains a model over 6000 samples with 5000 features in about 9min with 30-40 MB of communication with accuracy close to the plaintext trained model.

1.1 Our Approach

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

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

To reduce the communications of the online computation of our protocol we rely on techniques for function secret sharing which enables non-interactive computation during the online phase. In particular we use the multiple interval containment (MIC) functionality [10] to identify the input interval to use the approximation function as well as within the spline approximation to choose the right linear function.

Distributed Comparison Function (Appendix B). MIC gates leverage distributed comparison functions (DCFs) [10] which rely on function secret sharing [11] techniques. We introduce a new reduction from DCFs to incremental distributed point functions (iDPFs) [8], which is conceptually simpler and cheaper than the previous construction by Boyle et al. [10].

Secure Powers Computation with Fixed-Point Representation (Section 5.2). The sigmoid approximation for values above 1 that relies on Taylor approximation has two main components: secure exponentiation for evaluation of e^{-x} and the a secure protocol for power computation that enables the polynomial evaluation for the Taylor series. For the first part we leverage the construction for secure exponentiation of Kelkar et al. [33]. For the second part we present a new construction inspired by the Honey Badger secure powers computation protocol [37], which we extend to work with fractional values in fixed-point representation.

Online-Offline Balanced Protocol (Section 6). The most significant part of the offline computation for our protocol is the generation of the FSS keys, which are needed for the MIC gates. In the setting without trusted preprocessing where these keys need to be generated using two party computation, this presents significant costs that challenge the execution of the offline computation. Existing approaches either rely on general-purpose MPC, which in this case is expensive due to the need for secure evaluation of a PRG, or they use the Doerner-Shelat technique [18], which requires computation exponential in the input size. In the application of the MIC gate for the 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 if we are performing the spline evaluation. However, in the higher level interval containment functionality which identifies which type of approximation needs to be used, this is not longer the case, since we don't have any simple way to reduce the size of the input. This means that we would need an FSS gate with a large input domain, which would have extremely high offline computation.

Secure Comparison (Section 6.1). To overcome this challenge we modify the protocol to use a secure comparison functionality instead MIC to determine the first level of input partitions. We introduce a new comparison construction with a highly communication-efficient online phase while only having modest computation and round complexity. Our work uses techniques from secret-sharing MPC and FSS to improve over the online communication of Rathee et al.'s CrypTFlow2 [43] by $\approx 2.4\times$ and Couteau [17] by $\approx 4.1\times$ for 64-bit inputs and appropriate parameters. We reduce the number of communication rounds by similar factors, i.e., from 6 and 12 rounds respectively to 3.

Secure comparison is a fundamental building block for higher-level privacy-preserving applications. Couteau [17] present an extensive list of such applications including oblivious sorting, database search constructions, private set intersection, oblivious RAM, machine learning for applications such as classification, feature extraction, and generating private recommendations.

Secure DPF Key Generation (Appendix D). One of the main components for the offline computation in our construction is the generation of FSS keys for distributed point functions (DPFs). Doerner and Shelat [18] present a two party computation protocol for the DPF key generation algorithm of Boyle et al. [11]. However, this protocol only supports DPFs with boolean (that is, XOR-shared) output groups.

Many applications relying DPFs including the works of Boyle et al. [10], however, require arithmetically-shared output groups. While Boyle et al. [10] refer to Doerner and shelat [18] in their paper, the construction does not immediately generalize to that setting without additional modifications. One solution is using DPFs with boolean output shares, and then use a share conversion protocol (for example the one from Cryptoflow2 [42]). However, this usually requires a number of oblivious transfers proportional to the size of the output shares, which increases the online round and communication complexity

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

1.2 Related Work

A large body of work [2, 35, 39, 44, 47] has looked at the problem of computing various machine learning and statistical models within MPC. A subset of these works [33, 39] have focused on forms of regression (SecureML, Secure Poisson Regression). Other works such as MP-SPDZ [34], EMP-Toolkit [48], ABY^3 [38] and CryptFlow2 [42] provide solutions for logistic regression in the context of a general MPC/ML framework. In most approaches, the main difficulty lies in the computation of the nonlinear function, which for logistic regression is the sigmoid function.

Other recent works have focused on efficient implementations of the sigmoid function [36, 41, 44] using a variety of methods. Of these, SiRnn [41] has the best efficiency, using a dynamic fixed point representation together with Goldschmidt's approximations to get efficient protocols for sigmoid and other ML-related functionalities.

A separate thread of work has focused on function-secret-sharing (FSS) as a communication-efficient primitive for a variety of MPC problems, including learning. One prominent work by Boyle et al. [10] shows how to compute a variety of nonlinear functions using FSS, including spline-based approximations to the sigmoid function. However, these construction play poorly with fixed-point representations of inputs, since they do not handle truncated multiplications which are critical for fixed-point correctness, and therefore are hard to use for logistic regression. Other works such as AriaNN [45] leverage FSS gates to compute ReLU gates, and therefore neural nets, with low communication, but do not tackle logistic regression.

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

Secure Computation. Secure computation protocols enable functionalities where parties can compute a function on their joint private inputs in a way that guarantees only the output of the computation is revealed. Our protocol constructions are in a two-party setting and provide semi-honest security [25], i.e., the parties are assumed to follow the prescribed protocol. We denote the two parties by P_0 and P_1. The protocol may be divided into an offline preprocessing phase (independent of parties' inputs) and an online phase that depends on parties' inputs. The offline preprocessing may be performed by a trusted third party, or by the parties executing an MPC protocol.

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 [39]: Suppose x_0 and x_1 are the shares of $\widehat{x}$ held by party P_0 and P_1 respectively. Then, in order to perform truncate operation, both parties can just locally truncate the last $s - s'$ bits of their individual shares to get new shares x_0' and x_1'. Let x' denote the true truncated value of x after truncating the last $s - s'$ bits. Mohassel and Zhang [39] show that $\mathsf{Recon}(x_0', x_1') \in \{x' - 1, x', x' + 1\}$. In other words, this non-interactive truncation protocol incurs a small error in the least significant bit of the fractional part of the FXP value. For our purposes, this error will be tolerable as the FXP representation itself admits an error in the least significant bit of the fractional part compared to the actual real value.

2.1 Logistic Regression

Logistic regression is a probabilistic classifier which uses supervised machine learning [32]. The classification function f takes an observation which is a vector of features $\vec{x}_i$ and outputs the estimated class with highest likelihood. It leverages the sigmoid functionality $\sigma(z) = \frac{1}{1-e^{-z}}$ to assign probability an input feature vector $\vec{x}$ maps to class using the weight vector $\vec{w}$ and a bias term b describing the model. The evaluation $\sigma(\vec{(x)} \cdot \vec{w} + b)$ gives the probability of mapping $\vec{x}$ to the class 1.

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

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

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

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

SGD runs until convergence when the gradient norm falls below a threshold. However, in the context of secure computation protocols we will run a fixed number of iterations to avoid leakage about the private samples based on the time for convergence. The mini-batch technique makes each iteration over a subset of the samples rather than the whole batch.

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

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

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

2.2 Multiplication Triples

Suppose parties are holding additive-sharing of values $x, y \in \mathcal{R}$. Then they can use a functionality $\mathcal{F}_{\mathsf{Mult}}$ to get an additive sharing 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 correlated randomness in the form of beaver triples in the offline phase, and then consuming them in the offline phase. This incurs an online communication of 2 ring elements per-party.

For multiplying an $n \times m$ matrix $\mathbf{X}$ with another $m \times k$ matrix $\mathbf{Y}$, one could call a protocol for $\mathcal{F}_{\mathsf{Mult}}$ repeatedly on individual

values in the matrix. This would incur an online of $2nmk$ ring elements per party. However, there is a more (online) efficient matrix multiplication protocol based on *matrix* beaver triples which incurs an online communication of $2(nm + mk)$ per party. Hence, 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 yet another optimization compared to the naive method of calling $\mathcal{F}_{\mathsf{matMult}}$ repeatedly. This optimization is based on *correlated* matrix beaver triples and incurs an online communication of $2(nm + m\sum_i k_i)$ per party. Hence, we will use $\mathcal{F}_{\mathsf{corrMatMult}}$ to abstractly represent a functionality which enables this kind of multiplication.

Suppose parties are holding a boolean sharing of values $x, y \in \{0, 1\}$. Then they can use $\mathcal{F}_{\mathsf{AND}}$ functionality to get a boolean sharing of $z \in \{0, 1\}$ such that $z = x \wedge y$. The protocol for realizing $\mathcal{F}_{\mathsf{AND}}$ is similar to the protocol for realizing $\mathcal{F}_{\mathsf{Mult}}$. It uses *bit* beaver triples and incurs an online communication of 2 bits elements per-party.

Although the above multiplication functionalities are defined for integers, they can also be extended to real numbers represented in fixed-point format. For realizing such functionalities for fixed-point inputs, parties can use the same protocol that works over integers with an additional protocol for truncation at the end, where s least significant bits are truncated from the result in order to adjust the fractional scale. In our work, we use the non-interactive truncation protocol from SecureML [39] described in Section 2.

2.3 Function Secret Sharing

We use Boyle et al.'s definition of function secret sharing (FSS) [11]. At a high level, 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. Note that the output reconstruction in (2) is *additive*. Now, we define formally.

Definition 2.1. A 2-party FSS scheme is a pair of algorithms $(\mathsf{Gen}, \mathsf{Eval})$ such that:

- $\mathsf{Gen}(1^\lambda, \widehat{f})$ is a probabilistic polynomial time algorithm that given 1^λ and $\widehat{f}$, 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 $\mathbb{G}^{out}$.
- $\mathsf{Eval}(b, k_b, x)$ is a polynomial time algorithm that for a party index b, a key k_b defining $f_b : \mathbb{G}^{in} \to \mathbb{G}^{out}$ outputs $f_b(x) \in \mathbb{G}^{out}$.

Indeed, one can trivially split a function f by secret sharing f's truth table. The key challenge is to *compactly* represent the function shares f_0 and f_1 while keeping `Eval` efficient.

Secure Computation via FSS. [12] showed that the FSS paradigm can be used to efficiently evaluate some function families in 2PC. This 2PC via FSS can be cast in the preprocessing model, where Gen and Eval correspond to the offline/online phase, respectively. As in secret-sharing MPC, the function is represented as a circuit of gates. The parties step though the circuit gate-by-gate, following the semantics of gate evaluation in the respective protocol. Note that unlike in secret-sharing MPC, the FSS inputs and outputs are *public* whereas the function is *secret-shared.* As the parties cannot learn the values on any intermediate circuit wires, the protocol needs to take care to use masked inputs and outputs for each gate $g : \mathbb{G}^{in} \to \mathbb{G}^{out}$. That is, the input is masked with r^{in} and the output with r^{out}. Then, each gate g implements an offset function $g^{[r^{in}, r^{out}]}(x) = g(x - r^{in}) + r^{out}$. Hence, each gate first unmasks the input x and only then executes the function g. The output of g is masked with r^{out} prior to reconstruction. This step is repeated for each gate until both parties evaluate the last circuit gate. Importantly, P_0 and P_1 learn the output mask of the last gate, allowing them to obtain the circuit output.

Additional definitions and useful preliminaries appear in Appendix A.

3 SECURE LOGISTIC REGRESSION

Our goal is to develop concretely-efficient secure two-party computation protocols for logistic regression training, in particular focusing on online communication and rounds. As previous works in this direction [39, 46], we work with arithmetic secret-sharing (see Section 2) and train the model with stochastic gradient descent (SGD). We note that there are alternatives to SGD for training logistic regression, such as Bayesian approaches, conjugate gradient descent, and Newton's method. However, these have not been deeply explored in MPC, and are generally less efficient even in plaintext, hence we do not explore them in this work.

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

As we describe in the following sections, our novel contributions lie in the construction of sigmoid using a mix of MPC primitives including DCFs, DPFs, Taylor approximation, and an efficient secure exponentiation protocol.

Algorithm 1: Logistic Regression Protocol

Public inputs: Number of iterations T, dataset dimensions n, k, batch size B, learning rate α, regularization parameter λ.
Private inputs: Secret-shared dataset $[\![X]\!] \in R^{n \times k}$ and labels $[\![\mathbf{y}]\!] \in R^n$.

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

4 SECURE SIGMOID

The key challenge of computing a single step of SGD is evaluating real-valued sigmoid function. Traditionally, MPC protocols were

designed to evaluate functions represented either as Boolean circuits or arithmetic circuits. Sigmoid function cannot be succintly represented as such a circuit. This is because the sigmoid function requires to compute (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 traditionally approximated via Goldschmidt's or Newton's methods, which are expensive iterative methods. Similarly, exponentiation is either approximated by decomposing the exponent into bits, which is costly, or via low-degree polynomials and piecewise linear functions, which are inaccurate.

In this section, we present our sigmoid functionality (Algorithm 2). Note that this functionality represents our sigmoid approximation and differs from the cleartext sigmoid. We explain how our sigmoid approximation is MPC-friendly and describe how we securely implement this functionality. We point to Section 5.1 and Section 5.2 for detail on how we implement the more complex components of our functionality.

4.1 Sigmoid Approximation

As can be seen from Figure 3 in the appendix, the sigmoid function is 'symmetric' around the y-axis. More specifically, $S(x) + S(-x) = 1$ for all $x \in R$. This implies we can focus on evaluating $S(x)$ and then compute $S(-x) = 1 - S(x)$ locally.

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

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

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

This approximation requires to compute additions and powers of r. As a result, it can be expressed as an arithmetic circuit, and thus is MPC-friendly. While addition is a virtually free local operation, computing powers is an expensive interactive operation. We present a concretely efficient protocol for computing powers in Section 5.2 based on the protocol of [37]. Our protocol computes *all* powers of r (irrespective of the degree) in only 2 communication rounds.

However, this approximation works well only when $r << 1$. We therefore use this approximation only on the interval $[0, \frac{1}{e}]$. As $r = e^{-x}$, we use this technique when $x \geq 1$. In order to compute e^{-x}, we use the 1-round exponentiation technique of Kelkar et al. [33]. We note that the exponentiation protocol from [33] assumes a known (arbitrary) bound on how negative the exponent can be. So in order to comply with that assumption, we do not use this exponentiation protocol if the exponent is too negative. Rather, we just set the sigmoid output directly to 1. We fix the bound as $l_f/\log_2(e)$, i.e. whenever $x \geq l_f/\log_2(e)$, we set the sigmoid output to 1. This bound can be justified by observing that for any $x \geq l_f/\log_2(e)$, $e^{-x} < 2^{-l_f}$. Hence the fixed point representation of the result of exponentiation is exactly 0 in this case.

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

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

Algorithm 2: Approximate Sigmoid

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

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

$\mathsf{Sigmoid}(x)$:

```
1  if x < 0 then
2      S ← 1 − Sigmoid(−x).
3  else
4      if x < 1 then
5          S ← Spline(m, [0, 1))
6      else
7          if x log_2(e) ≥ l_f then
8              S ← 1
9          else
10             r ← e^{−x}
11             S = 1/(1+r) ← 1 − r + r^2 − . . . ± r^d
12 return S
```

5 SECURE SIGMOID WITH TRUSTED OFFLINE SETUP

In this section, we describe details for our approach for *securely* computing the sigmoid approximation described in Algorithm 2 with a focus on minimizing the online communication complexity. Towards that end, we assume that the offline phase is part of a trusted setup phase. In practical settings, such a trusted setup can be performed by a trusted third party. Another possibility, when the intermediate models are protected by DP (see Algorithm 11 in Appendix E), is to outsource the setup phase to (semi-honest) clients. These may provide a portion of the precomputed setup alongside the inputs they upload to the two MPC parties. We will discuss how to perform the offline phase in MPC as well in Section 6.

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

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

5.1 Secure Spline Computation

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

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

5.2 Secure Powers Evaluation

To evaluate a Taylor series approximation inside MPC, we need a procedure to securely compute a d-degree polynomial which, in turn, requires computing the (secret-shares of) consecutive powers $\{x, x^2, \ldots, x^d\}$ for a (secret-shared) input x. Naively, one could invoke $\mathcal{F}_{\mathsf{Mult}}$ repeatedly d times in order to generate these powers. However, this makes the communication-cost proportional to the degree d. In [37], the authors proposed a novel protocol to generate all d powers using a single element of online communication per party, where the masked value $x_{\mathsf{mask}} = x - r$ is revealed. The protocol leverages a new type of offline pre-processing correlation called "random powers". In such a correlation, parties have a sharing of $\{r, r^2, \ldots, r^d\}$ for a uniformly random $r \in \mathcal{R}$. For a (secret-shared) input x in the online phase, the parties "consume" these special correlations in order to generate a sharing of $\{x, x^2, \ldots, x^d\}$. The main observation in the protocol is the following relationship:

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

The aforementioned protocol works only for integer inputs (mapped to ring elements in the natural way) and it is unclear how to directly extend it to inputs represented in fixed-point format. The main challenge is that Equation 1 now needs to be evaluated over real numbers instead of ring elements in order to get the correct result. We observe that emulating the evaluation of Eq. 1 over reals inside a ring requires the following: i) Performing fixed point multiplications instead of ring multiplication (i.e. we need to perform a truncation operation after every ring multiplication to adjust the scale [2]), ii) Ensuring that none of the intermediate values in the computation wrap around the ring, since a multiplication wrapping prior to truncation corrupts the share. While incorporating the first condition into Eq. 1 might seem straightforward, it is less obvious how to incorporate the second condition. The reason is that the term r^{i+j} will almost always wrap around the ring when r is sampled from the fixed-point region of the ring. Note that this wraparound is not an issue when we want to evaluate Eq. 1 over integers.

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

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

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

In the actual protocol, we invoke a fixed-point adapted version of the powers protocol from [37] on both $+x_{\mathsf{fracMask}}$ and $-x_{\mathsf{fracMask}}$. Then parties can select the correct set of powers using a multiplexer where the selection bit is set to t. We describe our complete protocol in Algorithm 3 where we use $\mathcal{F}_{\mathsf{MUX2}}$ as a black-box.

When $\mathcal{F}_{\mathsf{MUX2}}$ is replaced by an actual 2-round OT protocol, the first round of OT can be parallelized with Line 2 by invoking $\mathcal{F}_{\mathsf{MUX2}}$ on $(p_i^c, p_i^{1 \oplus c}, f)$ instead, thus making the selection bit of $\mathcal{F}_{\mathsf{MUX2}}$ independent of the result of reconstruction on Line 2. Hence, the overall protocol will require 2 online rounds. The per-party online

[1] In our experiments, we implemented the naive approach without this optimization. Our calculations show that the overall sigmoid communication in our experiments will be reduced to ≈ 58% in the distributed (2PC) offline setting and to ≈ 44% in the trusted offline setting.

[2] 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.

communication cost is s bits for Line 2 and $1+2kl$ when realizing $\mathcal{F}_{\mathsf{MUX2}}$ using OT as described earlier. Thus the total communication happens to be $2(s+1+2kl)$ bits.

Algorithm 3: Fixed-point powers Protocol

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

Input : $[\![x]\!]$, where $x \in [0, 2^s)$ and $\widetilde{x} \in [0,1)$

Output : $[\![\widehat{y}]\!], [\![\widehat{y^2}]\!], \ldots, [\![\widehat{y^k}]\!]$, where $y = \widetilde{x}$

Precomputation: $[\![\widehat{r}]\!], [\![\widehat{r^2}]\!], \ldots, [\![\widehat{r^k}]\!]$, where $r \in \mathbb{R}$ and $r \leftarrow [0,1)$

1 $[\![x-r]\!] \leftarrow [\![x]\!] - [\![r]\!]$
2 $x_{\mathsf{fracMask}} := \mathsf{Recon}([\![x-r]\!]^s)$, where $[\![x-r]\!]^s$ is the s least significant bits of $[\![x-r]\!]$ and Recon happens in the ring Z_{2^s}.
3 Let $\langle c \rangle$ be a default sharing of bit c denoting the public carry bit in the most significant place during the above additive reconstruction.
4 $\{p_i^0\}_{i\in[k]} \leftarrow \Pi_{\mathsf{maskPowers}}(x^0_{\mathsf{fracMask}})$, where $x^0_{\mathsf{mask}} := 0^{l-s} || x_{\mathsf{mask}}$
5 $\{p_i^1\}_{i\in[k]} \leftarrow \Pi_{\mathsf{maskPowers}}(x^1_{\mathsf{fracMask}})$, where $x^1_{\mathsf{mask}} := 1^{l-s} || x_{\mathsf{mask}}$
// The actual value of $x-r$ is x^0_{fracMask} if $x \geq r$, and x^1_{fracMask} otherwise.
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 $l = i + j$
5 $\quad s \leftarrow 0$
6 $\quad$ **for** $i = 1$ *to* l **:**
7 $\quad\quad j \leftarrow l - i$
8 $\quad\quad s \mathrel{+}= A_{i-1,j}$
$\quad\quad$ // Invariant : $s = \sum_{k<i} [\![\widehat{y^{i-1-k} r^{j+k}}]\!]$
9 $\quad\quad A_{i,j} \leftarrow [\![\widehat{r^{i+j}}]\!] + \mathcal{F}_{\mathsf{fxpMult}}(x_{\mathsf{mask}}, s)$
$\quad\quad$ // Invariant: $A_{i,j}$ will store $[\![\widehat{y^i r^j}]\!]$ following Equation 1
10 **return** $A_{1,0}, A_{2,0}, \ldots, A_{k,0}$

5.3 Secure Polynomial Evaluation

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

5.4 Theoretical Cost Analysis of Secure Sigmoid

We will now analyze the online communication cost for our sigmoid protocol. Our overall secure sigmoid construction invokes 1 MIC gate requiring $2l$ bits of total communication, 2 secure exponentiations requiring $4l$ bits of total communication, 2 invocations of Π_{fxpPoly} requiring $4+12l$ bits of total communication, 2 secure spline invocations on 10 intervals between $[0,1)$ requiring $84l$ bits of total communication, and one invocation of $\mathcal{F}_{\mathsf{matMult}}$ in the end to combine the results requiring $24l$ bits of total communication. Thus, the total overall communication for the secure protocol is $126l+4$ bits of communication. With the dot-product optimization mentioned in Appendix I, this cost will be reduced to $54l+4$ bits.

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

6 SECURE SIGMOID WITH DISTRIBUTED OFFLINE SETUP

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

Looking back at our construction in the previous section, we observe that the FSS preprocessing forms the bottleneck cost of securely emulating the trusted offline phase in a 2PC setting. This happens because the FSS key generation algorithm involves the usage of a PRG, and naively running the FSS key generation algorithm inside 2PC will involve the cost of computing the PRG circuit (for e.g. AES) using 2PC. This will typically [3] blow up the communication cost of the offline phase by atleast linear in the size of PRG circuit.

In this section, we will discuss an alternative approach for computing sigmoid which will enable a communication efficient offline phase while adding a mild communication overhead in the online

[3]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.

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.

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

6.1 Secure Comparison

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

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

A common approach to computing secure comparison is divide-and-conquer [17, 23], which first splits the larger input strings into smaller strings, performs comparisons on these smaller strings, and then combines the results. Rathee et al.'s CrypTFlow2 [42] efficiently instantiates the aforementioned template for performing secure comparison via a formula from Garay et. al. [23]. In our observations, we identify that the major communication cost of CrypTFlow2's online phase stems from invoking 1-out-of-2^m oblivious transfers (OTs), where m is the bit length of the smaller strings.

Another line of work based on function secret sharing (FSS) [10] performs secure comparison using a distributed comparison function (DCF). One caveat of directly using FSS to perform the entire comparison (i.e. using a single DCF for comparing two large strings) is the expensive cost of running the FSS offline phase in 2PC. While Doerner and Shelat [18] propose an elegant approach for doing FSS offline phase, their technique is efficient only for small domains. This is because it requires locally computing an exponential (in input bit length) number of PRGs. There is currently no better communication-efficient technique in the literature for conducting the FSS offline phase.

Our insight is to combine the recent approaches in secret-sharing MPC and FSS literature to achieve the best of both worlds. We begin with CrypTFlow2 [42] protocol which, as mentioned earlier, is a divide-and-conquer type protocol. Specifically, for $x = x_1||x_0$ and $y = y_1||y_0$, where $x, y \in \{0, 1\}^l$ are l bit strings that we want to compare, the following relationship holds:

$$x < y = (x_1 < y_1) \oplus [(x_1 = y_1) \wedge (x_0 < y_0)] \tag{2}$$

The key idea is to recursively apply this equation $log\ q$ times to obtain q m-bit leaves. More specifically, let $x = x_q||\ldots||x_0$ and $y = y_q||\ldots||y_0$ where x_i, y_i are m-bit strings, $q = \frac{l}{m}$ (for ease of exposition, assume m divides l and q is a power of two), and λ is a security parameter. Now, in the first phase (the divide phase), CrypTFlow2 splits the large strings x, y into q smaller strings $\{x_i, y_i\}_{i \in [0, q-1]}$ and compares x_i against y_i for each $i \in [0, q-1]$. In the second phase (the conquer phase), CrypTFlow2 combines the results of comparisons on smaller strings efficiently in a binary-tree like fashion (induced by the recursive expansion of Equation 2) to compute the final comparison $x < y$.

To perform the first phase, CrypTFlow2 uses 1-out-of-2^m OTs which add $\approx 2^m$ communication cost to the protocol for each of the small string comparisons. Our main insight is that this phase can be completely replaced with FSS-based comparison which incurs a communication cost of only $2m$ bits per small string comparison. Note that since this phase only performs comparison on small strings, we can directly use FSS gate for small domains and avoid the expensive overhead costs in the offline phase.

Our full comparison protocol is presented in Algorithm 4. See Appendix A.1 for definitions of relevant sub-functionalities.

Before proceeding further, we note that an alternative formulation of secure comparison lets the two parties hold secret shares of x and y as input (instead of private inputs) and learn a secret shared bit b representing the comparison output. We will use $\mathcal{G}_{\mathsf{CMP}}$ to denote this alternative functionality which is more relevant in the context of secret-sharing MPC, for example in our secure logistic regression use-case. As mentioned in [17], this problem can be non-interactively and black-box reduced to $\mathcal{F}_{\mathsf{CMP}}$. The observation is based on the following relationship which reduces the task of comparing two shared values x, y to the task of securely computing the MSB of $x - y$:

$$x < y = MSB(x - y) \tag{3}$$

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

$$MSB(z) = MSB(z_0) \oplus MSB(z_1) \oplus \mathbf{1}\{2^{l-1} - 1 - z_0 < z_1\} \tag{4}$$

Using $2n$ invocations of $\mathcal{G}_{\mathsf{CMP}}$ and a n invocations of $\mathcal{F}_{\mathsf{AND}}$, one can easily realize the functionality captured by MIC gate for n public intervals. In our sigmoid use-case for distributed offline setup, we will replace the MIC gate with multiple invocation of $\mathcal{G}_{\mathsf{CMP}}$ and $\mathcal{F}_{\mathsf{AND}}$. Since the number of intervals is small (exactly 6 to be specific), this replacement increases the online communication cost of our protocol by only a small amount.

6.2 Secure FSS Key Generation

As we have seen, we can reduce secure comparison to evaluation of a DCF using Algorithm 4, and using Appendix B this further reduces to an iDPF evaluation. However, in order to implement the offline phase of our protocol, we also need to generate iDPF keys efficiently. We discuss our novel approach to this in Appendix D, together with the reasons why the approaches of [10] and [18] are insufficient.

Algorithm 4: Secure comparison protocol, $\prod_{\text{CMP}}$

Input: P_0, P_1 hold $x \in \{0,1\}^l$ and $y \in \{0,1\}^l$, respectively.
Output: P_0, P_1 learn a uniform boolean sharing $\mathbf{1}\{x < y\}$.
Parameters: l, m such that $m \leq l$. Fix $q = \lceil \frac{l}{m} \rceil$
FSS preprocessing: q independent pairs of FSS keys $\{k_0^{\mathsf{eq},i}, k_1^{\mathsf{eq},i}\}_{i\in[0,q-1]}$ for $\mathcal{F}^m_{\mathsf{EQ}}$, q independent pairs of FSS keys $\{k_0^{\mathsf{lt},i}, k_1^{\mathsf{lt},i}\}_{i\in[0,q-1]}$ for $\mathcal{F}^m_{\mathsf{CMP}}$. P_0 and P_1 hold input masks $\{\mathsf{r}_0^{\mathsf{in}^{\mathsf{eq},i}}, \mathsf{r}_0^{\mathsf{in}^{\mathsf{lt},i}}\}_{i\in[0,q-1]}$ and $\{\mathsf{r}_1^{\mathsf{in}^{\mathsf{eq},i}}, \mathsf{r}_1^{\mathsf{in}^{\mathsf{lt},i}}\}_{i\in[0,q-1]}$ respectively. Both parties hold a sharing of output mask $\{[\![\mathsf{r}^{\mathsf{out}^{\mathsf{eq},i}}]\!], [\![\mathsf{r}^{\mathsf{out}^{\mathsf{lt},i}}]\!]\}_{i\in[0,q-1]}$

1 P_0 & P_1 parse their input as $x \leftarrow x_{q-1}|| \dots ||x_0$ and $y \leftarrow y_{q-1}|| \dots ||y_0$, respectively, where $x_i, y_i \in \{0,1\}^m, q = l/m$.

// Part I: Compute shares of $\mathbf{1}\{x_i = y_i\}$ and $\mathbf{1}\{x_i < y_i\}$.

2 **for** $i \in \{0, \dots, q-1\}$ **:**

3 $\quad$ P_0 sets $x_i^{\mathsf{eq}} = x_i + \mathsf{r}_0^{\mathsf{in}^{\mathsf{eq},i}}$ and $x_i^{\mathsf{lt}} = x_i + \mathsf{r}_0^{\mathsf{in}^{\mathsf{lt},i}}$. It sends $x_i^{\mathsf{eq}}, x_i^{\mathsf{lt}}$ to P_1.

4 $\quad$ P_1 sets $y_i^{\mathsf{eq}} = x_i + \mathsf{r}_1^{\mathsf{in}^{\mathsf{eq},i}}$ and $y_i^{\mathsf{lt}} = x_i + \mathsf{r}_1^{\mathsf{in}^{\mathsf{lt},i}}$. It sends $y_i^{\mathsf{eq}}, y_i^{\mathsf{lt}}$ to P_0.

5 $\quad$ P_0 & P_1 locally invoke $\mathsf{Eval}^{\mathsf{cmp}}(b, k_b^{\mathsf{lt},i}, x_i^{\mathsf{lt}}, y_i^{\mathsf{lt}})$, where b is the party id, to obtain boolean sharing $[\![\mathsf{lt}'_{0,i}]\!]$. They set $[\![\mathsf{lt}_{0,i}]\!] = [\![\mathsf{lt}'_{0,i}]\!] \oplus [\![\mathsf{r}^{\mathsf{out}^{\mathsf{lt},i}}]\!]$.

6 $\quad$ P_0 & P_1 locally invoke $\mathsf{Eval}^{\mathsf{eq}}(b, k_0^{\mathsf{eq},i}, x_i^{\mathsf{eq}}, y_i^{\mathsf{eq}})$, where b is the party id, to obtain boolean sharing $[\![\mathsf{eq}'_{0,i}]\!]$. They set $[\![\mathsf{eq}_{0,i}]\!] = [\![\mathsf{eq}'_{0,i}]\!] \oplus [\![\mathsf{r}^{\mathsf{out}^{\mathsf{eq},i}}]\!]$.

// Part II: Combine shares of $\mathbf{1}\{x_i = y_i\}$ and $\mathbf{1}\{x_i < y_i\}$.

7 **for** $i \in \{1, \dots, \log q\}$ **:**

8 $\quad$ **for** $j \in \{0, \dots, \frac{q}{2^i} - 1\}$ **:**

9 $\quad\quad$ P_0 & P_1 invoke $\mathcal{F}_{\mathsf{AND}}$ using $[\![\mathsf{lt}_{i-1,2j}]\!]$ and $[\![\mathsf{eq}_{i-1,2j+1}]\!]$ to obtain $[\![\mathsf{temp}]\!]$.

10 $\quad\quad$ P_0 & P_1 set $[\![\mathsf{lt}_{i,j}]\!] = [\![\mathsf{lt}_{i-1,2j+1}]\!] \oplus [\![\mathsf{temp}]\!]$.

11 $\quad\quad$ P_0 & P_1 invoke $\mathcal{F}_{\mathsf{AND}}$ using $[\![\mathsf{eq}_{i-1,2j}]\!]$ and $[\![\mathsf{eq}_{i-1,2j+1}]\!]$ to obtain $[\![\mathsf{eq}_{i,j}]\!]$.

12 **return** $[\![\mathsf{lt}_{\log q,0}]\!]$

6.3 Theoretical cost analysis of Secure Sigmoid with Distributed Offline Setup

We will now analyze the online communication cost for our sigmoid protocol for this distributed offline setting. As mentioned before, the difference in this sigmoid compared to the previous section is that the task of MIC gate is performed using secure comparison protocol. In our work, we set the parameter $m = 16$ in the secure comparison protocol. For 6 intervals, emulating the MIC gate using secure comparison and AND gates requires a total communication of $48l + 192$ bits. Besides this, like the previous sigmoid construction, we invoke 2 secure exponentiations requiring $4l$ bits of total communication, 2 invocations of Π_{fxpPoly} requiring $4+12l$ bits of total communication, 2 secure spline invocations on 10 intervals between $[0, 1)$ requiring $84l$ bits of total communication. One additional change is the following: Instead of using one invocation of $\mathcal{F}_{\mathsf{matMult}}$ in the end to combine the results, we now use 6 invocations of $\mathcal{F}_{\mathsf{MUX}}$ requiring $24l + 12$ bits of total communication. This modification is needed because the outputs of secure comparison are boolean shares and hence not directly compatible for a multiplication with arithmetic shared values. Thus, the total overall communication for the secure protocol is $172l + 144$ bits of communication [4] . The entire protocol without the MIC gate emulation (using secure comparison) can be performed in 3 rounds as described in Section 5.4. Our emulation of MIC gate using secure comparison protocol requires $\log(l/m) + 2$ number of online rounds. For $m = 16$ and $l \leq 64$, this requires atmost 4 rounds. These rounds can be performed in parallel with the secure exponentiation, Π_{fxpPoly} and secure spline protocols. In the end, the $\mathcal{F}_{\mathsf{MUX}}$ rquires an additional 2 rounds of communication. Thus for $l \leq 64$, the sigmoid protocol requires 6 online rounds.

7 EXPERIMENTAL EVALUATION

Implementation Details. We implemented our technique in C++ and compile our system with the Bazel build system [5]. We use the native C++ `uint64_t` for most operations. For some operations, we use `uint_128` from the Abseil library [26].

Experimental Setup. We ran our experiments on two compute-optimized c2-standard-8 Google Cloud instances with 32 GB RAM and Intel Xeon CPU at 3.1GHz clock rate. Our implementation runs on a single thread and utilizes a single core of the instance. In the LAN setting, both instances were deployed in the us-central1 region where the mean network latency was 0.15ms and the bandwidth was $\approx 1.5GB/s$. In the WAN setting, one instance was in us-central1 while the other was in us-west2. The mean network latency was 49ms and the bandwidth $50MB/s$. All runtimes are end-to-end totals for client and server, including communication, with all computation sequentialized (i.e. server and client do not compute at the same time).

Cloud costs. We include the monetary cost of running our protocols on GCP, using the prices listed at https://cloud.google.com/pricing/list. For measuring computational cost, we use the CPU spot price of $0.02 per-hour for pre-emptible virtual machines, and use network cost of $0.08 per GB for egress to the internet. All prices are in USD. This reflects the bulk batch computation setting, with parties situated in different cloud providers, as has been used in other works such as [30].

7.1 Sigmoid Experiments

Approximating the sigmoid function is the most challenging and costly component of gradient descent (see our discussion in Appendix H). For that reason, we benchmark our sigmoid protocols separately. In this section, we refer to our sigmoid protocol with trusted offline setup (Section 5) as $V1$ and our sigmoid protocol with distributed offline setup (Section 6) as $V2$. We show runtime, communication and monetary cost in Table 1, for a batch of 10^2, 10^3, and 10^4 sigmoid inputs and the following parameters: 20 fractional bits, 63-bit ring size, 31 bit-width, 10 spline intervals in $[0, 1)$, and Taylor series of degree 10.

Sigmoid Accuracy. Figure 1 (also Figure 5 in the appendix) gives visual representations of our sigmoid accuracy as compared to the Python floating point implementation of sigmoid ("true" sigmoid). We note that in Figure 1, our sigmoid nearly exactly overlaps with the "true" sigmoid. In Figure 5 in appendix, we show that our error remains tiny, on the order of 10^{-4} when using 20 fractional bits.

[4] With the dot-product optimization mentioned in Appendix I, this cost will be reduced to $100l + 144$ bits

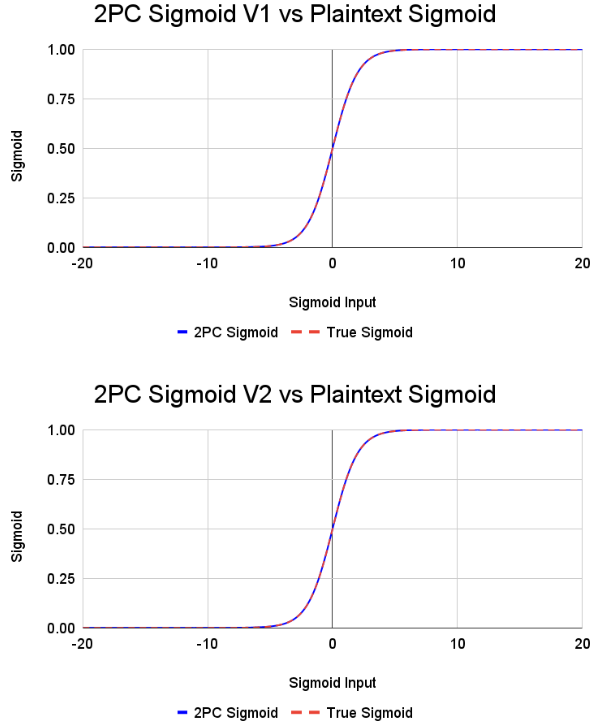


Figure 1: Our sigmoid with trusted offline setup (top) and distributed offline setup (bottom) executed in 2PC vs standard python sigmoid implementation.

Table 1: Comparison of our sigmoids with trusted (V1) and distributed (V2) offline setup to SiRnn.

Technique	Time for # Instances (sec) 10^2	10^3	10^4	Comm. per Instance (KB)	USD Cost per 10^6 runs
Sigmoid V1 (LAN)	1.89	19.19	192.23	**1.12**	**$0.192**
Sigmoid V1 (WAN)	2.27	19.95	191.97	**1.12**	**$0.192**
Sigmoid V2 (LAN)	1.61	16.27	164.47	**1.57**	**$0.211**
Sigmoid V2 (WAN)	2.28	17.49	166.427	**1.57**	**$0.212**
SiRnn (LAN)	**0.08**	**0.10**	**0.25**	4.88	$0.372

Benchmark Comparisons. We compare to the most recent secure sigmoid protocol SiRnn [41] in Table 1. In SiRnn paper, authors show that their sigmoid protocol strictly improves over other state-of-the-art sigmoid approximations such as MiniONN [36] and DeepSecure [44], hence we focus our comparison on SiRnn.

Note that our comparison is not exact. SiRnn was benchmarked on a 16-bit ring while we used 64-bit ring. Additionally, we used 20 bits of floating point precision while SiRnn used 12. SiRnn was run on a more powerful 3.7GHz processor (vs. our 3.1GHz processor), one of our network settings (50MB/s network with 49ms latency) was slower than SiRnn's network setting (377 MB/s LAN network with 0.8 ms latency). Furthermore, SiRnn only reports total time and does not divide their costs into offline/online phases (SiRnn is not really designed to be separated out into an offline-online model without nullifying the optimizations that make SiRnn efficient). We report only online time in our comparison.

Even with the differences in environment setting, the comparison is informative. We see that we have a gain of $3-4\times$ in online communication and a 30-50% lower monetary cost in a cloud environemnt. Furthermore, we have a fixed number of rounds versus an unspecified number of rounds in SiRnn. However, our computational costs are significantly higher, due to our use of relatively expensive (but still practically efficient) primitives such as DCF and DPF.

7.2 Logistic Regression Experiments

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

Table 2: Datasets we used for our experiments.

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

Datasets.

- *Titanic.* This dataset is used to predict which passengers survived the Titanic shipwreck. Survival is modeled as a function of passenger class, ticket cost, and basic demographics such as sex, age range etc. Our preprocessed dataset contains data on 714 passengers and 6 features.
- *Arcene.* This dataset contains information from the National Cancer Institute and the Eastern Virginia Medical School. The dataset is used to determine whether a patient has cancer. There are 100 patients and 10^4 features.
- *Criteo Uplift.* This dataset predicts whether a user targeted by advertising purchases a product (i.e. converts). We reduced the dataset size to 10000 data points and 15 features. The features constitute data about the user, whether the user was targeted by advertising, and if a visit occurred for the user. This dataset is highly imbalanced. From 10000 data points, there were only 47 conversions.
- *Gisette.* This dataset attempts to separate the digits 4 and 9. The digits were originally encoded in a fixed-size 28×28 image and preprocessed to yield 5000 features. Our preprocessed Gisette dataset contains 6000 digit samples.

Accuracy Evaluation. We compare accuracy of our 2PC protocols against a plaintext floating-point implementation in Table 4. We use 6 iterations of logistic regression, using parameters described above . Our 2PC protocols perform very similarly to plaintext logistic regression in all cases except Criteo Uplift, where we observe a

moderate loss in Accuracy and also F1 Score. This dataset is especially challenging for logistic regression since the distributions of labels are heavily imbalanced.

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

Comparison to previous works. There are relatively few works that focus on logistic regression training: most are focused on either inference or training of other models. The most prominent previous work discussing logistic regression is [39], which uses a relatively coarse approximation to logistic regression (see Figure 4), and has a high communication in their sigmoid implementation due to bitwise operations.

7.3 Secure Comparison Experiments

We benchmark our new comparison protocol separately. In this section, we evaluate secure comparison experimentally against Boyle et al.'s [10] FSS comparison gate. In Appendix F, we evaluate analytically against the state of the art.

Performance Comparison. We implement the FSS comparison protocol of Boyle et al. [10] and compare to our protocol $\prod_{\mathsf{CMP}}$ in Table 3. In $\prod_{\mathsf{CMP}}$, we split a 64-bit input string into four 16-bit smaller strings. Recall that our new comparison approach makes offline phase feasible with existing techniques. We achieve that while only adding a little communication in the online phase (0.06 KB per 64-bit comparison). Our experiment shows that our protocol in fact reduces total runtime from 0.39ms to 0.22ms on a low-latency LAN network.

Table 3: Comparison of our new $\prod_{\mathsf{CMP}}$ protocol to the FSS protocol [10] on a batch of 1000 inputs.

	$\prod_{\mathsf{CMP}}$	FSS Comparison [10]
LAN (sec)	**0.22**	0.39
WAN (sec)	0.56	**0.39**
Communication (KB)	63.62	**0**

8 CONCLUSION AND FUTURE WORK

In this work, we show that techniques from FSS can be combined with secret-sharing MPC to get the best of both worlds in terms of online communication cost. More specifically, we can get reduced communication in the online phase while still having an efficient offline phase. We demonstrate this idea successfully by designing a novel secure logistic regression training protocol with the best known online communication, a secure sigmoid evaluation construction with $3-4x$ communication improvement and a secure comparison protocol that reduces online communication over prior works by $\approx 2-4\times$ for appropriate parameters and similarly reduces their online communication rounds.

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

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

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

	2PC V1	2PC V2	2PC with DP V1	2PC with DP V2
	Titanic Dataset (500×6)			
LAN (sec)	57.1	54.4	57.1	54.0
WAN (sec)	61.3	59.9	61.5	59.2
Comm (MB)	3.4	4.7	3.4	4.7
Cost (USD)	0.06c	0.07c	0.06c	0.07c
	Arcene Dataset (70×10000)			
LAN (sec)	9.7	9.2	9.6	9.1
WAN (sec)	13.4	13.6	13.7	13.5
Comm (MB)	1.5	1.7	1.39	1.6
Cost (USD)	0.02c	0.02c	0.018c	0.02c
	Criteo Uplift Dataset (7000×15)			
LAN (min)	13.3	12.8	13.3	12.8
WAN (min)	13.6	12.8	13.6	12.8
Comm (MB)	46.9	65.4	46.9	65.4
Cost (USD)	0.81c	0.93c	0.81c	0.93c
	Gisette Dataset (4200×5000)			
LAN (min)	8.9	8.4	8.8	8.4
WAN (min)	9.3	8.8	9.1	8.6
Comm (MB)	28.7	39.7	28.4	39.5
Cost (USD)	0.53c	0.60c	0.52c	0.59c

While FSS-based techniques reduce online communication, they could come with more expensive offline phase. We show how to combine FSS and secret sharing techniques to reduce the input length for the FSS keys that are needed. Further, we introduce two party computation techniques for the offline FSS key generation that support arithmetic output shares that can be integrated with secret sharing computation.

Our construction can compare 128 bit numbers with 512 bits online communication and can evaluate a sigmoid function on 20-bit shared input with 1.12 KB of online communication. Training logistic regression across 6000 samples with 5000 features can be done with less than 30 MB.

REFERENCES

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.*

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

[3] S. Akinci, E. Kaynak, E. Atilgan, and Ş. Aksoy. 2007. 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] Apple and Google. 2021. Exposure Notifications Private Analytics. https://github.com/google/exposure-notifications-android/blob/master/doc/ENPA.pdf. (2021).

[5] Bazel. 2022. Bazel. https://bazel.build/. (2022).

[6] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. 2016. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. *Proc. Priv. Enhancing Technol.* (2016).

[7] 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. 2009. Secure Multiparty Computation Goes Live. In *Financial Cryptography and Data Security, 13th International Conference*, Roger Dingledine and Philippe Golle (Eds.).

[8] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2021. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 762–776.

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

[10] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2021. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *EUROCRYPT 2021, Part II (LNCS)*, Anne Canteaut and François-Xavier Standaert (Eds.), Vol. 12697. Springer, Heidelberg, 871–900. https://doi.org/10.1007/978-3-030-77886-6_30

[11] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 1292–1303. https://doi.org/10.1145/2976749.2978429

[12] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation with Preprocessing via Function Secret Sharing. In *TCC 2019, Part I (LNCS)*, Dennis Hofheinz and Alon Rosen (Eds.), Vol. 11891. Springer, Heidelberg, 341–371. https://doi.org/10.1007/978-3-030-36030-6_14

[13] Jeffrey Champion, abhi shelat, and Jonathan Ullman. 2019. Securely Sampling Biased Coins with Applications to Differential Privacy. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

[14] Rishav Chourasia, Jiayuan Ye, and Reza Shokri. 2021. Differential Privacy Dynamics of Langevin Diffusion and Noisy Gradient Descent. In *Advances in Neural Information Processing Systems*.

[15] Henry Corrigan-Gibbs. 2020. Privacy-preserving Firefox telemetry with Prio. https://rwc.iacr.org/2020/slides/Gibbs.pdf. (2020).

[16] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation, (NSDI)*. https://crypto.stanford.edu/prio/ (accessed 2020-12-09).

[17] Geoffroy Couteau. 2018. New Protocols for Secure Equality Test and Comparison. In *ACNS 18 (LNCS)*, Bart Preneel and Frederik Vercauteren (Eds.), Vol. 10892. Springer, Heidelberg, 303–320. https://doi.org/10.1007/978-3-319-93387-0_16

[18] Jack Doerner and abhi shelat. 2017. Scaling ORAM for Secure Computation. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 523–535. https://doi.org/10.1145/3133956.3133967

[19] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. 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)*.

[20] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography*, Shai Halevi and Tal Rabin (Eds.).

[21] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Found. Trends Priv. Secur.* (2018).

[22] David A. Freedman. 2009. *Statistical Models: Theory and Practice* (2 ed.). Cambridge University Press.

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

[24] Alexandre Gilotte. 2021. Results from the Criteo-AdKDD-2021 Challenge. https://medium.com/criteo-engineering/results-from-the-criteo-adkdd-2021-challenge-50abc9fa3a6. (2021).

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

[26] Google. 2022. Abseil - C++ Common Libraries. https://github.com/abseil/abseil-cpp. (2022).

[27] Google. 2022. https://developer.chrome.com/docs/privacy-sandbox/attribution-reporting/. https://developer.chrome.com/docs/privacy-sandbox/attribution-reporting/. (2022).

[28] Google. 2022. Privacy Sandbox. https://privacysandbox.com/intl/en_us/. (2022).

[29] Charlie Harrison, Mariana Raykova, Michael Kleeber, John Delaney, and Andres Munoz Medina. 2020. Multi-Browser Aggregation Service Explainer. https://github.com/WICG/conversion-measurement-api/blob/main/SERVICE.md. (2020).

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

[31] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. 2018. Distributed Learning without Distress: Privacy-Preserving Empirical Risk Minimization. In *Advances in Neural Information Processing Systems*.

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

[33] Mahimna Kelkar, Phi Hung Le, Mariana Raykova, and Karn Seth. 2022. Secure Poisson Regression. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/kelkar

[34] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.

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

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

[37] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. 2019. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and its Application to Anonymous Communication. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 887–903. https://doi.org/10.1145/3319535.3354238

[38] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 35–52.

[39] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 19–38. https://doi.org/10.1109/SP.2017.12

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

[41] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. SiRnn: A Math Library for Secure RNN Inference. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1003–1020. https://doi.org/10.1109/SP40001.2021.00086

[42] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 325–342. https://doi.org/10.1145/3372297.3417274

[43] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 325–342.

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

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

[46] Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas. 2019. Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 1335–1350. https://doi.org/10.1145/3319535.3339816

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

[48] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit. (2016).

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

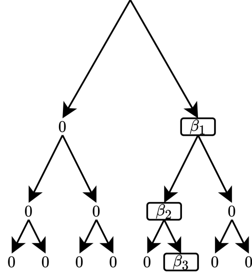


Figure 2: Incremental DPF gives compact secret sharing of values on the nodes of a binary tree with a single non-zero path. In this example, $\alpha = 101$ and the values on the path to the leaf at index α are $\beta_1, \beta_2, \beta_3$. All other nodes are 0. This figure shows the reconstructed secret shares $\mathsf{Eval}(0, k_0, \cdot) \oplus \mathsf{Eval}(1, k_1, \cdot)$. The keys are generated as $(k_0, k_1) \leftarrow \mathsf{Gen}^{\mathsf{idpf}}(\alpha, \beta_1, \beta_2, \beta_3)$.

[50] Jiayuan Ye and Reza Shokri. 2022. Differentially Private Learning Needs Hidden State (Or Much Faster Convergence). *CoRR* abs/2203.05363 (2022).

A ADDITIONAL PRELIMINARIES

Incremental Distributed Point Functions. Introduced by Boneh et al. [8], 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 $l \in [n]$:

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

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

We sometimes allow an iDPF to be evaluated over the empty prefix. We now present iDPF formally, see Figure 2 for more intuition. We closely follow the definitions of Boneh et al. [8], with on major difference being that we expose the EvalNext function as part of our definition. We will use this in our reduction from distributed comparison functions to iDPFs.

Definition A.1. A 2-party iDPF scheme is a tuple of three algorithms $(\mathsf{Gen}^{\mathsf{idpf}}, \mathsf{EvalNext}^{\mathsf{idpf}}, \mathsf{EvalPrefix}^{\mathsf{idpf}})$ such that:

- $\mathsf{Gen}^{\mathsf{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n)))$ is a PPT *key generation* algorithm that given security parameter 1^λ and a function description $(\alpha, (\mathbb{G}_1, \beta_1), \ldots, (\mathbb{G}_n, \beta_n))$, outputs a pair of keys and public parameters $(k_0, k_1, \mathsf{pp} = (\mathsf{pp}_1, \ldots, \mathsf{pp}_n))$.

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

  Intuitively, EvalNext represents the evaluation on some partial value $x \in \{0,1\}^{l-1}$ and outputs a secret sharing y_b^l of the value on the $x||x_l$th node of the binary tree and an updated state st_b^l.
- $\mathsf{EvalPrefix}^{\mathsf{idpf}}(b, k_b, \mathsf{pp}, (x_1, \ldots, x_l))$ is a polynomial time *prefix evaluation* algorithm that given a party id $b \in \{0,1\}$, iDPF key k_b, public parameters pp, and input prefix $(x_1, \ldots, x_l) \in \{0,1\}^l$, outputs an additive secret sharing of the output value y_b^l.

Next, we present iDPF correctness and security.

Definition A.2. (Gen, EvalNext, EvalPrefix) from Definition A.1 is an iDPF scheme if it satisfies the following requirements:

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

A naive approach to constructing iDPF would be to generate one DPF key for each prefix length, i.e. a total of n independent keys. Then, evaluate $x \in \{0,1\}^l$ with the lth key. This solution would yield key size *quadratic* in the input length n. [8] gives a more direct construction with key size linear in n.

Theorem A.3 (Concrete cost of iDPF [8]). *Given a PRG $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda+2}$, 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 $\approx n(\lambda + n)$ [10].

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 B.

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

A.1 Miscellaneous 2PC functionalities

A.1.1 Oblivious Transfer. We will use $\mathcal{F}^k_{\mathsf{OT}}$ to denote the two-party 1-out-of-2 chosen input oblivious transfer functionality, where the sender's input to $\mathcal{F}_{\mathsf{OT}}$ are two strings $m_0, m_1 \in \{0,1\}^k$ and the receiver's input is a choice bit $c \in \{0,1\}$. The receiver obtains m_c as output from $\mathcal{F}^k_{\mathsf{OT}}$ whereas the sender has no output. In the pre-processing model, one can generate random OT (ROT) correlation which consists of (r_0, r_1, b) where $r_0 \in \{0,1\}^k, r_1 \in \{0,1\}^k, b \in \{0,1\}$ and distribute this across the two-parties. The sender gets (r_0, r_1) whereas the receiver gets (b, r_b). In the online phase, these ROT correlations can be cheaply consumed to realize $\mathcal{F}^k_{\mathsf{OT}}$ efficiently.

A.1.2 Multiplexer. Following prior work [42], we will use the $\mathcal{F}_{\mathsf{MUX}}$ to denote a multiplexer functionality. Suppose parties hold arithmetic shares of x and boolean sharing of a selection bit b. Then they can use $\mathcal{F}_{\mathsf{MUX}}$ to get an arithmetic sharing of x if $b = 1$, and arithmetic sharing of 0 otherwise. As mentioned in prior work [42], a protocol for $\mathcal{F}_{\mathsf{MUX}}$ can be realized using 2 (simulataneous) calls to $\mathcal{F}_{\mathsf{OT}}$. In some scenarios, a variant of $\mathcal{F}_{\mathsf{MUX}}$, denoted by $\mathcal{F}_{\mathsf{MUX2}}$, might be more useful. It takes as input arithmetic shares of x_0 and x_1, along with boolean sharing of a selection bit b. Then it outputs a fresh sharing of x_0 if $b = 0$, and a fresh arithmetic sharing of x_1 otherwise. A protocol for $\mathcal{F}_{\mathsf{MUX2}}$ can be realized using a single call to $\mathcal{F}_{\mathsf{MUX}}$ in the following way: Parties locally compute a sharing of $x_1 - x_0$, invoke $\mathcal{F}_{\mathsf{MUX}}$ on it using the share of bit b, and finally locally add the sharing of x_0 to their output from $\mathcal{F}_{\mathsf{MUX}}$.

A.1.3 Functionalities based on FSS. We will now describe some of the functionalities which can be realized efficiently using Function Secret Sharing primitive. We note that gates based on FSS operate on masked inputs and produce masked outputs (instead of standard secret-sharing MPC gates which operates on input shares and produce output shares). Specifically, a masked value x_{mask} for a secret input x is computed as $x_{\mathsf{mask}} := x + r$, where r is a uniform random element from the same domain as x. The mask r is sampled during an offline phase and is used in constructing the pre-processing material for FSS based gates. As described in [10], 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:

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

Boyle et. al. [12] 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 5.

Algorithm 5: 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.

$\mathsf{Gen}^{\mathsf{eq}}_n(1^\lambda, \mathsf{r}^{\mathsf{in}}_0, \mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{out}})$:

1 Let $\mathsf{r}^{\mathsf{in}}_0 \in \mathbb{G}_1$ and $\mathsf{r}^{\mathsf{in}}_1 \in \mathbb{G}_2$.
2 Let $\alpha \leftarrow \mathsf{r}^{\mathsf{in}}_0 - \mathsf{r}^{\mathsf{in}}_1$, $\beta = 1$.
3 $k'_0, k'_1 \leftarrow \mathsf{Gen}^{\mathsf{DPF}}(1^\lambda, \alpha, \beta)$
4 Sample random additive shares $\mathsf{r}^{\mathsf{out}}_0, \mathsf{r}^{\mathsf{out}}_1 \leftarrow [\![\mathsf{r}^{\mathsf{out}}]\!]$.
5 Let $k_b = k'_b || \mathsf{r}^{\mathsf{out}}_{\mathsf{b}}$.
6 **return** (k_0, k_1)

// Part II: Online Phase.

$\mathsf{Eval}^{\mathsf{eq}}_n(b, k_b, x_{\mathsf{mask}}, y_{\mathsf{mask}})$:

7 Parse $k_b = k'_b || \mathsf{r}^{\mathsf{out}}_{\mathsf{b}}$.
8 **return** $\mathsf{Eval}^{\mathsf{DPF}}(b, k'_b, x_{\mathsf{mask}} - y_{\mathsf{mask}}) + \mathsf{r}^{\mathsf{out}}_{\mathsf{b}}$

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. [10] in Algorithm 6. 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 [10]'s $x > y$ and (2) the output group is $\mathbb{U}_2$ instead of $\mathbb{U}_N$.

Algorithm 6: FSS Gate for $\mathcal{F}^{n}_{\mathsf{CMP}}$

Input: $\mathsf{P}_0, \mathsf{P}_1$ 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: $\mathsf{P}_0, \mathsf{P}_1$ learn a uniform boolean sharing $b_{\mathsf{mask}} = b \oplus \mathsf{r}^{\mathsf{out}}$, where $b := \mathbf{1}\{x < y\}$.

// Part I: Offline Phase.

$\mathsf{Gen}^{\mathsf{cmp}}_{n}(1^{\lambda}, \mathsf{r}^{\mathsf{in}}_0, \mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{out}})$:

1. Let $y = (2^n - (\mathsf{r}^{\mathsf{in}}_0 - \mathsf{r}^{\mathsf{in}}_1)) \in \mathbb{U}_N$ and $\alpha^{(n-1)} = y_{[0,n-1)}$.
2. $(k^{(n-1)}_0, k^{(n-1)}_1) \leftarrow \mathsf{Gen}^{\mathsf{DDCF}}_{n-1}(1^{\lambda}, \alpha^{(n-1)}, \beta_1, \beta_2, \mathbb{U}_2)$, where $\beta_1 = 1 \oplus y_{[n-1]}, \beta_2 = y_{[n-1]} \in \mathbb{U}_2$.
3. Sample random $\mathsf{r}^{\mathsf{out}}_0, \mathsf{r}^{\mathsf{out}}_1 \leftarrow \mathbb{U}_N$ s.t. $\mathsf{r}^{\mathsf{out}}_0 \oplus \mathsf{r}^{\mathsf{out}}_1 = \mathsf{r}^{\mathsf{out}}$.
4. For $b \in \{0,1\}$, let $k_b = k^{(n-1)}_b || \mathsf{r}^{\mathsf{out}}_{\mathsf{b}}$.
5. **return** (k_0, k_1)

// Part II: Online Phase.

$\mathsf{Eval}^{\mathsf{cmp}}_{n}(b, k_b, x_{\mathsf{mask}}, y_{\mathsf{mask}})$:

6. Parse $k_b = k^{(n-1)}_b || \mathsf{r}^{\mathsf{out}}_{\mathsf{b}}$.
7. Set $z = (x_{\mathsf{mask}} - y_{\mathsf{mask}}) \in \mathbb{U}_N$.
8. Set $m^{(n-1)}_b \leftarrow \mathsf{Eval}^{\mathsf{DDCF}}_{n-1}(b, k^{(n-1)}_b, z^{(n-1)})$, where $z^{(n-1)} = 2^{n-1} - z_{[0,n-1)} - 1$.
9. **return** $b \cdot z_{[n-1]} + m^{(n-1)}_b - 2 \cdot z_{[n-1]} \cdot m^{(n-1)}_b + \mathsf{r}^{\mathsf{out}}_{\mathsf{b}}$

Algorithm 7: FSS Gate for $\mathcal{F}_{\mathsf{MIC}}$

Input: $\mathsf{P}_0, \mathsf{P}_1$ 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: $\mathsf{P}_0, \mathsf{P}_1$ learn a uniform arithmetic sharing of $b_{i\,\mathsf{mask}} = b_i + \mathsf{r}^{\mathsf{out}}_{\mathsf{i}}$, where $b_i := \mathbf{1}\{p_i \le x \le q_i\}$.

// Part I: Offline Phase.

$\mathsf{Gen}^{\mathsf{mic}}_{n,m,\{p_i,q_i\}_i}(1^{\lambda}, \mathsf{r}^{\mathsf{in}}, \{\mathsf{r}^{\mathsf{out}}_{\mathsf{i}}\}_{i \in [m]})$:

1. Let $\gamma = (N-1) + \mathsf{r}^{\mathsf{in}}$
2. $(k^{(N-1)}_0, k^{(N-1)}_1) \leftarrow \mathsf{Gen}^{\mathsf{DCF}}_{n}(1^{\lambda}, \gamma, 1, \mathbb{U}_N)$
3. **for** $i = 1$ *to* m **:**
4. Set $q'_i = q_i + 1$, $\alpha^{(p)}_i = p_i + \mathsf{r}^{\mathsf{in}}$, $\alpha^{(q)}_i = q_i + \mathsf{r}^{\mathsf{in}}$, $\alpha^{(q')}_i = q_i + 1 + \mathsf{r}^{\mathsf{in}}$.
5. Sample random $z_{i,0}, z_{i,1} \leftarrow \mathbb{U}_N$ such that: $z_{i,0} + z_{i,1} = \mathsf{r}^{\mathsf{out}} + \mathbf{1}\{\alpha^{(p)}_i > \alpha^{(q)}_i\} - \mathbf{1}\{\alpha^{(p)}_i > p_i\} + \mathbf{1}\{\alpha^{(q')}_i > q'_i\} + \mathbf{1}\{\alpha^{(q)}_i = N-1\}$
6. For $b \in \{0,1\}$, let $k_b = k^{(N-1)}_b || \{z_{i,b}\}_i$
7. **return** (k_0, k_1)

// Part II: Online Phase.

$\mathsf{Eval}^{\mathsf{mic}}_{n,m,,\{p_i,q_i\}_i}(b, k_b, x_{\mathsf{mask}})$:

8. Parse $k_b = k^{(N-1)}_b || \{z_{i,b}\}_i$.
9. **for** $i = 1$ *to* m **:**
10. Set $q'_i = q_i + 1 \bmod N$.
11. Set $x^{(p)}_i = x + (N - 1 - p_i)$ and $x^{(q')}_i = x + (N - 1 - q'_i)$.
12. Set $s^{(p)}_{i,b} \leftarrow \mathsf{Eval}^{\mathsf{DCF}}_{n}(b, k^{(N-1)}_b, x^{(p)}_i)$.
13. Set $s^{(q')}_{i,b} \leftarrow \mathsf{Eval}^{\mathsf{DCF}}_{n}(b, k^{(N-1)}_b, x^{(q')}_i)$.
14. $y_{i,b} = b \cdot (\mathbf{1}\{x_{\mathsf{mask}} > p_i\} - \mathbf{1}\{x_{\mathsf{mask}} > q'_i\} - s^{(p)}_{i,b} + s^{(q')}_{i,b} + z_{i,b})$.
15. **return** $\{y_{i,b}\}_i$

Multiple Interval Containment (MIC) gate. Boyle et. al. [10] 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 $b_i = \mathbf{1}\{p_i \le x \le q_i\}$.

B 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 $f^{<}_{\alpha_1 \ldots \alpha_{n-1},\beta}$ 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) $x_1, \ldots, x_{n-1} \ne \alpha_1, \ldots, \alpha_{n-1}$. Then no matter what α_n and x_n are, $v_n = v_{n-1}$.
(2) $x_1, \ldots, x_{n-1} = \alpha_1, \ldots, \alpha_{n-1}$, and $\alpha_n = 0$. Then no matter what x_n is, $x \ge \alpha$, and so $v_n = v_{n-1} = 0$.
(3) $x_1, \ldots, x_{n-1} = \alpha_1, \ldots, \alpha_{n-1}$, and $\alpha_n = 1, x_n = 1$. Then $x = \alpha$ and therefore $v_n = v_{n-1} = 0$.
(4) $x_1, \ldots, x_{n-1} = \alpha_1, \ldots, \alpha_{n-1}$, and $\alpha_n = 1, x_n = 0$. Then $v_{n-1} = 0$, but $v_n = \beta$.

Observe that only in the last case, $v_n \ne v_{n-1}$, 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 $x_1, \ldots, x_{n-1}$, 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 8 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 $i = 2, \ldots, n$, 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 B.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.*

Proof B.1 (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.3 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.3 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.*

Algorithm 8: 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** $((k_0, [\![\beta_1]\!]^0), (k_1, [\![\beta_1]\!]^1), \mathsf{pp})$

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

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}

Comparison with original DCF construction. Boyle et al. [10] presented a direct construction of DCF by carefully modifying and making non black-box changes to a prior DPF construction [11]. 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. [10] 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.

C SECURE COMPARISON PROTOCOL

C.1 Extensions and Optimizations

We will now discuss some extensions and optimizations for Protocol $\prod_{\mathsf{CMP}}$ which are borrowed from CrypTFlow2 [42].

Note that as written, Protocol $\prod_{\mathsf{CMP}}$ only works when $q = \left\lceil \frac{l}{m} \right\rceil$ is a power of 2. If that is not the case, then the induced recursion tree is not a perfect binary tree. However, this case can be handled by creating maximal possible perfect binary trees and connecting the roots of the same using the Equation 2 as described in Cryptflow2 [42].

As observed in CrypTFlow2 [42], the 2 calls to $\mathcal{F}_{\mathsf{AND}}$ in steps 9 and 11 have a common input $[\![\mathsf{eq}_{i-1,2j+1}]\!]$. This fact can be leveraged by using a pair of correlated-bit triple for realizing the two $\mathcal{F}_{\mathsf{AND}}$ calls. A pair of correlated bit-triple is of the form $([\![a_1]\!], [\![b]\!], [\![c_1]\!])$, $([\![a_2]\!], [\![b]\!], [\![c_2]\!])$, for $a_1, a_2, b \in_R \{0,1\}$, where $a_1 \wedge b = c_1$ and $a_2 \wedge b = c_2$. Such correlated-bit triple enable two calls to $\mathcal{F}_{\mathsf{AND}}$ using just 3 elements of communication (per party) whereas standard triple requires 4 elements of communication (per party). Such a correlated bit triple can be efficiently generated using 1 invocation of $\binom{8}{1}\mathsf{OT}_2$ which will cost $2\lambda + 16$ bits of communication in the offline phase (by leveraging OT extension techniques). In the online phase, consuming a correlated bit triple requires 6 bits of total communication to compute both $\mathcal{F}_{\mathsf{AND}}$ calls.

C.2 Cost Analysis

We will now analyze the communication cost, rounds and computation complexity of our secure comparison protocol $\prod_{\mathsf{CMP}}$. To do so, we will separately estimate the cost of the online phase and the offline phase.

- Online phase
  - Communication : Line 5 and 6 require invocation of an FSS gate for comparison and equality, respectively, on m bit strings. Each invocation of an FSS gate requires communication of m bits per party. Since Line 5 and 6 are executed for a total of $q = l/m$ times, the total communication cost of Part I is $4l$. The communication cost for Part II, after accounting for the optimizations described in CrypTFlow2 [42], is exactly $6(q-1) - 2\log q$. This brings the total online communication cost of our protocol to be $4l + 6q - 6 - 2\log q$ bits.
  - Rounds : We note that all q invocations of Line 5 and 6 can be performed in parallel. Since FSS gates only require 1 round of interaction, Part I only costs 1 round. In Part II, all invocations within the second loop can be parallelized however the first loop needs to be sequentially computed. Therefore, Part II ends up costing $\log q$ rounds. In total, $\prod_{\mathsf{CMP}}$ requires $\log q + 1$ rounds.
  - Computation : The computation cost is mainly dominated by Line 5 and 6 which require evaluation of FSS gate for comparison and equality testing. Each evalutation of FSS gate on m bit inputs requires m PRG evaluations. This results in a total of $2l$ PRG evaluations per FSS gate per party.
- Offline phase: The cost of the offline phase depends on the cost of generating FSS keys (for later use in Part I) and bit-triples (for later use in Part II). We will analyze it in the following points.
  - Communication : For Part II, we use the estimates from CrypTFlow2 [42] which depend on the cost of generating $q-1-\log q$ correlated bit-triple and $\log q$ standard bit-triple. By leveraging OT extension techniques, this can be generated using a total communication of $(2\lambda+16)(q-1) - \lambda \log q$ bits (excluding the cost of base OTs). For estimating the offline cost corresponding to Part I, we appeal to the Doerner-Shelat technique [18] which we describe in Appendix D. As per our estimate, each DCF and DPF generation will require a communication cost of $m(12\lambda + 10)$ and $2 + m(12\lambda + 8)$ bits of communication respectively. Since we are using q DCFs and DPFs in total, the communication cost will become $24l\lambda + 18l + 2q$. Adding this with the cost for generating bit-triples, we get a total offline communication cost of $\lambda(24l + 2q - \log q) + 18(l+q) - 2\lambda - 16$.
  - Rounds : The round complexity of offline phase is dominated by the cost of generating DCF keys using the Doerner-Shelat technique which requires 5 rounds per level of DCF tree. Since there are m levels in the tree, the total round complexity comes out to be $5m$.

- Computation : The computation cost of offline phase is dominated by the local PRG evaluations needed for generating DCF and DPF keys. The Doerner-Shelat technique (see Section 6.2) requires a PRG evaluation for each node in the DCF/DPF tree. Since there are $2^{m+1} - 1$ nodes in the tree and we need q DCFs and DPFs, the total computation cost is approximately equivalent to the cost of performing $2q(2^{m+1} - 1)$ PRG evaluations.

D EFFICIENT 2PC GENERATION OF FSS KEYS

As we have seen, we can reduce secure comparison to evaluation of a DCF using Algorithm 4, and using Appendix B this further reduces to an iDPF evaluation. However, in order to implement the offline phase of our protocol, we also need to generate iDPF keys efficiently.

A straight-forward way to generate these keys in MPC is to implement $\mathsf{Gen}^{\mathsf{iDPF}}$ using a generic MPC compiler. This, however, has the drawback of requiring PRG calls inside the MPC, making this approach inefficient in practice. Doerner and shelat [18] present a construction that does not require secure PRG evaluations. While, it comes at a computation cost that is linear in the domain size (i.e., exponential in the input size), and its round complexity is linear in the input size, it is still efficient enough in our case, where the domain for any single DCF is small.

However, the original Doerner-shelat construction is not sufficient to obtain DPF 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.

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 [10] 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 [10]. 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 9. 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 8 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), $[\![W^0_{CW}]\!] = [\![W^1_{CW}]\!]$ in Step 11, and so the last $\mathcal{F}_{\mathsf{MUX2}}$ call can be saved in that case, making the entire second MPC linear.

E 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. [31] for the setting of logistic regression presented in Algorithm 10. Jayaraman et. al. [31] 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 11 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 of DP noise were presented by Dwork at al. [19] and Champion et al. [13].

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 11. 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 [14, 50] show that keeping the DP-SGD

Algorithm 9: Secure Distributed Gen^{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 Convert : $\{0,1\}^\lambda \to \{0,1\}^{\lambda+1}$ be PRGs.

Each party P_b performs the following:

1 Sample $s_b^\emptyset \in \{0,1\}^\lambda$, set $t_b^\emptyset = b$.
2 **for** $l = 1$ *to* n **:**
3 For all $w \in \{0,1\}^{l-1}$, compute $s_b^{w,L}||t_b^{w,L}||s_b^{w,R}||t_b^{w,R} = G(s_b^w)$.
4 Compute $s_b^L||t_b^L||s_b^R||t_b^R = \bigoplus_{w\in\{0,1\}^{l-1}} s_b^{w,L}||t_b^{w,L}||s_b^{w,R}||t_b^{w,R}$.
5 **Secure Computation**:
- Inputs: Boolean sharing of α_l, arithmetic sharing of $\{s_b^L, s_b^R, t_b^L, t_b^R\}_{b\in\{0,1\}}$.
- Compute:

$$\llbracket s^R \rrbracket \leftarrow \llbracket s_0^R \rrbracket \oplus \llbracket s_1^R \rrbracket$$
$$\llbracket s^L \rrbracket \leftarrow \llbracket s_0^L \rrbracket \oplus \llbracket s_1^L \rrbracket$$
$$\llbracket s_{CW} \rrbracket \leftarrow \mathcal{F}_{\mathsf{MUX2}}\Big(\llbracket s^R \rrbracket, \llbracket s^L \rrbracket, \llbracket \alpha_l \rrbracket\Big)$$
$$\llbracket t_{CW}^L \rrbracket \leftarrow \llbracket t_0^L \rrbracket \oplus \llbracket t_1^L \rrbracket \oplus \llbracket \alpha_l \rrbracket \oplus \llbracket 1 \rrbracket$$
$$\llbracket t_{CW}^R \rrbracket \leftarrow \llbracket t_0^R \rrbracket \oplus \llbracket t_1^R \rrbracket \oplus \llbracket \alpha_l \rrbracket$$

- Output $s_{CW}, t_{CW}^L, t_{CW}^R$ to both
6 For all $w \in \{0,1\}^{l-1}$, set $\tilde{s}_b^{w||0}||\tilde{s}_b^{w||1} \leftarrow (s_b^{w,L}||s_b^{w,R}) \oplus t_b^w \cdot (s^{CW}||s^{CW})$
7 For all $w \in \{0,1\}^{l-1}$, set $t_b^{w||0}||t_b^{w||1} \leftarrow (t_b^{w,L}||t_b^{w,R}) \oplus t_b^w \cdot (t_{CW}^L||t_{CW}^R)$
8 For all $w \in \{0,1\}^l$, set $s_b^w||W_b^w \leftarrow \text{Convert}(\tilde{s}_b^w)$
9 Compute $W_b^l \leftarrow \sum_{w\in\{0,1\}^l} W_b^w$.
10 Compute $T_b^l \leftarrow b + (-1)^b \cdot \sum_{w\in\{0,1\}^l} t_b^w$.
Let τ_b^0 and τ_b^1 denote the two least significant bits of T_b.
11 **Secure Computation**:
- Inputs: Arithmetic sharing of β_l, private inputs $W_b^l, \tau_b^0, \tau_b^1$ for Party P_b.
- Compute:

$$\llbracket t^* \rrbracket \leftarrow 1 \oplus \tau_0^1 \oplus \tau_1^1 \oplus (\tau_0^0 \cdot \tau_1^0)$$
$$\llbracket W_{CW}^0 \rrbracket \leftarrow \llbracket \beta_l \rrbracket - W_0^l + W_1^l$$
$$\llbracket W_{CW}^1 \rrbracket \leftarrow -\llbracket \beta_l \rrbracket + W_0^l - W_1^l$$
$$\llbracket W_{CW} \rrbracket \leftarrow \mathcal{F}_{\mathsf{MUX2}}\Big(\llbracket W_{CW}^0 \rrbracket, \llbracket W_{CW}^1 \rrbracket, \llbracket t^* \rrbracket\Big).$$

- Output W_{CW} to both
12 Set $CW^l \leftarrow s_{CW}||t_{CW}^L||t_{CW}^R||W_{CW}$
13 Output $k_b \leftarrow s_b^\emptyset||CW^1||\ldots||CW^n$

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 10: DP SGD

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

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

1 Let $\mathbf{w_0}$ be the initial model with arbitrary weights
2 **for** $t = 1$ *to* T **:**
3 Compute gradient $\mathbf{g_t} \leftarrow \frac{1}{B}\mathbf{X}_B^T \times (\mathsf{Sigmoid}(\mathbf{X}_B \times \mathbf{w_{t-1}}) - \mathbf{Y_B})$
4 Perturb gradient $\widetilde{\mathbf{g_t}} \leftarrow \mathbf{g_t} + \mathcal{N}(0, \sigma^2 I_p)$ where $\sigma^2 = \frac{8G^2 T \log(1/\delta)}{n^2\epsilon^2}$
5 Update model $\mathbf{w_t} \leftarrow \mathbf{w_{t-1}} - \alpha \cdot \widetilde{\mathbf{g_t}}$
6 **return** $\mathbf{w}_T$

Algorithm 11: DP-SGD Logistic Regression Protocol

Public inputs: Number of iterations T, dataset dimensions n, k, batch size B, Lipschitz value $G = 1$, smoothness value $L = 0.25$, learning rate $\alpha = 1/L$, DP parameters ϵ and δ, regularization parameter λ.

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

1 Let $\mathbf{w_0}$ be the initial model with arbitrary weights.
2 **for** $t = 1$ *to* T **:**
3 **for** $b = 1$ *to* $\lfloor n/B \rfloor$ **:**
4 $i \leftarrow (b-1)\cdot B + 1$
5 $j \leftarrow \min(n, b\cdot B)$
6 $\llbracket X_B \rrbracket \leftarrow \llbracket X_{i\ldots j} \rrbracket$
7 $\llbracket \mathbf{u} \rrbracket \leftarrow \llbracket X_B \rrbracket \cdot \mathbf{w_{t-1}}$
8 $\llbracket \mathbf{s} \rrbracket \leftarrow \mathcal{F}_{\mathsf{Sigmoid}}(\mathbf{u})$
9 $\llbracket \mathbf{d} \rrbracket \leftarrow \llbracket \mathbf{s} \rrbracket - \llbracket \mathbf{y}_{i\ldots j} \rrbracket$
10 $\llbracket \mathbf{g} \rrbracket \leftarrow \mathcal{F}_{\mathsf{matMult}}(\llbracket X_B{}^\top \rrbracket, \llbracket \mathbf{d} \rrbracket)$
11 $\llbracket \mathbf{w_t} \rrbracket \leftarrow \llbracket \mathbf{w_{t-1}} \rrbracket - (\alpha/B)\cdot(\llbracket \mathbf{g} \rrbracket + \lambda \cdot \llbracket \mathbf{w_{t-1}} \rrbracket) + \llbracket r_t \rrbracket$
12 $\mathbf{w_t} \leftarrow \mathsf{Reconstruct}(\llbracket \mathbf{w_t} \rrbracket)$
13 **return** $\mathbf{w_T}$.

Jayaraman et. al. [31] 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 [31], 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 11.

F COMPARISON PROTOCOL MICRO-BENCHMARKS

In this section, we benchmark the offline and online communication costs of our new comparison protocol $\prod_{\mathsf{CMP}}$ for different values of l (the bit length of the comparison inputs) and m (into which the l-bit

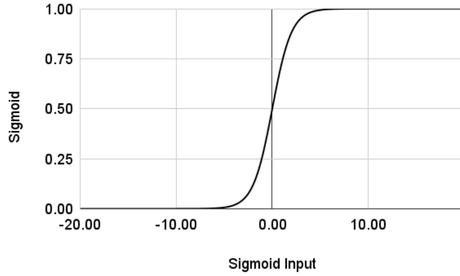


Figure 3: Plaintext Sigmoid Function

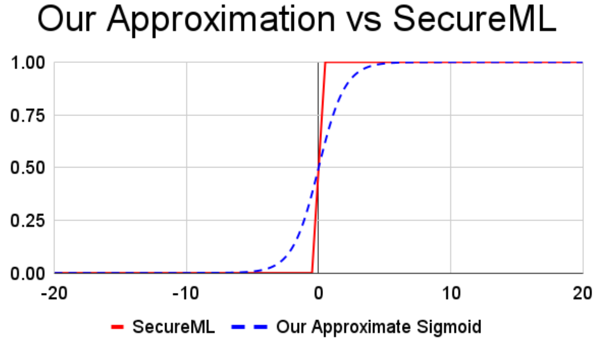


Figure 4: Plaintext Comparison of V1 sigmoid to SecureML's approximation.

inputs are split) in Table 7, and compare it against Cryptflow2 [42] in Table 8. Note that the offline costs in Table 8 for $m \geq 16$ are empty as the paper does not specify the appropriate OT extension costs for this parameter regime. We present a comprehensive comparison to state of the art in Table 6. We choose appropriate parameters for the block length m for each work. Note that if $l \leq m$, then we set $m = l$. Our offline costs exclude the cost of base oblivious transfers.

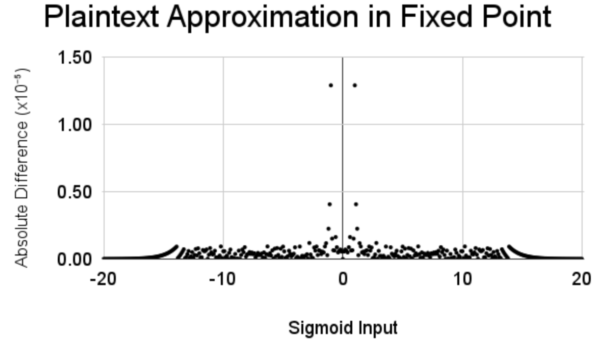
G ADDITIONAL FIGURES

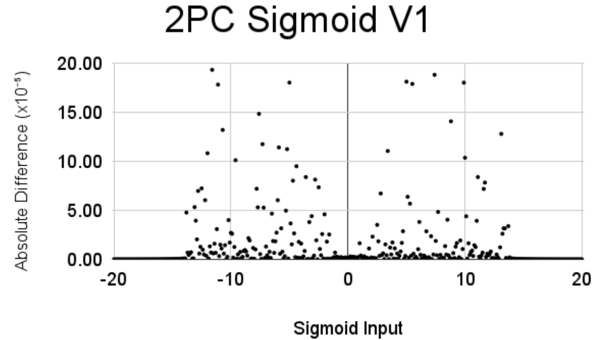
In this section, we show plaintext sigmoid in Figure 3 and demonstrate how our plaintext sigmoid approximation compares to the plaintext approximation in SecureML in Figure 4. In Figure 5, we measure the absolute error (defined w.r.t. plaintext python implementation shown in Figure 3) for three different implementations: plaintext sigmoid approximation in fixed point (top), 2PC sigmoid with trusted offline setup (middle), and 2PC sigmoid with distributed offline setup (bottom). We do this experiment for input values in the range $[-20, 20]$ at increments of 0.1.

H BOTTLENECK COST OF SECURE LOGISTIC REGRESSION

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

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





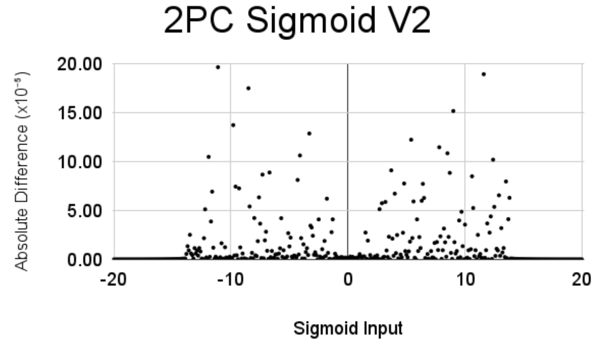


Figure 5: Absolute difference error ($\times 10^{-5}$) of our sigmoid approximation in plaintext fixed point representation (top), with trusted offline setup run in 2PC (middle), and with distributed offline setup run in 2PC (bottom).

Table 6: Concrete communication and round costs of our comparison protocol vs. prior works as functions of l.

	Our Approach (m = 16)		CrypTFlow2 (m = 4) [42]		CrypTFlow2 (m = 7) [42]		Couteau16 [17]	
l	comm.	rounds	comm.	rounds	comm.	rounds	comm.	rounds
				Offline Phase				
4	1.51 KB	80 rounds	0.03 KB	1 round	**0.03 KB**	**1 round**	0.19 KB	2 rounds
8	3.02 KB	80 rounds	0.08 KB	1 round	**0.08 KB**	**1 round**	0.44 KB	2 rounds
16	6.04 KB	80 rounds	0.19 KB	1 round	**0.13 KB**	**1 round**	1.02 KB	2 rounds
32	12.09 KB	80 rounds	0.43 KB	1 round	**0.24 KB**	**1 round**	1.85 KB	3 rounds
64	24.21 KB	80 rounds	0.93 KB	1 round	**0.55 KB**	**1 round**	3.83 KB	3 rounds
128	48.47 KB	80 rounds	1.95 KB	1 round	**1.11 KB**	**1 round**	6.36 KB	3 rounds
				Online Phase				
4	**8 bits**	**1 round**	20 bits	2 rounds	20 bits	2 rounds	30 bits	2 rounds
8	**16 bits**	**1 round**	60 bits	3 rounds	144 bits	3 rounds	162 bits	6 rounds
16	**32 bits**	**1 round**	142 bits	4 rounds	416 bits	4 rounds	308 bits	6 rounds
32	**132 bits**	**2 rounds**	308 bits	5 rounds	978 bits	5 rounds	530 bits	12 rounds
64	**270 bits**	**3 rounds**	642 bits	6 rounds	2290 bits	6 rounds	1120 bits	12 rounds
128	**548 bits**	**4 rounds**	1312 bits	7 rounds	4714 bits	7 rounds	2101 bits	12 rounds

Table 7: Comm. cost of $\prod_{\mathsf{CMP}}$ as function of l and m.

l \ m	4	8	16	32	64	128
			Offline Phase (KB)			
4	1.51	-	-	-	-	-
8	3.04	3.02	-	-	-	-
16	6.10	6.05	6.04	-	-	-
32	12.26	12.14	12.09	12.07	-	-
64	24.58	24.33	24.21	24.16	24.14	-
128	49.24	48.72	48.47	48.35	48.30	48.28
			Online Phase (bits)			
4	16	-	-	-	-	-
8	36	32	-	-	-	-
16	78	68	64	-	-	-
32	164	142	132	128	-	-
64	338	292	270	260	256	-
128	688	594	548	526	516	512

Table 8: Comm. cost of CrypTFlow2 as function of l and m.

l \ m	4	8	16	32	64	128
			Offline Phase (KB)			
4	0.03	-	-	-	-	-
8	0.08	0.03	-	-	-	-
16	0.19	0.08	-	-	-	-
32	0.44	0.19	-	-	-	-
64	0.94	0.44	-	-	-	-
128	1.95	0.94	-	-	-	-
			Online Phase (bits)			
4	20	-	-	-	-	-
8	60	264	-	-	-	-
16	142	788	6.5×10^4	-	-	-
32	308	1838	1.9×10^5	4.2×10^9	-	-
64	642	3940	4.5×10^5	1.2×10^{10}	1.8×10^{19}	-
128	1312	8146	9.8×10^5	3×10^{10}	5.5×10^{19}	3.4×10^{38}

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

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

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

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

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

I 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 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.