

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, New York
schoppmann@google.com

Karn Seth
Google, New York
karn@google.com

ABSTRACT

We present a new construction for secure logistic regression training, which enables two parties to train a 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 protocol with better communication; secure spline evaluation and secure powers computation protocols for fixed-point values; and a new comparison protocol that optimizes online communication. We also present a new two-party protocol for generating keys for distributed point functions (DPFs) over arithmetic sharing, where previous constructions do this only for Boolean outputs. We implement our protocol in an end-to-end system and benchmark its efficiency. We can securely evaluate a sigmoid in 18 ms online time and 0.5 KB of online communication. Our system can train a model over a database with 70,000 samples and 15 features with online communication of 208.09 MB and online time of 2.24 hours at the cost of 6.11c over WAN. Our benchmarks demonstrate that we reduce online communication over state of the art by $\approx 10\times$ for sigmoid and $\approx 38\times$ for logistic regression training.

1 INTRODUCTION

One of the most ubiquitous ways to understand and use large amounts of data is to train models which capture the most significant general properties of the underlying data. In many settings the dataset used for the model training is owned by different parties that have agreed to cooperate and create a common model across their datasets but do not want to share record level data. Secure multi-party computation (MPC) [39, 73] enables distributed processing of their joint data which guarantees that neither party learns anything more about the data than its designated output.

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

of executing a secure computation protocol for the agreed upon functionality. Apart from keeping the input data confidential from any single entity, this model also restricts the data to a specific use case, which the computing parties have to agree on in advance.

Outsourcing a secure computation to a set of non-colluding servers has been applied in practice several times in the past. The first practical application of MPC, which was used to run a sugar beet auction in Denmark in 2009 [11], relied on three “virtual auctioneers”, Danisko, DKS and SIMAP, who had shares of the inputs of all sellers and bidders and executed an MPC protocol for the auction. A second example is a study that was run by the Estonian government, to test whether students working during studies is correlated with worse performance and dropouts [9]. This study needed to join tax records with education records which are held by different government entities and are not shared. To do this in a privacy preserving manner they executed an MPC with three parties: the Estonian Information System’s Authority, the Ministry of Finance IT center, and the company Cybernetica. The two databases were shared among the three parties who executed an MPC protocol implementing the study methodology.

The two-server setting, which we focus on in this work, was leveraged in the system Prio [24] 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 [23] where the two aggregators were run by Mozilla and the Internet Security Research Group (ISRG). The same design underlies the Private Analytics system implemented by Google and Apple in their Exposure Notifications system [5], where the aggregators are the National Cancer Institute (NCI) at the National Institutes of Health (NIH), and ISRG.

An ongoing effort by Google Chrome, called Privacy Sandbox [43], 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 [42], considers a similar measurement goal, which is to compute aggregate measurements across attributed conversions from all users. Again, MPC with distributed data stewardship can be used for this kind of measurement [46].

While the previous two examples show that privately aggregating user data into histograms is useful by itself, the functionality needs for measurement systems go far beyond, and require more complex model training. Here, communication between the two computing parties quickly becomes the most expensive part of the system. For example, while it may be beneficial for privacy to place the two servers into data centers operated by different cloud providers (e.g., AWS and Google Cloud in the case of ENPA[5]), this

incurs egress charges for all traffic between the two servers, which can be significantly higher than intra-cloud traffic costs. These cloud network costs significantly outweigh computation costs in most settings. Low online communication cost is therefore a crucial design goal for practical secure training protocols.

Logistic Regression. Logistic regression is a tool used for many modeling and measurement settings. It is often used for binary classification and prediction in medical [14, 35], engineering [59], 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 [38]. While not as powerful as Deep Neural Networks (DNNs), it turns out to still be broadly useful for important applications so we focus on it in this work.

Online-offline Computation Model. Our constructions consider the online-offline computation model [34] 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 does not assume a trusted party for the offline stage and proposes that the offline preprocessing is also generated using secure computation between two computation parties. While it is well-known that MPC can be used to distribute any computation that a trusted party could perform [39, 73], efficiency is a concern. We therefore investigate how to efficiently perform the offline phase of our protocols using MPC, though with a greater emphasis on keeping the online phase as cheap as possible.

Differentially Private Output In our scenario, the two computation parties may reveal the output logistic model to a designated output receiver, or alternatively may hold the model shares and later answer inference queries in a distributed manner. While we are not aware of any attacks that use a logistic regression model to recover the input database, the question of how much information different models reveal about the data used for training is an active research area. Making the output differentially private [31] is one approach to guarantee that it cannot be used to extract individual records. Thus, we also consider differentially private logistic regression training in our distributed protocol.

Our Contributions. New Secure Logistic Regression. We present new constructions for two party secure logistic regression training over a database that is cryptographically shared between the two parties. Our constructions optimize the online communication cost of existing approaches ($\approx 38\times$ reduction over MP-SPDZ [52]) while maintaining accuracy close to plaintext training. We present two different protocols: the first one optimizes solely for online communication, while the second one trades off some of the efficiency in the online phase for supporting efficient distributed computation in the offline phase. Both constructions can facilitate a differentially private output model.

Accurate Secure Sigmoid. The core technical component in our logistic regression construction is a new protocol for secure sigmoid evaluation on input that is shared between two parties. It uses a new approximation approach for the sigmoid functionality which achieves 10^{-4} error using 20 fractional bits. The final protocol offers improved communication cost for its online phase. This cost is $\approx 10\times$ smaller than the communication of the state-of-the-art sigmoid construction of SiRNN [64] that uses 16-bit ring while our construction uses 63-bit rings. It is also $\approx 31\times$ better than the online communication for sigmoid in MP-SPDZ [52]. A secure sigmoid evaluation runs between 16 and 19ms in different network setting and includes 0.5-1.18 KB of communication.

Communication Efficient Constant Round Secure Comparison. A main building block for our sigmoid construction is a new comparison protocol for l -bit numbers which uses a new reduction to small bit length comparison, a novel all-prefix AND sub-protocol along with inner product that works in *constant* number of rounds for online computation. It uses only three communication rounds in total whereas the state-of-the-art SynCirc uses rounds logarithmic in l . The online communication for 128-bit numbers comparison is only 522 bits and has an improvement of $\approx 1.3 - 2.6\times$ over SynCirc. Secure comparison is a core building block in a broad range of functionalities far beyond the scope of this paper, such as auctions, database search, biometric authentication, combinatorial problems. This construction may therefore be of independent interest. As a concrete estimate of our improvement, our new secure comparison protocol can be used to execute Batcher’s widely used sorting network [6] in a semi-honest 2-party setting on a 10^4 sized array of 64 bit integers using just 13.69 MB of communication whereas prior works such as SynCirc [61], CryptFlow2 [65], Couteau16 [25] will require 17.88 MB, 36.32 MB and 63.37 MB respectively.

New Techniques for (i)DPFs. The communication and round efficiency of our constructions leverage (incremental) distributed point functions ((i)DPFs) [13, 16] techniques inside our MPC protocols in new ways. We present a new technique for computing the all-prefix Boolean AND of n secret shared bits using iDPFs, which achieves the round efficiency of our new secure comparison protocol. We also present a new construction for two-party generation of distributed point function (DPF) keys with arithmetically shared output values, which is used for distributed offline preprocessing.

Efficient Constructions with Fixed-Point Inputs. The accuracy of our computation relies on fixed-point representation of shared values. We present new constructions for spline evaluation and secure powers computation with fixed point input representation. The latter is

used for secure Taylor approximation where existing approaches work only for integers.

Implementation and Evaluation. We present end-to-end implementation of our protocols, and to our knowledge we are the first implementation that combines FSS-based and secret-sharing-based techniques. We evaluate the costs of our protocols including microbenchmarks for our building blocks such as a new secure comparison and sigmoid evaluation, as well as an end-to-end evaluation of secure logistic regression training. We can train a model over cryptographically shared data of 4200 samples with 5000 features in about 9min with 13MB of communication, which amounts to 0.40c cost. We achieve accuracy close to the plaintext trained model (less than 1% difference). We reduce online communication over state of the art by $\approx 2\times$ for comparison, $\approx 10\times$ for sigmoid and $\approx 38\times$ for logistic regression training.

1.1 Our Approach

We outline the main ideas of our approach in this section and present detailed related work discussion in Appendix D.

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

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

To reduce communication of the online phase of our protocol we rely on techniques from function secret sharing [16] which enable non-interactive computation. In particular we use the multiple interval containment (MIC) gate [15] to identify which interval the input falls into in order to use the approximation function. We also use the MIC gate within the spline approximation on the interval $[0, 1)$ to choose the right linear function.

Distributed Comparison Function (Appendix E). MIC gates leverage distributed comparison functions (DCFs) [15] which rely on function secret sharing [17]. We introduce a reduction from DCFs to incremental distributed point functions (iDPFs) [13], which is conceptually simpler than the previous construction by [15].

Secure Powers Computation with Fixed-Point Representation (Section 5.2) A sigmoid is computed as $1/(1 + e^{-x})$. Our sigmoid approximation for values above 1 has two main components: secure exponentiation for evaluation of $r = e^{-x}$ followed by a secure protocol for powers computation that enables the polynomial evaluation for the Taylor series for $1/(1 + r)$. For the first part we leverage secure exponentiation of Kelkar et al. [51]. For the second part we present a new construction inspired by the

HoneyBadgerMPC secure powers protocol [56], which we extend to work with fractional values in fixed-point representation.

Online-Offline Balanced Protocol (Section 6). The most costly part of our offline computation is the generation of FSS keys, which are needed for MIC gates. In the setting without trusted preprocessing these keys need to be generated using two-party computation, presenting significant costs challenging the offline phase feasibility. Existing approaches either rely on general-purpose MPC, which is expensive because of the need to securely evaluate a PRG, or they use the Doerner-Shelat technique [29], which requires computation exponential in the input size. When applying the MIC gate to spline approximation, this is not an issue because the inputs can be made short by truncation, leveraging the fact that the input is a fixed point number with absolute value ≤ 1 . Nevertheless, in the interval containment functionality, which identifies which type of sigmoid approximation needs to be used, this is no longer the case. This is because we do not have any simple way to reduce the input size. Hence, we would need an FSS gate with a large input domain, which would have extremely high offline computation.

Secure Comparison (Section 6.1). To overcome this challenge we modify the protocol to use a secure comparison functionality instead of MIC to determine the first level of input partitions. We introduce a comparison construction with a highly communication-efficient, constant round online phase while only having modest computation complexity. It relies on a new reduction from n -bit numbers comparison to comparison on single bits using functionality that computes the AND over the bits in all bit prefixes of a number. We present a new non-interactive construction for the latter functionality, where the input is split among two parties, which uses iDPFs. The resulting protocol improves over the online communication of Rathee et al.’s CryptFlow2 [66] by $\approx 2.4\times$ and Couteau [25] 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 [25] presents an extensive list of such applications including oblivious sorting, database search constructions, private set intersection, oblivious RAM, machine learning for applications such as classification, feature extraction, and generating private recommendations.

Secure DPF Key Generation (Appendix F). In Appendix F, we give a new 2-party protocol for generating DPF keys for arithmetic-shared outputs using MPC. This is an important extension to [29], who only handle Boolean-shared outputs. Our construction only requires a single additional oblivious transfer in the offline phase, independent of the size of the output shares.

2 PRELIMINARIES

Notation. Given a finite set S , $x \leftarrow S$ indicates that an element x is sampled uniformly at random from S . For any positive integer n , \mathbb{Z}_n denotes the set of integers modulo n . $[k]$ denotes the set of integers $\{1, \dots, k\}$. 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} , $v_{i\dots j}$ denotes the vector

with elements v_i, \dots, v_j . Likewise, for a matrix M , $M_{i\dots j}$ denotes the matrix containing rows i through j from M .

Fixed-Point Representation. A fixed-point representation is parameterized by a tuple $(\mathcal{R}, w, s, \text{Fix})$ where \mathcal{R} is a ring, w represents the bitwidth, s represents the scale (or the fractional bitwidth), and $\text{Fix} : \mathbb{R} \rightarrow \mathcal{R}$ 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 $\text{Fix}(x) = \lfloor x \cdot 2^s \rfloor \bmod L$. In this mapping, all real numbers having absolute value at most 2^{w-s} have a corresponding fixed-point representation in the ring. Specifically, non-negative real numbers are mapped to $[0, 2^w)$ whereas negative real numbers are mapped to $(L - 2^w, L)$ in their two's complement representation. Let $\mathcal{R}^* = [0, 2^w) \cup (L - 2^w, L)$ denote the part of the ring where fixed-point numbers are represented. Note that two distinct real values might have the same fixed point representation because of the limited fractional bitwidth. We will use \tilde{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* [39], i.e., the parties are assumed to follow the prescribed protocol. We denote the two parties by P_0 and P_1 . The security in such a model is captured by the standard real/ideal paradigm whereby the view of an adversary in the real-world, that corrupts one of the parties (either P_0 or P_1), can be efficiently simulated in the ideal world where parties interact with an ideal functionality. A detailed description of the security model can be found in Appendix L.

The protocol Π may be divided into an offline preprocessing phase Π_{offline} (independent of parties' inputs) and an online phase Π_{online} that depends on parties' inputs. In practical settings, Π_{offline} may be performed by a trusted third party, or by the parties executing an MPC protocol. However, Π_{online} is always performed by parties using MPC. Due to space constraints, we do not provide formal proofs of security of our protocols. In the semi-honest model, the security of our protocols follows directly from the security of the underlying primitives.

Secret Sharing. We use $\llbracket x \rrbracket^{\mathcal{R}}$ 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 $\llbracket x \rrbracket = (\llbracket x \rrbracket_0, \llbracket x \rrbracket_1)$ to denote that P_0 and P_1 get shares $\llbracket x \rrbracket_0$ and $\llbracket x \rrbracket_1$ respectively, such that $\llbracket x \rrbracket_0 + \llbracket x \rrbracket_1 = x$ in \mathcal{R} . An additive sharing is random if $\llbracket x \rrbracket_0$ and $\llbracket x \rrbracket_1$ are uniformly distributed in \mathcal{R} subject to $\llbracket x \rrbracket_0 + \llbracket x \rrbracket_1 = x$. 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 $\mathcal{F}_{\text{truncate}}$ to reduce the scale to s' bits where $0 \leq s' \leq s$. An efficient instantiation of $\mathcal{F}_{\text{truncate}}$ was described in SecureML [58]: Suppose x_0 and x_1 are the shares of \widehat{x} held by party P_0 and P_1

respectively. Then, to perform the truncate operation, both parties can locally truncate the last $s - s'$ bits of their individual shares to get new shares x'_0 and x'_1 . Let x' denote the true truncated value of x after truncating the last $s - s'$ bits. SecureML [58] shows that $\text{Recon}(x'_0, x'_1) \in \{x' - 1, x', x' + 1\}$. In other words, this non-interactive truncation protocol incurs a small error in the least significant bit of the fractional part of the FXP value. For our purposes, this error is tolerable as the FXP representation itself admits an error in the least significant bit of the fractional part compared to the actual real value.

2.1 Logistic Regression

Logistic regression is a probabilistic classifier and a supervised learning algorithm [50]. The classification function f takes an observation, which is a vector of features \vec{x}_i , and outputs the class y with highest likelihood. It leverages the sigmoid functionality $\sigma(z) = \frac{1}{1+e^{-z}}$ to assign probability determining the class to an input feature vector \vec{x} , using the weight vector \vec{w} and a bias term b , which form the model. More specifically, $\sigma(\vec{x} \cdot \vec{w} + b)$ outputs the probability of mapping \vec{x} to the class 1.

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

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

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

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 minibatch technique uses a subset of the samples in each iteration rather than the whole batch.

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

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

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

2.2 Multiplication Triples

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

For multiplying an $n \times m$ matrix X with another $m \times k$ matrix Y , there is a special matrix multiplication protocol based on *matrix* Beaver triples which incurs an online communication of $2(nm + mk)$

per party. We will use $\mathcal{F}_{\text{matMult}}$ to abstractly represent a functionality which enables multiplication of two additively shared matrices.

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

These functionalities can be extended to real numbers represented in fixed-point format by adding an additional truncation protocol at the end, where s least significant bits are truncated from the result in order to adjust the fractional scale. We use the non-interactive SecureML truncation [58] described in Section 2.

2.3 Two Party Computation Functionalities

Multiplexer Following [65], we will use \mathcal{F}_{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}_{MUX} to get an arithmetic sharing of x if $b = 1$, and arithmetic sharing of 0 otherwise. A protocol for \mathcal{F}_{MUX} can be realized using 2 simultaneous OTs. In some scenarios, a variant of \mathcal{F}_{MUX} , denoted by $\mathcal{F}_{\text{MUX2}}$, might be more useful. It takes arithmetic shares of x_0 and x_1 , along with Boolean sharing of a selection bit b , and outputs a fresh sharing of x_0 if $b = 0$, and a fresh sharing of x_1 otherwise. A protocol for $\mathcal{F}_{\text{MUX2}}$ can be realized using a single call to \mathcal{F}_{MUX} as follows: Parties locally compute a sharing of $x_1 - x_0$, invoke \mathcal{F}_{MUX} on it with the sharing of b , and finally locally add the sharing of x_0 to their output from \mathcal{F}_{MUX} .

2.4 Function Secret Sharing

We use Boyle et al.’s definition of function secret sharing (FSS) [16]. A 2-party FSS is an algorithm that efficiently splits a function f into two additive shares f_0 and f_1 . These shares must satisfy the following two properties: (1) f_i hides f and (2) $f_0(x) + f_1(x) = f(x)$ for every input x . Output reconstruction in (2) is *additive*. Formally:

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

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

Distributed point function (DPF) [16] is an FSS for a point function that evaluates non-zero on a single point. A DPF allows a compressed 2-party secret-sharing of a point function. Incremental distributed point functions (iDPFs) [13] are a generalization of DPFs which allow compressed sharing of a binary tree with 2^n leaves and a unique special path from root to leaf, i.e., there is a single non-zero path in the tree, ending at leaf α , whose nodes have non-zero values β_1, \dots, β_n . More specifically, iDPF allows a 2-party secret-sharing of an *all-prefix point* function $f_{\alpha, \vec{\beta}}$, where $\alpha \in \{0, 1\}^n$, $\vec{\beta} = ((\mathbb{G}_1, \beta_1), \dots, (\mathbb{G}_n, \beta_n))$, and for each $l \in [n]$:

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

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

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

Secure Computation via FSS. Boyle et. al. [15, 18] showed that the FSS paradigm can be used to efficiently evaluate some function families in 2PC in the preprocessing model, where Gen and Eval correspond to the offline and online phase, respectively.

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

3 SECURE LOGISTIC REGRESSION

We aim to develop concretely-efficient secure two-party computation protocols for logistic regression training, focusing on online communication and rounds of interaction. The setting is as follows: There are two parties (servers), each holding a share of private dataset X , consisting of n examples (rows) and k features (columns), and a sharing of the vector of n labels y . They wish to train a logistic regression model w and end up with a sharing of w . The two parties are non-colluding and semi-honest. In this setting, the security of a protocol Π is formally captured via the real-ideal paradigm as mentioned in Section 2 and detailed in Appendix L.

Like previous works [58, 70], we leverage arithmetic secret-sharing (see Section 2) and train the model w with stochastic gradient descent (SGD). Our protocol is described in Algorithm 1. It makes heavy use of correlated matrix-vector multiplication using Beaver triples, and crucially depends on an implementation of the sigmoid function in MPC.

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

4 SECURE SIGMOID

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

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

Division is sometimes approximated via Goldschmidt’s [40] method, which is expensive. Alternatively, exponentiation can be approximated by decomposing the exponent into bits [26], which is costly, or via low-degree polynomials [4], which is inaccurate.

In this section, we present our sigmoid functionality (Algorithm 2) which is actually the sigmoid approximation we achieve.

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 $\llbracket X \rrbracket \in R^{n \times k}$ and labels $\llbracket y \rrbracket \in R^n$.

```
1 Let  $\llbracket w_0 \rrbracket$  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      $\llbracket X_B \rrbracket \leftarrow \llbracket X_{i..j} \rrbracket$ 
7      $\llbracket u \rrbracket \leftarrow \mathcal{F}_{\text{corrMatMult}}(\llbracket X_B \rrbracket, \llbracket w_{t-1} \rrbracket)$ 
8      $\llbracket s \rrbracket \leftarrow \mathcal{F}_{\text{Sigmoid}}(\llbracket u \rrbracket)$ 
9      $\llbracket d \rrbracket \leftarrow \llbracket s \rrbracket - \llbracket y_{i..j} \rrbracket$ 
10     $\llbracket g \rrbracket \leftarrow \mathcal{F}_{\text{corrMatMult}}(\llbracket X_B^\top \rrbracket, \llbracket d \rrbracket)$ 
11     $\llbracket w_t \rrbracket \leftarrow \llbracket w_{t-1} \rrbracket - (\alpha/B) \cdot (\llbracket g \rrbracket + \lambda \cdot \llbracket w_{t-1} \rrbracket)$ 
12 return  $\llbracket w_T \rrbracket$ .
```

We describe how we securely implement this approximate functionality, pointing to Section 5.1 and Section 5.2 for detail on how we implement the more complex components of our functionality.

4.1 Sigmoid Approximation

The sigmoid function is 'symmetric' around the y -axis. More specifically, $S(x) + S(-x) = 1$ for all $x \in \mathbb{R}$. This implies we can focus on evaluating $S(x)$ and then compute $S(-x) = 1 - S(x)$ locally. For $x \geq 0$, we need to compute both division and exponentiation in MPC. First, we show how we bypass directly computing division.

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

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

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 [56]. Our protocol computes *all* powers of r (irrespective of the degree) in only 2 communication rounds.

However, this approximation works well only when $r \ll 1$. We therefore use this approximation only on the interval $[0, \frac{1}{e}]$. As $r = e^{-x}$, we use this technique when $x \geq 1$. In order to compute e^{-x} , we use the 1-round exponentiation technique of [51]. We note that the exponentiation protocol from [51] 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 5.1.¹

For security, neither party should learn which technique is used to compute sigmoid (i.e. in which interval x belongs). Thus, all evaluations are run simultaneously. At the end, the right output is obviously selected and secret-shared between the parties.

Algorithm 2: Approximate Sigmoid

Parameters:

Let m be the number of lines defining a spline.

Let l_f be the number of fractional bits.

Let d be the degree of Taylor series approximation.

Private input:

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

Sigmoid(x) :

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

5 SECURE SIGMOID WITH TRUSTED SETUP

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

In designing our protocol, we assume that the offline phase of our protocol is part of a trusted setup phase². In practical settings, such a trusted setup can be performed by a trusted third party. Another possibility, when the intermediate models are protected by DP (see Algorithm 10 in Appendix G), is to outsource the setup phase to (semi-honest) clients. These clients may provide a portion of the precomputed setup alongside the inputs they upload to the

¹One could imagine using only splines to approximate sigmoid on the entire $(-\infty, \infty)$ interval as has been done in prior works [58]. However, this would require a large number of spline intervals and potentially higher degree splines for the approximation to work well, thus increasing the cost of protocol. Therefore, in our heterogeneous approach, we use the spline based approximation only for the smaller interval $[-1, 1]$. For the remaining interval, we use the new exponentiation combined with Taylor series based approximation.

²Regardless of how the trusted setup phase is actually realized, it is assumed that this setup phase is performed by an entity which is not colluding with any party.

two MPC parties. In case a trusted setup is infeasible, we discuss how to perform the offline phase in MPC as well in Section 6.

Our sigmoid approximation will work by first using \mathcal{F}_{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}_{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}_{MIC} with the outputs of the tailored sigmoid computations to “select” the output of sigmoid on x using the approximation corresponding to the interval in which x lies. This dot product can be computed using standard Beaver multiplication [8]. In 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 $S : \mathbb{R} \rightarrow \mathbb{R}$ 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 $\vec{B} = [b_1, b_2, \dots, b_m]$ where $b_i = 1\{p_i \leq x < q\}$. 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 $\mathcal{F}_{\text{matMult}}$. 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 $\mathcal{F}_{\text{matMult}}$. In Appendix J, we describe an optimized protocol for performing the dot-product (for the specific case of splines) which reduces the overall communication of spline to just 6 elements of communication. Crucially, this optimization makes the online communication cost of spline independent of the number of intervals m .

Note that in our Sigmoid Approximation described in Algorithm 2, we use spline only when the input is between $[0, 1)$. This means that the spline protocol only needs to be executed on the fractional bits of the input. In other words, given a positive fixed point input x , let $y = x \bmod 2^s$. It is easy to see that y represents the fractional bits of x . In the secret-shared setting, parties can locally compute $\llbracket y \rrbracket := \llbracket x \rrbracket \bmod 2^s$ to derive a sharing of the fractional bits of x in

the smaller ring $\mathcal{R} = \mathbb{Z}_{2^s}$. Now parties can use (shares of) y for evaluating the MIC component of the spline protocol, thus reducing the domain size of the MIC from l bits to s bits. This observation will be needed later in Section 6. If we set the output domain of MIC to be \mathbb{Z}_{2^l} and compute the \vec{Q} over $\mathcal{R} = \mathbb{Z}_{2^l}$, we can ensure that the final output of spline protocol is shared in $\mathcal{R} = \mathbb{Z}_{2^l}$ to be compatible for further computations.

5.2 Secure Powers Evaluation

To evaluate a Taylor series approximation inside MPC, we need a procedure to securely compute a d -degree polynomial which, in turn, requires computing the (secret-shares of) consecutive powers $\{x, x^2, \dots, x^d\}$ for a (secret-shared) input x . Naively, one could invoke $\mathcal{F}_{\text{Mult}}$ repeatedly d times in order to generate these powers. However, this makes the communication-cost proportional to the degree d . In [56], the authors proposed a novel protocol to generate all d powers using a single element of online communication per party, where the masked value $x_{\text{mask}} = x - r$ is revealed. The protocol leverages a new type of offline pre-processing correlation called “random powers”. In such a correlation, parties have a sharing of $\{r, r^2, \dots, r^d\}$ for a uniformly random $r \in \mathcal{R}$. For a (secret-shared) input x in the online phase, the parties “consume” these special correlations in order to generate a sharing of $\{x, x^2, \dots, x^d\}$. The main observation in the protocol is the following relationship:

$$\llbracket x^i r^j \rrbracket = \llbracket r^{i+j} \rrbracket + x_{\text{mask}} \left(\sum_{l=0}^{i-1} \llbracket x^{i-1-l} r^{j+l} \rrbracket \right) \quad (1)$$

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 ³), (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 evaluating 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

³A 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.

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 $+x_{\text{fracMask}}$ if $x \geq r$, and $-x_{\text{fracMask}}$ otherwise. We also observe that parties can locally compute a sharing of bit $t = 1\{x \geq r\}$ as shown in Line 7 in Algorithm 3.

In the actual protocol, we invoke a fixed-point adapted version of the powers protocol from [56] on both $+x_{\text{fracMask}}$ and $-x_{\text{fracMask}}$. 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 $\mathcal{F}_{\text{MUX2}}$ as a black-box.

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

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 $\Pi_{\text{fxpPowers}}$ to learn sharing of $\{x, x^2, \dots, x^k\}$, 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 $\Pi_{\text{fxpPowers}}$. We observe that one could do better by modifying $\Pi_{\text{fxpPowers}}$ as follows: in Line 9, instead of invoking $\mathcal{F}_{\text{MUX2}}$ 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 $\mathcal{F}_{\text{MUX2}}$ on inputs $(P^0, P^1, \langle t \rangle)$. This reduces the total communication cost of the protocol to only $2(s + 1 + 2l)$ bits, making it *independent* of the degree d of the polynomial Q . We refer to this optimized protocol as Π_{fxpPoly} .

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

Algorithm 3: Fixed-point powers Protocol

```

 $\Pi_{\text{fxpPowers}}$  :
Input :  $\llbracket x \rrbracket$ , where  $x \in [0, 2^s)$  and  $\tilde{x} \in [0, 1)$ 
Output :  $\llbracket \tilde{y} \rrbracket, \llbracket \tilde{y}^2 \rrbracket, \dots, \llbracket \tilde{y}^k \rrbracket$ , where  $y = \tilde{x}$ 
Precomputation:  $\llbracket r \rrbracket, \llbracket r^2 \rrbracket, \dots, \llbracket r^k \rrbracket$ , where  $r \in \mathbb{R}$  and  $r \leftarrow [0, 1)$ 

1  $\llbracket x - r \rrbracket \leftarrow \llbracket x \rrbracket - \llbracket r \rrbracket$ 
2  $x_{\text{fracMask}} := \text{Recon}(\llbracket x - r \rrbracket^s)$ , where  $\llbracket x - r \rrbracket^s$  is the  $s$  least
   significant bits of  $\llbracket x - r \rrbracket$  and  $\text{Recon}$  happens in the ring  $\mathbb{Z}_{2^s}$ .
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  $\{p_i^0\}_{i \in [k]} \leftarrow \Pi_{\text{maskPowers}}(x_{\text{fracMask}}^0)$ , where  $x_{\text{fracMask}}^0 := 0^{l-s} \| x_{\text{mask}}$ 
5  $\{p_i^1\}_{i \in [k]} \leftarrow \Pi_{\text{maskPowers}}(x_{\text{fracMask}}^1)$ , where  $x_{\text{fracMask}}^1 := 1^{l-s} \| x_{\text{mask}}$ 
   // The value of  $x - r$  is  $x_{\text{fracMask}}^0$  if  $x \geq r$ , else  $x_{\text{fracMask}}^1$ .
6 Let  $f$  denote the bit of  $\llbracket x - r \rrbracket$  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  $\forall i \in [k] : \text{res}_i \leftarrow \mathcal{F}_{\text{MUX2}}(p_i^0, p_i^1, \langle t \rangle)$ .
   // Parties use the  $t$  bit to select the correct set of powers.
10 return  $\text{res}_1, \text{res}_2, \dots, \text{res}_k$ 
.....
// Local subprocedure invoked by each party  $P_i$ 
 $\Pi_{\text{maskPowers}}$  :
Input:  $x_{\text{mask}}$  where  $x_{\text{mask}} \in [0, 2^s)$ 
Output:  $\llbracket \tilde{y} \rrbracket, \llbracket \tilde{y}^2 \rrbracket, \dots, \llbracket \tilde{y}^k \rrbracket$ , where  $y = \widetilde{x_{\text{mask}}}$ 

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

```

securely emulating the trusted offline phase in a 2PC setting. This happens because the FSS key generation algorithm involves the usage of a PRG, and naively running the FSS key generation algorithm inside 2PC will involve the cost of computing the PRG circuit (e.g. AES) using 2PC. This will typically⁴ blow up the communication cost of the offline phase by at least linear in the size of PRG circuit.

In this section, we will discuss an alternative approach for computing sigmoid which will enable a communication efficient offline phase while adding a mild communication overhead in the online phase. We do this by simply replacing the offline-expensive MIC

⁴This 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.

gate (which is based on FSS) with a novel communication efficient secure comparison protocol.

In the previous construction, we have used the FSS based \mathcal{F}_{MIC} functionality at two different places. We use \mathcal{F}_{MIC} to determine if the shares of input x lie in the range $[0, 1)$, $[1, \frac{1}{\log_2(e)})$, $[\frac{1}{\log_2(e)}, \infty)$, or the negative equivalents of these ranges. Besides this, we also use \mathcal{F}_{MIC} as a sub-protocol inside the secure spline functionality. Our new construction will be nearly a drop-in replacement for the first \mathcal{F}_{MIC} functionality based on FSS. As mentioned in Section 5.1, the second spline only needs to operate on the fractional bits of the input and thus can be instantiated on a much smaller domain of s bits. Thus, we retain the second \mathcal{F}_{MIC} as we can use [29] to efficiently generate FSS keys in the distributed setting when $s \leq 24$.

However, one difference is that our construction returns XOR Boolean shares rather than arithmetic shares of a Boolean value. This means that rather than a Beaver-multiplication based dot-product, we instead use \mathcal{F}_{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}_{MUX} input to either retrieve shares of the sigmoid evaluation on the interval, or shares of 0, and then add together these shares across all intervals to select the sigmoid result.

6.1 Secure Comparison

Suppose party P_0 and P_1 have a private input x and y respectively. The output of a secure comparison functionality, henceforth denoted as \mathcal{F}_{CMP} , is a boolean sharing of $\mathbf{1}\{x < y\}$, a bit indicating the result of comparison, where x and y are bitstrings of length l (interpreted as unsigned bit representation of positive integers). Formally, $\mathcal{F}_{\text{CMP}}^l(x, y) \rightarrow (b_0, b_1)$, where $x, y \in \{0, 1\}^l$ and b_0, b_1 is a Boolean sharing of bit $b := \mathbf{1}\{x < y\}$.

A common approach to computing secure comparison is divide-and-conquer [25, 36, 65], which first splits the larger input strings into smaller strings, performs comparisons on these smaller strings, and then combines the results. However, these protocols have non-constant number of rounds in the online phase due to a logarithmic depth recursion tree. Cheetah [47] optimizes the offline communication of [65] using VOLE-style OT extension which is orthogonal to our focus on online efficiency.

Another line of work based on function secret sharing (FSS) [15] performs secure comparison using a distributed comparison function (DCF). This technique allows an online optimal protocol for secure comparison having 1 round and 1 element of communication per party. However, the caveat of directly using FSS to perform comparison is the expensive cost of running the FSS offline phase in 2PC. While Doerner and Shelat [29] propose an elegant approach for performing FSS offline phase, their technique is efficient only for small domains (i.e. input bit lengths less than 20). This is because it requires locally computing an exponential (in input bit length) number of PRGs. There is at this moment no better concretely communication-efficient technique in the literature for conducting the FSS offline phase.

Another line of work originating from ABY [27] and its successors ABY 2.0 [60] and SynCirc [61] solve secure comparison by running a GMW-style MPC on a (variant of) Boolean adder circuit. These techniques have round complexity proportional to

the depth of the circuit which is $O(\log l)$ whereas our protocol is constant round (at most 3). In Table 6, we compare the cost of our protocol with SynCirc (the most optimized work in this direction). Rabbit [57] solves the secure comparison problem in the dishonest majority multiparty setting using Boolean adder circuit and other techniques, has higher communication cost and $O(\log l)$ rounds.

In our approach, we start by looking at the decomposition of comparison problem for l -bit strings in terms of comparison and equality operations on smaller sub-strings as described in Garay et al. [36]. Formally, for $x = x_1 || x_2$ and $y = y_1 || y_2$, where $x, y \in \{0, 1\}^l$ are l -bit strings, the following relationship holds:

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

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

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

Looking ahead, the reason for splitting l length bit-string into smaller sub-strings of length m bits is to leverage the power of FSS gates for small input domains (e.g. $m = 16$ bits) which have an efficient offline phase. Assuming, $l_i = x_i \stackrel{?}{<} y_i$ and $e_i = x_i \stackrel{?}{=} y_i$, we can rewrite the above equivalence as:

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

At a high-level, our protocol:

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

Step 1 and Step 2 follow directly from FSS gates for comparison and equality constructed in [15] and described in Appendix B. Step 4 can be performed in a standard way using bit Beaver triples. We elaborate on Step 3, i.e. how to use the iDPF in order to compute the all-prefix AND of e_i values. We first observe: A single DPF can be easily used to compute the Boolean AND of k bits b_1, \dots, b_k . The observation is that the AND of k bits can be represented as a point function in the following way:

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

where $b = b_1 || \dots || b_k$.

Note that in our context, we want to compute the AND on e_i values. A naive solution is to use $q - 1$ independent DPFs to compute

all the prefix AND values i.e. $e_1, e_1 \wedge e_2, e_1 \wedge e_2 \wedge e_3$ and so on. However, since the ANDs are correlated and have an incremental pattern, we can instead use a *single iDPF* to perform the above task much more efficiently. Let’s consider the following point function:

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

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

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

As mentioned in Section 2.4, using an FSS scheme for function f in the context of MPC is done via the corresponding offset function which works on the masked input value $\tilde{x} := x + r_{in}$ instead of the actual private value x . The input mask r_{in} is defined in the offline phase and is used to define the parameters for FSS key generation. In [15, 18], the authors use this technique to create an equality check gate and comparison gate via FSS schemes such as DPF and DCF respectively. However, leveraging iDPF in order to create useful MPC gates has been unexplored. In the preceding paragraph, we outlined the way an iDPF can be leveraged for computing the all-prefix AND of multiple bits. However, to use this idea in the context of MPC, we need to operate on masked input and somehow encode the mask inside the iDPF without affecting the correctness. Here we observe that the typical way of masking via group addition i.e. $\tilde{x} := x + r_{in}$ and then trying to set the special point $\alpha = 2^{q-1} - 1 + r_{in}$ does not work. This is because if $x_1 || \dots || x_k$ is a length k prefix of $2^{q-1} - 1$, then it doesn’t imply that $\tilde{x}_1 || \dots || \tilde{x}_k$ is also a length k prefix of α . This means that instantiating a iDPF at α and then performing incremental evaluations on the masked input would not lead to the correct prefix AND result. Our solution to this problem is to use XOR masking instead of the usual group addition based masking. Specifically, we define the masked input $\tilde{x} := x \oplus r_{in}$ and then instantiate an iDPF with the special point $\alpha = (2^{q-1} - 1) \oplus r_{in}$. It is easy to see that with this masking technique, the following equivalence holds: $x_1 || \dots || x_k$ is a length k prefix of $2^{q-1} - 1$ iff $\tilde{x}_1 || \dots || \tilde{x}_k$ is a length k prefix of α . We describe the protocol Π_{CMP} formally in Figure 2

7 EXPERIMENTAL EVALUATION

Implementation Details. We implemented our constructions in C++ and compiled our system with the Bazel build system [7]. We used native C++ `uint64_t` for most operations. For FSS operations, we used `uint_128` from the Abseil library [41].

Experimental Setup. We ran our experiments on two compute-optimized c2-standard-8 Google Cloud instances with 32 GB RAM and Intel Xeon CPU at 3.1 GHz clock rate (except for experiments run on the Criteo datasets, where we used compute-optimized c2-standard-60 instances with 240 GB RAM and same CPU). Our implementation runs on a single thread and uses a single core of each instance. In the LAN setting, both instances were deployed in the us-central1 region where the mean network latency was 0.15ms and the bandwidth was $\approx 2.1\text{GB/s}$. In the WAN setting, one

instance was in us-central1 while the other was in us-west2. The mean network latency was 46.20ms and the bandwidth $\approx 60\text{MB/s}$. All runtimes and communication are end-to-end totals and include both the client and server costs, with all computation sequentialized (i.e., server and client do not compute at the same time).

Cloud costs. We include the monetary cost of running our protocols on the Google Cloud Platform (GCP), using the prices listed on the GCP website. For computational cost, we use the CPU spot price of \$0.02 per-hour for pre-emptible virtual machines, and use network cost of \$0.08 per GB for egress to the internet. All prices are in USD. This reflects batch computation with parties situated in different cloud providers, as has been used in other works [48].

7.1 Sigmoid Experiments

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

Sigmoid Accuracy Our sigmoid approximation error as compared to the Python floating point implementation of sigmoid (“true” sigmoid) is at most on the order of 10^{-4} when run on points uniformly spaced on the interval $[-20, 20]$ with 20 fractional bits.

Table 1: Comparison of the online cost of our sigmoids with trusted (v1) and distributed (v2) offline setup to SiRNN and MP-SPDZ’s accurate sigmoid implementation.

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

Benchmark Comparisons. We compare to the most recent secure sigmoid protocols in Table 1. We focus on *accurate* sigmoid approximations as inaccurate approximations often result in worse models than in standard logistic regression (see Section 7.2). Our comparison includes SiRNN [64], whose sigmoid protocol strictly improves over other state-of-the-art sigmoid approximations such as MiniONN [55] and DeepSecure [68]. We also compare to the sigmoid approximation presented in MP-SPDZ⁶ [52].

⁵We were not able to compile 10^4 sigmoid executions into the bytecode used by the MP-SPDZ virtual machine as our device ran out of memory.

⁶MP-SPDZ presents different sigmoid approximations. We focus our comparison on their accurate sigmoid approximation, which directly computes exponentiation and reciprocal in MPC.

We retrieved the SiRNN values directly from [64] and extrapolated the cost for our network settings⁷. For MP-SPDZ, we ran their sigmoid implementation in the trusted dealer mode.

We observe a gain of $\approx 10\times$ in communication efficiency over SiRNN and $\approx 30\times$ over MP-SPDZ. We also reduce the rounds for sigmoid ($\approx 25\times$ over SiRNN and $\approx 31\times$ over MP-SPDZ): Our v1 sigmoid requires 4 rounds while our v2 sigmoid requires 6 rounds, while we estimate that SiRNN uses ≈ 100 communication rounds (though the number of rounds is not discussed in SiRNN) while MP-SPDZ uses 124 rounds. While our construction has higher computation than SiRNN and MP-SPDZ, we have better monetary costs. We decrease costs by $\approx 2.7\times$ over SiRNN and by $\approx 8.4\times$ over MP-SPDZ on LAN. On a higher latency WAN, we decrease costs by $\approx 2.7\times$ over SiRNN and by $\approx 8.6\times$ over MP-SPDZ. SecFloat [62], a concurrent work, does not explicitly provide a sigmoid protocol but our estimate is that it will require 271 rounds and 47.5 KB of communication (as it uses 187 rounds and 37.23 KB per exponentiation and 84 rounds and 10.27 KB per division) whereas we use only 4/6 rounds 0.5KB/1.18KB of communication for v1/v2 respectively.

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

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

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

- MIC gate: Requires a key size of $(l \cdot \lambda + l^2 + 3l + \lambda) + 2 \cdot n_I \cdot l$ bits where n_I is the number of MIC intervals [15]. For $l = 63$ and $n_I = 6$, we get a key size of 1.6 KB.
- Exponentiation: Requires a correlation consisting of 2 field elements of at most l bits [51]. For $l = 63$, the cost is 0.015 KB.
- Polynomial: Requires a correlation consisting of (shares of) d incremental powers of a random ring element. For $l = 63$, we get a cost of 0.077 KB.
- Spline: Requires a MIC gate key for 10 intervals and 1 Beaver triple. For $l = 63$, we get a cost of 0.56 KB.
- Dot product: For performing dot product of two vectors of length 6, we require 6 Beaver triples. For $l = 63$, we get a cost of 0.046 KB.

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

⁷Note that SiRNN does not have offline phase. The entire protocol cost is online.

For the v2 setting, the offline cost is measured by the total communication cost needed to generate the required preprocessing material in MPC. In the following, we assume that the amortized cost of generating a Beaver triple for $l = 63$ bit ring is 0.4375 KB [67] and a bit Beaver triple is 0.0175 KB [65]. The v2 sigmoid protocol consists of the following components:

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

In the v2 sigmoid protocol, we invoke 1 MIC gate, 2 exponentiations, 2 polynomial calls, 2 spline calls and 6 Mux calls. This requires a total offline communication cost (across both parties) of 284 KB per party. The offline computation cost is dominated by FSS key generation which involves 2^n MPC calls to AES for n -bit FSS inputs. Based on the estimate of 360 million AES calls per second (as reported in [15]), we decided what value of n would be reasonable/feasible. In the v2 sigmoid protocol, we invoke FSS when computing the Spline MIC gate on s bit inputs. Therefore, setting $s = 20$ enables an estimated offline computation time of 23 milliseconds per sigmoid.

7.2 Logistic Regression Experiments

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

Table 2: Datasets we used for our experiments.

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

We used the following datasets:

- *Titanic* [33]. This dataset is used to predict which passengers survived the Titanic shipwreck.

- *Arcene* [45]. This dataset contains information from the National Cancer Institute and the Eastern Virginia Medical School. It is used to determine whether a patient has cancer.
- *Gisette* [45]. This dataset distinguishes the digits 4 and 9.
- *Criteo Uplift* [28]. This dataset predicts whether a user targeted by advertising purchases a product (i.e. converts). We used two versions of this dataset. In *Criteo Uplift 1*, we used a random subset of the original dataset. This dataset was highly imbalanced. From 10^5 data points, there were only 470 conversions. In *Criteo Uplift 2*, we sampled the examples such that 10% of the labels were positive. I.e., there were 10^4 positive labels.

Accuracy Evaluation. We compare accuracy of our 2PC protocols against a plaintext Python floating-point implementation in Table 4. We use the parameters from Table 2 and train for 6 iterations of logistic regression, which is enough for the plaintext algorithm to converge. Our 2PC protocols are close to plaintext logistic regression in all cases.

Performance Evaluation. We present our end-to-end runtime and total communication costs in Table 5. All versions use the parameters from Table 2, and run for 6 iterations. Our runtimes and communication are totals for both parties. We observe that our costs grow nearly linearly with the number of examples, but are relatively independent of the number of features. This emphasizes that sigmoid is a significant portion of our protocol costs.

Comparison to previous works. We first note that some key works in the area have relatively coarse sigmoid approximations, and we do not do a detailed comparison with these works. These works include the piecewise approximations of SecureML (3 pieces) [58] and MP-SPDZ (5 pieces) [52]. As a result of these coarse approximations, SecureML and MP-SPDZ do not closely match plaintext logistic regression: Running logistic regression in plaintext with SecureML’s and MP-SPDZ’s sigmoid approximations on a subset of the Criteo uplift dataset with 100,000 examples and 470 positive examples (trained on 6 iterations over the *full* dataset, with minibatches of 100 examples) yields 0 and ≈ 0.42 F1 score, respectively, while Python plaintext (and training in plaintext with our approximation) yields ≈ 0.48 (0.47 when run in MPC). ABY2.0 [60] improves on the efficiency of SecureML’s logistic regression by reducing the online runtime, but uses the same coarse sigmoid approximation.

Note that even if we settled for the sigmoid approximation used in SecureML and ABY2.0 (and similarly MP-SPDZ), our work would still offer an improvement in online communication. This is because their sigmoid approximation essentially reduces to secure comparisons and AND gates. Our improved comparison protocol (see in Table 6) would reduce their training costs.

Turning to accurate logistic regression approximations, MP-SPDZ [52] has one sigmoid approximation that results in comparable accuracy to our protocol. We ran the online phase of MP-SPDZ logistic regression on a dataset of 1000 examples and 10 features and compared to our logistic regression implementation with v1 sigmoid. One iteration of MP-SPDZ communicates ≈ 18.8 MB while our work requires ≈ 0.5 MB, a 38 \times improvement. On an ultra low-latency LAN, our runtime is slower (≈ 0.3 s for MP-SPDZ and 18.6s for our technique) but we perform better on a higher latency WAN

(MP-SPDZ takes ≈ 25.3 s while our technique takes 19.6s) due to MP-SPDZ’s multi-round nature.

SiRNN [64] implements sigmoid, but not logistic regression, so we limit ourselves to the sigmoid comparison in the Section 7.1.

7.3 Secure Comparison Experiments

We compare our secure comparison protocol Π_{CMP} against Boyle et al. [15]’s FSS comparison gate. In Appendix C, we provide an analytical cost comparison with latest works.

Performance Comparison The FSS comparison gate of Boyle et al. [15] presents the lowest known online communication for comparison, but the offline phase is computationally infeasible for 64 bit inputs. We show that with a relatively small increase to our costs (2 \times online communication and a runtime increase from 0.364ms to 0.532ms on a LAN network and a batch of 1000 comparisons), we can make the offline phase computationally feasible (see discussion in Section 6.1). We present our experiments in Table 3.

Table 3: Comparison of our new Π_{CMP} protocol to the FSS protocol [15] on a batch of 1000 inputs.

	Π_{CMP}	FSS Comparison [15]
LAN (sec)	0.532	0.364
WAN (sec)	0.671	0.410
Communication (KB)	29.55	15.63

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

	Python Plaintext	2PC Approach V1	2PC Approach V2
Titanic Dataset			
F1 Score	0.77551	0.77551	0.77551
Accuracy	0.79439	0.79439	0.79439
Arcene Dataset			
F1 Score	0.76923	0.76923	0.76923
Accuracy	0.8	0.8	0.8
Gisette Dataset			
F1 Score	0.96987	0.96540	0.96540
Accuracy	0.97056	0.96611	0.96611
Criteo Uplift 1 Dataset			
F1 Score	0.47910	0.46336	0.46336
Accuracy	0.993	0.992	0.992
Criteo Uplift 2 Dataset			
F1 Score	0.86367	0.86155	0.86155
Accuracy	0.971	0.970	0.970

8 CONCLUSION

We show that techniques from FSS can be combined with secret-sharing MPC to get the best of both worlds. Specifically, we can have reduced communication in the online phase while still having an efficient offline phase. We demonstrate this idea by designing a novel secure logistic regression training protocol with the best known online communication ($\approx 38\times$ lower than MP-SPDZ) and a secure sigmoid evaluation construction with $\approx 10\times$ online communication reduction over state-of-the-art SiRnn. Along the way, we developed many building blocks of independent interest.

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

	2PC V1	2PC V2	2PC with DP V1	2PC with DP V2
Titanic Dataset (500 × 6)				
Comm (MB)	1.49	3.38	1.49	3.38
LAN (sec)	55.34	54.29	55.07	53.46
WAN (sec)	58.7317	58.8215	58.4406	58.5171
LAN Cost (USD)	0.04c	0.06c	0.04c	0.06c
WAN Cost (USD)	0.04c	0.06c	0.04c	0.06c
Arcene Dataset (70 × 10000)				
Comm (MB)	1.24	1.50	1.18	1.45
LAN (sec)	9.47	9.06	9.39	9.00
WAN (sec)	12.99	13.02	12.90	13.04
LAN Cost (USD)	0.015c	0.017c	0.014c	0.016c
WAN Cost (USD)	0.017c	0.019c	0.016c	0.019c
Gisette Dataset (4200 × 5000)				
Comm (MB)	13.00	28.88	12.97	28.85
LAN (min)	8.67	8.56	8.59	8.49
WAN (min)	9.06	8.96	8.88	8.79
LAN Cost (USD)	0.39c	0.51c	0.39c	0.51c
WAN Cost (USD)	0.40c	0.52c	0.40c	0.52c
Críteo Uplift 1/2 Datasets (70000 × 15)				
Comm (MB)	208.09	472.61	208.09	472.61
LAN (hr)	2.23	2.33	2.23	2.32
WAN (hr)	2.24	2.34	2.23	2.33
LAN Cost (USD)	6.09c	8.36c	6.09c	8.34c
WAN Cost (USD)	6.11c	8.37c	6.09c	8.36c

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] Abdelrahman Aly and Nigel P. Smart. 2019. Benchmarking Privacy Preserving Scientific Operations. In *ACNS 19 (LNCS)*, Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung (Eds.), Vol. 11464. Springer, Heidelberg, 509–529. https://doi.org/10.1007/978-3-030-21568-2_25
- [5] Apple and Google. 2021. Exposure Notifications Private Analytics. <https://github.com/google/exposure-notifications-android/blob/master/doc/ENPA.pdf>. (2021).
- [6] Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 307–314.
- [7] Bazel. 2022. Bazel. <https://bazel.build/>. (2022).
- [8] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO ’91 (LNCS)*, Joan Feigenbaum (Ed.), Vol. 576. Springer, Heidelberg, 420–432. https://doi.org/10.1007/3-540-46766-1_34
- [9] 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).
- [10] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. 2014. From input private to universally composable secure multi-party computation primitives. In *2014 IEEE 27th Computer Security Foundations Symposium*. IEEE, 184–198.
- [11] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Kroigaard, 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.).
- [12] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmity Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. *Proceedings of Machine Learning and Systems* 1 (2019), 374–388.
- [13] 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.
- [14] Carl Boyd, Mary Ann Tolson, and Wayne S. Copes. 1987. Evaluating Trauma Care. *The Journal of Trauma: Injury, Infection, and Critical Care* (1987).
- [15] 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
- [16] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *EUROCRYPT 2015, Part II (LNCS)*, Elisabeth Oswald and Marc Fischlin (Eds.), Vol. 9057. Springer, Heidelberg, 337–367. https://doi.org/10.1007/978-3-662-46803-6_12
- [17] 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>
- [18] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation with Pre-processing 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
- [19] Junyoung Byun, Woojin Lee, and Jaewook Lee. 2021. Parameter-free HE-friendly Logistic Regression. *Advances in Neural Information Processing Systems* 34 (2021), 8457–8468.
- [20] 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*.
- [21] Jung Hee Cheon, Wootae Kim, and Jai Hyun Park. 2022. Efficient Homomorphic Evaluation on Large Interval. *Cryptology ePrint Archive* (2022).
- [22] 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*.
- [23] Henry Corrigan-Gibbs. 2020. Privacy-preserving Firefox telemetry with Prio. <https://rwc.iacr.org/2020/slides/Gibbs.pdf>. (2020).
- [24] 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).
- [25] 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
- [26] Ivan Damgård, Matthias Fitz, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. 2006. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *TCC 2006 (LNCS)*, Shai Halevi and Tal Rabin (Eds.), Vol. 3876. Springer, Heidelberg, 285–304. https://doi.org/10.1007/11681878_15
- [27] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY-A framework for efficient mixed-protocol secure two-party computation.. In *NDSS*.
- [28] Diemert Eustache, Betlei Artem, Christophe Renaudin, and Amini Massih-Reza. 2018. A Large Scale Benchmark for Uplift Modeling. In *Proceedings of the AdKDD and TargetAd Workshop, KDD, London, United Kingdom, August, 20, 2018*. ACM.
- [29] 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>
- [30] Cynthia Dwork, Krishnamurthy 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)*.
- [31] 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.).
- [32] Cynthia Dwork and Kobbi Nissim. 2004. Privacy-Preserving Datamining on Vertically Partitioned Databases. In *CRYPTO 2004 (LNCS)*, Matthew Franklin (Ed.), Vol. 3152. Springer, Heidelberg, 528–544. https://doi.org/10.1007/978-3-540-28628-8_32
- [33] Haas Charles Eaton John. 1994. Titanic: Triumph and Tragedy. In *W. W. Norton & Company*.
- [34] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Found. Trends Priv. Secur.* (2018).
- [35] David A. Freedman. 2009. *Statistical Models: Theory and Practice* (2 ed.). Cambridge University Press.
- [36] 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.
- [37] Adrià Gascón, Philipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. 2017. Privacy-Preserving Distributed Linear Regression on High-Dimensional Data. *POPETs 2017*, 4 (Oct. 2017), 345–364. <https://doi.org/10.1515/popets-2017-0053>

- [38] Alexandre Gilotte. 2021. Results from the Criteo-AdKDD-2021 Challenge. <https://medium.com/criteo-engineering/results-from-the-criteo-adkdd-2021-challenge-50abc9fa3a6>. (2021).
- [39] Oded Goldreich. 2009. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press.
- [40] Robert E. Goldschmidt. 1964. Applications of division by convergence. *Master's thesis, MIT* (1964).
- [41] Google. 2022. Abseil - C++ Common Libraries. <https://github.com/abseil/abseil-cpp>. (2022).
- [42] Google. 2022. <https://developer.chrome.com/docs/privacy-sandbox/attribution-reporting/>. <https://developer.chrome.com/docs/privacy-sandbox/attribution-reporting/>. (2022).
- [43] Google. 2022. Privacy Sandbox. https://privacysandbox.com/intl/en_us/. (2022).
- [44] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. 2022. LLAMA: A Low Latency Math Library for Secure Inference. *Proceedings on Privacy Enhancing Technologies* 4 (2022), 274–294.
- [45] Ben-Hur Asa Dror Gideon Guyon Isabelle, Gunn Steve. 2004. Result analysis of the NIPS 2003 feature selection challenge. In *NIPS, 2004*.
- [46] 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).
- [47] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. *IACR Cryptol. ePrint Arch.* 2022 (2022), 207.
- [48] 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.
- [49] 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*.
- [50] Daniel Jurafsky and James H. Martin. 2009. *Speech and Language Processing (3rd Edition)*.
- [51] 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/usexsecurity22/presentation/kelkar>
- [52] 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.
- [53] 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).
- [54] Yehuda Lindell. 2017. How to simulate it—a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography* (2017), 277–346.
- [55] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. 2017. Oblivious neural network predictions via miniomn transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 619–631.
- [56] 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>
- [57] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. 2021. Rabbit: Efficient Comparison for Secure Multi-Party Computation. In *Financial Cryptography and Data Security*, Nikita Borisov and Claudia Diaz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 249–270.
- [58] 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>
- [59] 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).
- [60] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 2165–2182.
- [61] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. SynCirc: Efficient Synthesis of Depth-Optimized Circuits for Secure Computation. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 147–157.
- [62] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. 2022. SecFloat: Accurate Floating-Point meets Secure 2-Party Computation. *Cryptology ePrint Archive* (2022).
- [63] Deevashwer Rathee, Pradeep Kumar Mishra, and Masaya Yasuda. 2018. Faster PCA and Linear Regression through Hypercubes in HELib. *Cryptology ePrint Archive*, Report 2018/801. (2018). <https://eprint.iacr.org/2018/801>.
- [64] 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>
- [65] 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>
- [66] 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.
- [67] Deevashwer Rathee, Thomas Schneider, and KK Shukla. 2019. Improved multiplication triple generation over rings via RLWE-based AHE. In *International Conference on Cryptology and Network Security*. Springer, 347–359.
- [68] Bita Darvish Rouhani, M Sadegh Riazai, and Farinaz Koushanfar. 2018. Deepsecure: Scalable provably-secure deep learning. In *Proceedings of the 55th annual design automation conference*. 1–6.
- [69] Théo Ryffel, Pierre Tholoniati, 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.
- [70] Philipp 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>
- [71] 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).
- [72] Qiang Yang, Yang Liu, Yong Cheng, Yan Kang, Tianjian Chen, and Han Yu. 2019. Federated learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 13, 3 (2019), 1–207.
- [73] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 162–167.
- [74] Jiayuan Ye and Reza Shokri. 2022. Differentially Private Learning Needs Hidden State (Or Much Faster Convergence). *CoRR* abs/2203.05363 (2022).

A DEFINITIONS: IDPF, DCF, DDCF

Incremental Distributed Point Functions. Introduced by Boneh et al. [13], incremental distributed point functions (iDPF) are a generalization of the standard distributed point function (DPF). At a high level, a DPF is a compressed pseudorandom 2-party secret-sharing of a unit vector of length 2^n . More specifically, DPF allows a compressed 2-party secret-sharing of a point function $f_{\alpha,\beta}$ where $\alpha \in \{0, 1\}^n$, $\beta \in \mathbb{F}$, and:

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

Such a secret sharing is represented by a pair of keys (k_0, k_1) where key k_b is the share held by Party P_b . Incremental DPFs (iDPF) are a generalization of DPF which allow compressed sharing of a binary tree with 2^n leaves and a unique special path from root to leaf. I.e., there is a single non-zero path in the tree, ending at leaf α , whose nodes have non-zero values β_1, \dots, β_n . More specifically, iDPF allows a 2-party secret-sharing of an *all-prefix point* function $f_{\alpha,\vec{\beta}}$, where $\alpha \in \{0, 1\}^n$, $\vec{\beta} = ((\mathbb{G}_1, \beta_1), \dots, (\mathbb{G}_n, \beta_n))$, and for each $l \in [n]$:

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

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

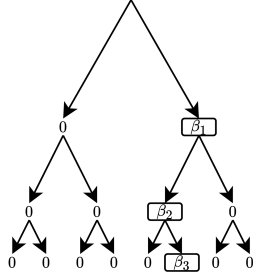


Figure 1: Incremental DPF gives compact secret sharing of values on the nodes of a binary tree with a single non-zero path. In this example, $\alpha = 101$ and the values on the path to the leaf at index α are $\beta_1, \beta_2, \beta_3$. All other nodes are 0. This figure shows the reconstructed secret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

We sometimes allow an iDPF to be evaluated over the empty prefix. We now present iDPF formally, see Figure 1 for more intuition. We closely follow the definitions of Boneh et al. [13], with one major difference being that we expose the 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 $(\text{Gen}^{\text{idpf}}, \text{EvalNext}^{\text{idpf}}, \text{EvalPrefix}^{\text{idpf}})$ such that:

- $\text{Gen}^{\text{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \dots, (\mathbb{G}_n, \beta_n)))$ is a PPT *key generation* algorithm that given security parameter 1^λ and a function description $(\alpha, (\mathbb{G}_1, \beta_1), \dots, (\mathbb{G}_n, \beta_n))$, outputs a pair of keys and public parameters $(k_0, k_1, \text{pp} = (\text{pp}_1, \dots, \text{pp}_n))$. Recall that $\alpha \in \{0, 1\}^n$ represents the index of the leaf at the bottom of the non-zero path while $\beta_1 \in \mathbb{G}_1, \dots, \beta_n \in \mathbb{G}_n$ correspond to the values on the nodes of the non-zero path (apart from the root node). pp includes the public values $\lambda, n, (\mathbb{G}_1, \dots, \mathbb{G}_n)$.
- $\text{EvalNext}^{\text{idpf}}(b, \text{st}_b^{l-1}, \text{pp}_l, x_l)$ 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 .
- $\text{EvalPrefix}^{\text{idpf}}(b, k_b, \text{pp}, (x_1, \dots, x_l))$ 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 $(x_1, \dots, x_l) \in \{0, 1\}^l$, outputs an additive secret sharing of the output value y_b^l .

Next, we present iDPF correctness and security.

Definition A.2. $(\text{Gen}, \text{EvalNext}, \text{EvalPrefix})$ 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 $\vec{\beta} = ((\mathbb{G}_1, \beta_1), \dots, (\mathbb{G}_n, \beta_n))$, level $l \in [n]$, and input prefix $(x_1, \dots, x_l \in \{0, 1\}^l)$, the following requirements hold:
 - EvalNext: $\Pr[y_0^l + y_1^l = f_{\alpha, \vec{\beta}}(x_1, \dots, x_l)] = 1$, where probability is taken over: $(k_0, k_1, \text{pp}) \leftarrow \text{Gen}^{\text{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \dots, (\mathbb{G}_n, \beta_n)))$. And for each $b \in \{0, 1\}$, y_b^l is:
 - (1) $\text{st}_b^0 \leftarrow k_b$
 - (2) **for** $j = 1$ to l :
 - (3) $(\text{st}_b^j, y_b^j) \leftarrow \text{EvalNext}^{\text{idpf}}(b, \text{st}_b^{j-1}, \text{pp}_j, x_j)$
 - (4) **return** y_b^l
 - EvalPrefix: $\Pr[y_0^l + y_1^l = f_{\alpha, \vec{\beta}}(x_1, \dots, x_l)] = 1$, where probability is taken over: $(k_0, k_1, \text{pp}) \leftarrow \text{Gen}^{\text{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \dots, (\mathbb{G}_n, \beta_n)))$. And for each $b \in \{0, 1\}$: $y_b^l \leftarrow \text{EvalPrefix}^{\text{idpf}}(b, k_b, \text{pp}, (x_1, \dots, x_l))$
- **Security.** For every $b \in \{0, 1\}$, there is a PPT simulator Sim_b , such that for every sequence $((\alpha, \vec{\beta})_\lambda)_{\lambda \in \mathbb{N}}$ 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 $_\lambda$: $(k_0, k_1, \text{pp}) \leftarrow \text{Gen}^{\text{idpf}}(1^\lambda, (\alpha, (\mathbb{G}_1, \beta_1), \dots, (\mathbb{G}_n, \beta_n)))$, Output (k_b, pp)
 - Ideal $_\lambda$: Output $\text{Sim}_b(1^\lambda, (n, \mathbb{G}_1, \dots, \mathbb{G}_n))$

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 l th key. This solution would yield key size *quadratic* in the input length n . [13] gives a more direct construction with key size linear in n .

THEOREM A.3 (CONCRETE COST OF iDPF [13]). *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 $\approx n(\lambda + n)$ [15].

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

Dual Distributed Comparison Function (DDCF) DDCF is a variant of DCF and a class of functions $f_{\alpha, \beta_1, \beta_2} : \{0, 1\}^n \rightarrow \mathbb{G}$. 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 $f_{\alpha, \beta_1, \beta_2} = \beta_2 + f_{\alpha, \beta_1 - \beta_2}^<(x)$.

B FUNCTIONALITIES BASED ON FSS

We will now describe some of the functionalities which can be realized efficiently with Function Secret Sharing. We note that gates based on FSS operate on masked inputs and produce masked outputs (instead of standard secret-sharing MPC gates which operate on input shares and produce output shares). Specifically, a masked value x_{mask} for a secret input x is computed as $x_{\text{mask}} := x + r$, 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 [15], 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 $1\{x = y\}$. More formally:

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

Boyle et. al. [18] constructed an equality gate by making two observations. First, $x = y$ can be evaluated by zero-testing $x - y$, i.e. $1\{x - y = 0\}$. 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 $r_0^{\text{in}}, r_1^{\text{in}}$ their masks. Then, equality holds when $x - r_0^{\text{in}} = y - r_1^{\text{in}}$, or equivalently, $x - y = r_0^{\text{in}} - r_1^{\text{in}}$. In other words, we evaluate a DPF function that evaluates to $\beta = 1$ when $\alpha = r_0^{\text{in}} - r_1^{\text{in}}$, 0 otherwise. We present the full construction in Algorithm 4.

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

Input: P_0, P_1 hold $x_{\text{mask}} := x + r_0^{\text{in}}$, where $x_{\text{mask}} \in \mathbb{G}_1$, and $y_{\text{mask}} := y + r_1^{\text{in}}$, where $y \in \mathbb{G}_2$
Output: P_0, P_1 learn a uniform Boolean sharing $b_{\text{mask}} = b \oplus r^{\text{out}}$, where $b := 1\{x = y\}$.

// Part I: Offline Phase.

$\text{Gen}_n^{\text{eq}}(1^\lambda, r_0^{\text{in}}, r_1^{\text{in}}, r^{\text{out}})$:

- 1 Let $r_0^{\text{in}} \in \mathbb{G}_1$ and $r_1^{\text{in}} \in \mathbb{G}_2$.
- 2 Let $\alpha \leftarrow r_0^{\text{in}} - r_1^{\text{in}}$, $\beta = 1$.
- 3 $k'_0, k'_1 \leftarrow \text{Gen}^{\text{DPF}}(1^\lambda, \alpha, \beta)$
- 4 Sample random additive shares $r_0^{\text{out}}, r_1^{\text{out}} \leftarrow \llbracket r^{\text{out}} \rrbracket$.
- 5 Let $k_b = k'_b \parallel r_b^{\text{out}}$.
- 6 **return** (k_0, k_1)

// Part II: Online Phase.

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

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

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 $1\{x < y\}$.

We present the comparison gate of Boyle et. al. [15] in Algorithm 5. This comparison gate requires a single invocation of DDCF, and thus a single invocation of DCF. Note that we slightly modify the

protocol to make it syntactically compatible with our secure comparison. I.e., we (1) write the comparison for $x < y$ rather than [15]'s $x > y$ and (2) the output group is \mathbb{U}_2 instead of \mathbb{U}_N .

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

Input: P_0, P_1 hold $x_{\text{mask}} := x + r_0^{\text{in}}$, where $x_{\text{mask}} \in \mathbb{G}_1$, and $y_{\text{mask}} := y + r_1^{\text{in}}$, where $y \in \mathbb{G}_2$

Output: P_0, P_1 learn a uniform boolean sharing $b_{\text{mask}} = b \oplus r^{\text{out}}$, where $b := 1\{x < y\}$.

// Part I: Offline Phase.

$\text{Gen}_n^{\text{cmp}}(1^\lambda, r_0^{\text{in}}, r_1^{\text{in}}, r^{\text{out}})$:

- 1 Let $y = (2^n - (r_0^{\text{in}} - r_1^{\text{in}})) \in \mathbb{U}_N$ and $\alpha^{(n-1)} = y_{[0, n-1]}$.
- 2 $(k_0^{(n-1)}, k_1^{(n-1)}) \leftarrow \text{Gen}_{n-1}^{\text{DDCF}}(1^\lambda, \alpha^{(n-1)}, \beta_1, \beta_2, \mathbb{U}_2)$, where $\beta_1 = 1 \oplus y_{[n-1]}, \beta_2 = y_{[n-1]} \in \mathbb{U}_2$.
- 3 Sample random $r_0^{\text{out}}, r_1^{\text{out}} \leftarrow \mathbb{U}_N$ s.t. $r_0^{\text{out}} \oplus r_1^{\text{out}} = r^{\text{out}}$.
- 4 For $b \in \{0, 1\}$, let $k_b = k_b^{(n-1)} \parallel r_b^{\text{out}}$.
- 5 **return** (k_0, k_1)

// Part II: Online Phase.

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

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

Multiple Interval Containment (MIC) gate. Boyle et. al. [15] presented an FSS gate for the \mathcal{F}_{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 $b_i = 1\{p_i \leq x \leq q_i\}$.

C SECURE COMPARISON PROTOCOL BENCHMARKS

In Table 6, we benchmark the offline and online communication costs of our new comparison protocol Π_{CMP} for different values of l (the bit length of the comparison inputs). We set $m = 16$ as the parameter in our secure comparison protocol. We compare it against CrypTFlow2 [65] (where we set $m = 4$ as used by the authors⁸), SynCirc [61] and Couteau's protocol [25]. Note that if $l \leq m$, then we set $m = l$. Our offline costs exclude the cost of base oblivious transfers. For SynCirc [61], we could not obtain values for $l = 4, 8, 128$ bits as the paper doesn't report costs for these cases.

D RELATED WORK

There is an extremely large number of works in the field of secure learning, differing across several dimensions. These dimensions include the type of model being computed (linear/logistic/poisson regression, deep neural nets), the way data is distributed across parties (secret-shared, vertically partitioned, horizontally partitioned, federated), and the type of security provided (differentially private with central or local DP, MPC with semi-honest, malicious

⁸Higher values of m in CrypTFlow2 lead to an exponential increase in the online communication cost.

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

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

Preprocessing:

- For all $i \in [q-1]$: $(k_{0,i}^{\text{eq}}, k_{1,i}^{\text{eq}}) \leftarrow \text{Gen}^{\text{eq}}(1^\lambda, r_i^{\text{eq,in}}, s_i^{\text{eq,in}}, r_i^{\text{eq,out}})$. P_0 gets $k_{0,i}^{\text{eq}}, r_i^{\text{eq,in}}$ whereas P_1 gets $k_{1,i}^{\text{eq}}, s_i^{\text{eq,in}}$
- For all $i \in [q]$: $(k_{0,i}^{\text{cmp}}, k_{1,i}^{\text{cmp}}) \leftarrow \text{Gen}^{\text{cmp}}(1^\lambda, r_i^{\text{cmp,in}}, s_i^{\text{cmp,in}}, r_i^{\text{cmp,out}})$. P_0 gets $k_{0,i}^{\text{cmp}}, r_i^{\text{cmp,in}}, [r_i^{\text{cmp,out}}]_0$ whereas P_1 gets $k_{1,i}^{\text{cmp}}, s_i^{\text{cmp,in}}, [r_i^{\text{cmp,out}}]_1$
- $(k_0^{\text{iDPF}}, k_1^{\text{iDPF}}) \leftarrow \text{Gen}^{\text{iDPF}}(1^\lambda, \alpha \oplus r^{\text{eq,out}}, \{\beta_i\}_{i \in [q]})$ where $\alpha = 2^{q-1} - 1, \beta_i = 1, r^{\text{eq,out}} = r_1^{\text{eq,out}} || \dots || r_q^{\text{eq,out}}$. P_0 gets k_0^{iDPF} whereas P_1 gets k_1^{iDPF}

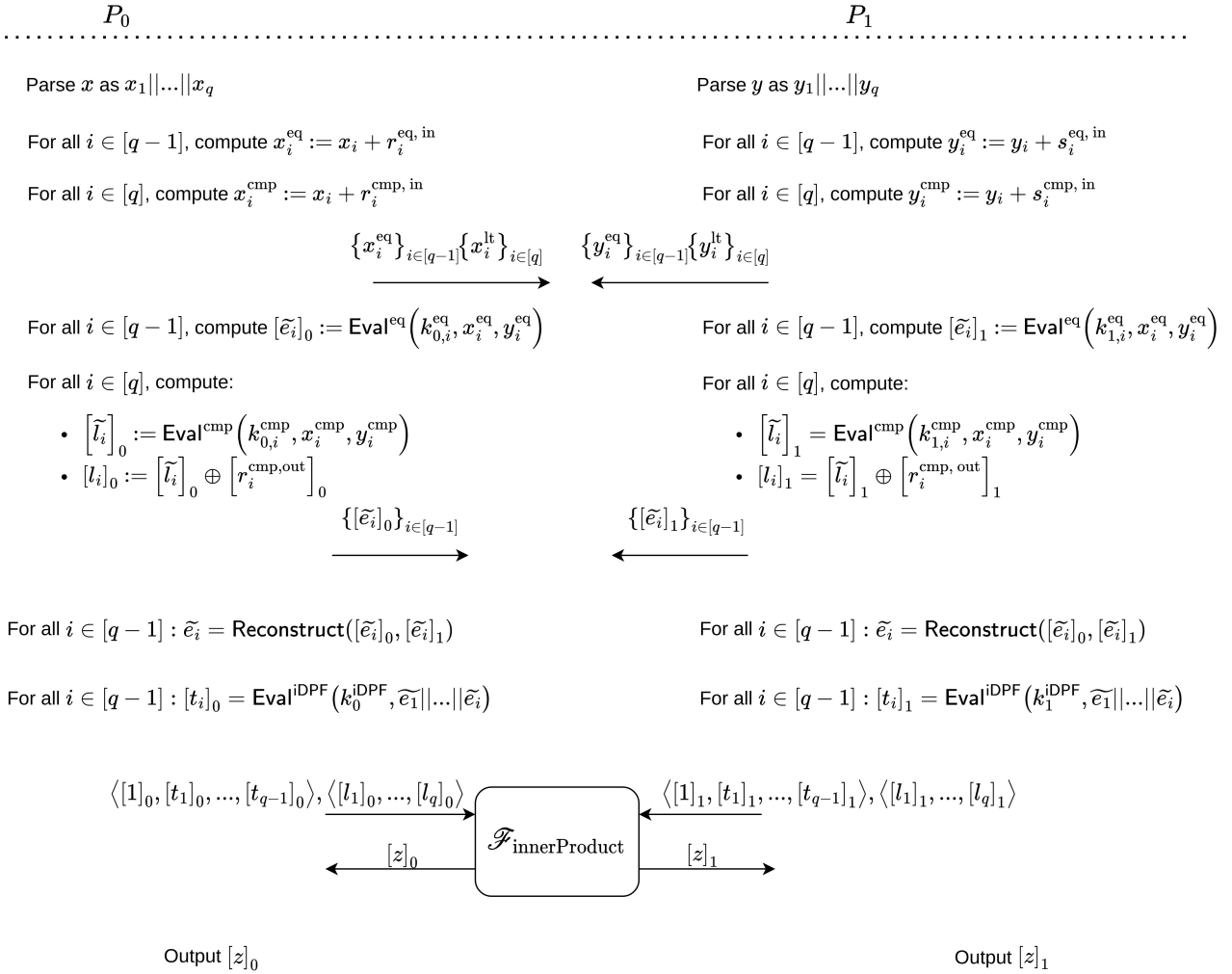


Figure 2: Constant round secure comparison protocol Π_{CMP} for l bit inputs.

security, honest majority, and so on) [2, 12, 19, 21, 32, 37, 51–53, 58, 58, 60, 63, 64, 68, 71, 72].

We focus our discussion on the state-of-the-art works that best match our setting and offer informative comparisons. Specifically,

we focus on works that compute logistic regression using secure multi-party computation in the semi-honest setting, with data secret-shared among the computing parties. We restrict ourselves to works that have 2 or 3 servers only. The number of works in

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

Input: P_0, P_1 hold $x_{\text{mask}} := x + r_0^{\text{in}}$, where $x_{\text{mask}} \in \mathbb{G}_1$, and $y_{\text{mask}} := y + r_1^{\text{in}}$, where $y \in \mathbb{G}_2$
Output: P_0, P_1 learn a uniform arithmetic sharing of $b_{i_{\text{mask}}} = b_i + r_i^{\text{out}}$, where $b_i := \mathbf{1}\{p_i \leq x \leq q_i\}$.

// Part I: Offline Phase.

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

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

// Part II: Online Phase.

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

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

this setting is relatively manageable, and we group them into 3 categories in our discussion: those with a highly accurate sigmoid approximation, those with a coarse sigmoid approximation, and those based on homomorphic encryption.

For works with highly accurate sigmoid approximations, we restrain our detailed comparison to 2 key works, MP-SPDZ [52] and SiRNN [64], which are the state of the art in this area, to our best understanding. Our key difference with both of these works is our approach to sigmoid computation. MP-SPDZ takes the approach of computing exponentiation and division in MPC, which results in a large number of rounds. SiRNN does essentially the same, using novel protocols for each. They use a clever approach of adjusting the fixed-point precision to reduce the costs, and introduce a novel Lookup-Table based exponentiation together with a novel adaptation of Goldschmidt's division. However, both these approaches end up taking a large number of rounds (\approx dozens) because of the complexity of division and exponentiation in MPC. Our key improvement is to achieve high accuracy with a constant number of rounds and less communication using a combination of FSS primitives and a customized sigmoid approach with different approximations computed over different intervals. A detailed experimental comparison can be found in Sections 7.1 and 7.2.

There are also works such as SecureML [58] and ABY2.0 [60] which use a much coarser sigmoid approximation. As a result, both works have worse accuracy on logistic regression. We are focused

on preserving accuracy of logistic regression, and so do not engage in a detailed comparison with these works. We observe briefly that these works rely mainly on AND gates and secure comparisons, and since we propose an improved secure comparison approach, replacing the secure comparisons in [58, 60] with those in our work is likely to offer an improvement. This is discussed further with concrete accuracy numbers in Section 7.2.

There are several works which leverage homomorphic encryption in order to compute Logistic Regression, for example [19] and [21]. These works are interesting because they do not require interaction between parties: one party performs the computation on the entire encrypted dataset. However, this approach comes with a large computational and communication overhead compared to MPC-based approaches (where the communication means the size of the initial encrypted datasets). We see this as a significantly different approach and setting, and so we do not perform a detailed comparison with them in the evaluation section. This is consistent with the approach taken by the MPC-based works we cite above.

Separately from logistic regression and sigmoid computation, there are several related works focused on developing Function-Secret-Sharing or adjacent primitives [15, 18, 60], and using them for machine learning [69]. Our work is influenced by several of these papers, and our techniques can be seen as building on theirs, specifically by combining them with secret-sharing-MPC based approaches.

Another of our key contributions is a new secure comparison. This is rich area of research [15, 25, 27, 36, 60, 61, 65], since comparison is critical for the nonlinear computations in machine learning tasks. We go over the most relevant related works directly in Section 6.1 where we present our new secure comparison protocol. We explain how our approach differs from and builds on these works.

Finally, we discuss an important concurrent work, LLAMA [44], which builds on SiRNN [64] by using FSS to reduce the cost of comparisons in the online phase, similarly to us. Since the work is concurrent, we do not provide a detailed comparison in the evaluation section, but describe the key differences here. One difference lies in the way LLAMA computes sigmoid. While we use a custom approximation to the sigmoid function, LLAMA uses a single spline gate that computes a degree-2 polynomial over each interval, using enough intervals so as to reduce their chosen accuracy metric (ULP error) is small (≤ 4). They use the spline gate from [15], and also run into its key limitation: this spline construction only works on integer values, because the underlying polynomial evaluation does not perform truncated multiplication needed for fixed point values. The LLAMA authors get around this by assuming the ring is large enough to accommodate untruncated multiplication, and then perform a truncation in a single separate round. In contrast, our work gives a new protocol for computing polynomials over secret-shared fixed point numbers that implicitly handles truncation. In this way, we can work with smaller rings, thereby gaining efficiency. We note that our fix only works for fixed-point numbers that have no integer part (i.e. have absolute value < 1), but this turns out to be fine for our use: we only use polynomial approximations for a fixed region of the input where this condition holds.

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

l	Our Approach ($m = 16$)		CrypTFlow2 ($m = 4$) [65]		SynCirc [61]		Couteau16 [25]	
	comm.	rounds	comm.	rounds	comm.	rounds	comm.	rounds
Offline Phase								
4	0.75 KB	80 rounds	0.03 KB	1 round	-	-	0.19 KB	2 rounds
8	1.51 KB	80 rounds	0.08 KB	1 round	-	-	0.44 KB	2 rounds
16	3.02 KB	80 rounds	0.19 KB	1 round	3.43 KB	1 round	1.02 KB	2 rounds
32	9.07 KB	80 rounds	0.43 KB	1 round	8.82 KB	1 round	1.85 KB	3 rounds
64	21.74 KB	80 rounds	0.93 KB	1 round	16.07 KB	1 round	3.83 KB	3 rounds
128	46.71 KB	80 rounds	1.95 KB	1 round	-	-	6.36 KB	3 rounds
Online Phase								
4	8 bits	1 round	20 bits	2 rounds	-	-	30 bits	2 rounds
8	16 bits	1 round	60 bits	3 rounds	-	-	162 bits	6 rounds
16	32 bits	1 round	142 bits	4 rounds	84 bits	3 rounds	308 bits	6 rounds
32	100 bits	2 rounds	308 bits	5 rounds	178 bits	3 rounds	530 bits	12 rounds
64	242 bits	3 rounds	642 bits	6 rounds	316 bits	4 rounds	1120 bits	12 rounds
128	522 bits	3 rounds	1312 bits	7 rounds	-	-	2101 bits	12 rounds

E BLACK-BOX REDUCTION FROM DCF TO IDPF

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

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

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

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

Algorithm 7 shows our construction in detail. In addition to the two DPF keys, the two parties obtain an additional secret-shared value, which can be interpreted as the iDPF evaluation at the empty

prefix. It is used to initialize v_1 . For $i = 2, \dots, n$, v_i is then constructed from v_{i-1} and $\delta = (1-x) \cdot y_i$, where y_i is the iDPF evaluation at level i . Correctness follows by the above recursion argument.

THEOREM E.1 (CONCRETE COST OF DCF USING IDPF). *Given a PRG $G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2\lambda+2}$, 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 E.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 Gen_{DCF} and Eval_{DCF} algorithms can be computed based on the underlying cost of Gen_{iDPF} and $\text{Eval}_{\text{iDPF}}$ 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 Gen_{iDPF} (and hence in Gen_{DCF}) turns out to be $2(n-1)m'$ where $m' = 1 + \lceil m/\lambda \rceil$. In Eval_{DCF} , we perform an $\text{EvalNext}_{\text{iDPF}}$ 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.

Comparison with original DCF construction. Boyle et al. [15] presented a direct construction of DCF by carefully modifying and making non black-box changes to a prior DPF construction [17]. We provide a conceptually simpler DCF construction by making black-box use of iDPFs (which have a richer structure than DPF).

Algorithm 7: DCF to iDPF reduction

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

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

 $\text{Eval}_n^{\text{DCF}}(b, (k_b, \llbracket \beta_1 \rrbracket^b), \text{pp}, x) :$

- 1 Let $x = x_1, \dots, x_n \in \{0, 1\}^n$ be the bit decomposition of x
 - 2 Let $v_1 = (1 - x_1) \cdot \llbracket \beta_1 \rrbracket^b, \text{st}_1 = k_b$
 - 3 **for** $i = 2$ **to** n **:**
 - 4 $(\text{st}_i, y_i) \leftarrow \text{EvalNext}_{n-1}^{\text{iDPF}}(b, \text{st}_{i-1}, k_b, x_1 \dots x_{i-1})$
 - 5 $v_i \leftarrow v_{i-1} + (1 - x_i) \cdot y_i$
 - 6 **return** v_{n-1}
-

As an added benefit, the key size of our DCF construction is smaller than Boyle et al. [15] by $\lambda + m + 2$ bits. In terms of computation, our construction doesn't require any PRG evaluations at the first bits, and so it saves $m' = \lfloor m/\lambda \rfloor$ PRG evaluations.

F EFFICIENT 2PC GENERATION OF FSS KEYS

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

A straight-forward way to generate these keys in MPC is to implement Gen^{iDPF} using a generic MPC compiler. This, however, has the drawback of requiring PRG calls inside the MPC, making this approach inefficient in practice. [29] present a construction that does not require secure PRG evaluations. While, it comes at a computation cost that is linear in the domain size (i.e., exponential in the input size), and its round complexity is linear in the input size, it is still efficient enough in our case, where the domain for any single DCF is small.

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

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

Also note that while previous work [15] 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 [15]. The main challenge for this construction is the fact that in order to compute the value correction words included in the DPF keys, the parties need to identify which one of them holds share 1 and which one holds share 0 of the control bit corresponding to the node on the evaluation path at every level. There are 2^l nodes at level l , and each party can locally evaluate its shares for all nodes, but the parties do not know which node is on the evaluation path.

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

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

We present the details our Doerner-shelat construction for iDPFs with arbitrary output groups in Algorithm 8. The two parties hold secret shares of α and $\{\beta_i\}_{i \in [n]}$, and would like to generate the iDPF keys for $f_{\alpha, \{\beta_i\}_{i \in [n]}}$. In order to get a protocol for distributed DCF key generation, observe that we only need to compute shares β_1, \dots, β_n in Algorithm 7 given $\alpha_1, \dots, \alpha_n$ and β . As $\beta_i = \alpha_i \cdot \beta$, this reduces to n parallel calls to \mathcal{F}_{MUX} . Finally, observe that in groups where $-x = x$ (such as boolean sharing), $\llbracket W_{CW}^0 \rrbracket = \llbracket W_{CW}^1 \rrbracket$ in Step 11, and so the last $\mathcal{F}_{\text{MUX}2}$ call can be saved in that case, making the entire second MPC linear.

G ADDING DIFFERENTIAL PRIVACY

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

However, we consider here the case where the trained regression model is released to a single party and the goal is to guarantee DP for the model parameters. Since our training construction used SGD, we will also use the DP-SGD approach introduced by Abadi et al. [1] for general SGD ML training and the instantiation of Jayaraman et al. [49] for the setting of logistic regression presented in Algorithm 9. Jayaraman et al. [49] provides a two party computation protocol for secure training of logistic regression when the input data is

Algorithm 8: 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 $G : \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2(\lambda+1)}$ and Convert : $\{0, 1\}^\lambda \rightarrow \{0, 1\}^{\lambda+1}$ be PRGs.

Each party P_b performs the following:

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

$$[[s^R]] \leftarrow [[s_0^R]] \oplus [[s_1^R]]$$

$$[[s^L]] \leftarrow [[s_0^L]] \oplus [[s_1^L]]$$

$$[[s_{CW}]] \leftarrow \mathcal{F}_{\text{MUX2}}([s^R], [s^L], [\alpha_l])$$

$$[[t_{CW}^L]] \leftarrow [[t_0^L]] \oplus [[t_1^L]] \oplus [\alpha_l] \oplus [1]$$

$$[[t_{CW}^R]] \leftarrow [[t_0^R]] \oplus [[t_1^R]] \oplus [\alpha_l]$$

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

- 6 For all $w \in \{0, 1\}^{l-1}$, set $\tilde{s}_b^{w,0} || \tilde{s}_b^{w,1} \leftarrow (s_b^{w,L} || s_b^{w,R}) \oplus t_b^w \cdot (s_{CW} || s_{CW})$
- 7 For all $w \in \{0, 1\}^{l-1}$, set $t_b^{w,0} || t_b^{w,1} \leftarrow (t_b^{w,L} || t_b^{w,R}) \oplus t_b^w \cdot (t_{CW}^L || t_{CW}^R)$
- 8 For all $w \in \{0, 1\}^l$, set $s_b^w || W_b^w \leftarrow \text{Convert}(\tilde{s}_b^w)$
- 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:

$$[[t^*]] \leftarrow 1 \oplus \tau_0^1 \oplus \tau_1^1 \oplus (\tau_0^0 \cdot \tau_1^0)$$

$$[[W_{CW}^0]] \leftarrow [[\beta_l]] - W_0^l + W_1^l$$

$$[[W_{CW}^1]] \leftarrow -[[\beta_l]] + W_0^l - W_1^l$$

$$[[W_{CW}]] \leftarrow \mathcal{F}_{\text{MUX2}}([W_{CW}^0], [W_{CW}^1], [[t^*]]).$$

- Output W_{CW} to both

- 12 Set $CW^l \leftarrow s_{CW} || t_{CW}^L || t_{CW}^R || W_{CW}$
 - 13 Output $k_b \leftarrow s_b^0 || CW^1 || \dots || CW^n$
-

horizontally partitioned between the two parties. We adapt their framework to the setting where the input is fully secret-shared between the two parties.

In Algorithm 10 we give the pseudocode for implementing the DP-SGD algorithm in MPC. The MPC protocol is similar to the non-DP algorithm in Algorithm 1, except in 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 et al. [30] and Champion et al. [20].

If we only want to guarantee DP from the output of the secure logistic regression training, then we can reveal the DP gradient update to the two computation parties as shown in Algorithm 10. This would enable some efficiency optimization replacing a secure matrix multiplication with a plaintext matrix multiplication. While this approach still provides DP for the output, it is not known what is the exact privacy comparison between revealing only the final DP output model and all intermediate DP gradient updates. However, recent works [22, 74] show that keeping the DP-SGD intermediate states hidden allows for faster convergence and spending less privacy budget for strongly convex loss functions for noisy stochastic gradient descent. Our DP secure computation training algorithm supports hiding these intermediate states at the same online communication cost.

Algorithm 9: DP SGD

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

Private inputs: Dataset X, y having k features

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

Jayaraman et al. [49] 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 [49], adding the noise iteratively to the gradient or directly to the output may have different impact on the accuracy of the final model depending on the setting, though adding noise iteratively generally results in more accurate models. We are able to support both options between Algorithms 1 (with output-perturbation at the end of training) and Algorithm 10.

H 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 and MP-SPDZ (piecewise approximation of 5 pieces) 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

Algorithm 10: DP-SGD Logistic Regression Protocol

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

Private inputs: Secret-shared dataset $\llbracket X \rrbracket \in R^{n \times k}$ and labels $\llbracket y \rrbracket \in R^n$. 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 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 $\llbracket X_B \rrbracket \leftarrow \llbracket X_{i..j} \rrbracket$
 - 7 $\llbracket u \rrbracket \leftarrow \llbracket X_B \rrbracket \cdot w_{t-1}$
 - 8 $\llbracket s \rrbracket \leftarrow \mathcal{F}_{\text{Sigmoid}}(u)$
 - 9 $\llbracket d \rrbracket \leftarrow \llbracket s \rrbracket - \llbracket y_{i..j} \rrbracket$
 - 10 $\llbracket g \rrbracket \leftarrow \mathcal{F}_{\text{matMult}}(\llbracket X_B^T \rrbracket, \llbracket d \rrbracket)$
 - 11 $\llbracket w_t \rrbracket \leftarrow \llbracket w_{t-1} \rrbracket - (\alpha/B) \cdot (\llbracket g \rrbracket + \lambda \cdot \llbracket w_{t-1} \rrbracket) + \llbracket r_t \rrbracket$
 - 12 $w_t \leftarrow \text{Reconstruct}(\llbracket w_t \rrbracket)$
 - 13 **return** w_T .
-

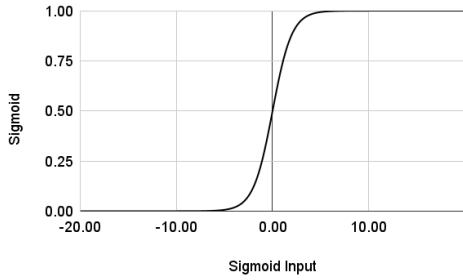


Figure 3: Plaintext Sigmoid Function

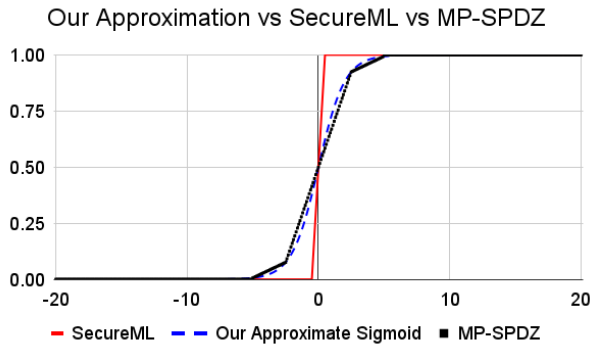
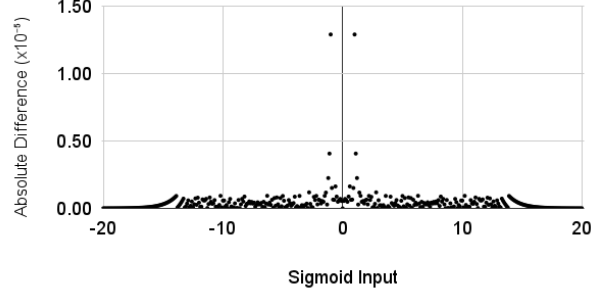


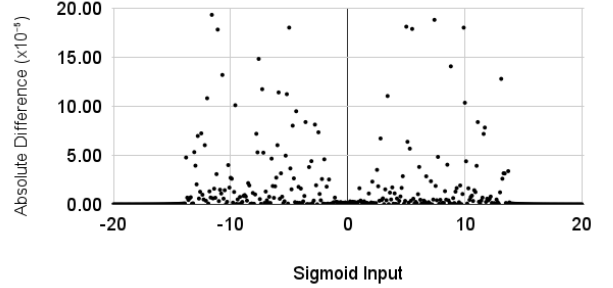
Figure 4: Plaintext Comparison of V1 sigmoid to SecureML's and MP-SPDZ's approximations.

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.

Plaintext Approximation in Fixed Point



2PC Sigmoid V1



2PC Sigmoid V2

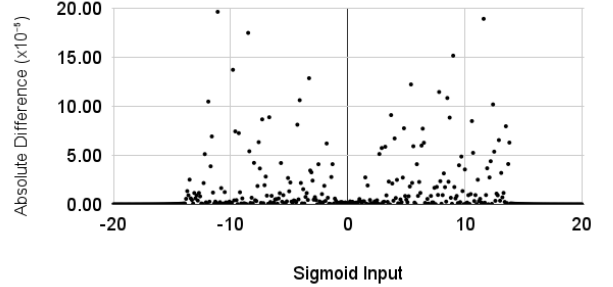


Figure 5: Absolute difference error ($\times 10^{-5}$) of our sigmoid approximation in plaintext fixed point representation (top), with trusted offline setup run in 2PC (middle), and with distributed offline setup run in 2PC (bottom).

I 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 logistic regression training comes out to be $2nk + T \cdot \lfloor n/B \rfloor (2k + 2B)$ elements.

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

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

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

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).

J 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 $\llbracket x \rrbracket$ 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 $\llbracket x \rrbracket$ to get $\llbracket a_i x + b_i \rrbracket$ using local operations. For n intervals, P_0, P_1 hold:

$$\llbracket a_1 x + b_1 \rrbracket, \dots, \llbracket a_n x + b_n \rrbracket$$

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

$$\llbracket d \rrbracket = \llbracket 0, 0, 1, 0, \dots, 0 \rrbracket$$

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:

$$\llbracket a_t \rrbracket \leftarrow \llbracket d_1 \rrbracket a_1 + \dots + \llbracket d_n \rrbracket a_n$$

$$\llbracket b_t \rrbracket \leftarrow \llbracket d_1 \rrbracket b_1 + \dots + \llbracket d_n \rrbracket b_n$$

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

$$\llbracket a_t x + b_t \rrbracket$$

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

K FAILURE PROBABILITY

The non-interactive fixed point truncation protocol from [58] and the single round exponentiation protocol from [51] are probabilistic i.e. with some probability, that can be made arbitrary low by increasing the ring size, the output of these protocols can be incorrect. Since we use these two primitives as sub-protocols in our logistic regression protocol, it also induces an error probability on the overall training algorithm.

Each invocation of the non-interactive fixed point truncation protocol [58] has an error probability of $p_{\text{trunc}} = \frac{2^{w+1}}{2^l}$. We use this as a subprotocol in every instance of fixed point multiplication to adjust the scale. In each matrix multiplication, we truncate once after the accumulation (i.e. for multiplying matrix M_1 with matrix M_2 , we do the usual Beaver multiplication without truncation to get a matrix M_3 , and then truncate each element of M_3 by appropriate scale). This ensures (as pointed out in [58]) that the probability of errors introduced due to truncation is low and the error union bound scales proportional to $|M_3|$ (instead of being proportional to $|M_1| \cdot |M_2|$ which would have been the case if we truncate before accumulating).

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

Additionally, each exponentiation protocol from [51] has a failure probability $p_{\text{exp}} \approx \frac{2^{w+1}}{2^l}$. We invoke the exponentiation protocol twice per input in the batch, hence a total of $2B$ exponentiations per iteration.

Now we can bound the total failure probability of one iteration of training by $2Bp_{\text{exp}} + (131B + k)p_{\text{trunc}}$ using union bound. Assuming t is the total number of iterations in the training and plugging the values of p_{exp} and p_{trunc} , we get a total failure probability bound across all iterations as $(133B + k) \cdot \frac{2^{w+1}}{2^l} \cdot t$. Compared to SecureML [58], our failure probability only reduces the security by ≈ 7 bits while providing a much more accurate training (due to our better sigmoid approximation).

L SEMI-HONEST SECURE TWO PARTY COMPUTATION

The following description of semi-honest two-party computation is standard in the literature and has been taken from [54].

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

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

Privacy by Simulation. As expected, we wish to formalize the idea that a protocol is secure if whatever can be computed by a party participating in the protocol can be computed based on its input and output only. This is formalized according to the simulation paradigm by requiring the existence of a simulator who generates the view of a party in the execution. However, since the parties here have input and output, the simulator must be given a party's input and output in order to generate the view. Thus, security here is formalized by saying that a party's view in a protocol execution be simulatable given its input and output. This formulation implies that the parties learn nothing from the protocol execution beyond what they can derive from their input and prescribed output.

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

Definition of Security. We begin with the following notation:

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

Definition L.1. Let $f = (f_1, f_2)$ be a functionality. We say that a protocol π securely computes f in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time algorithms \mathcal{S}_1 and \mathcal{S}_2 such that

$$\begin{aligned} & \{(\mathcal{S}_1(1^\lambda, x, f_1(x, y)), f(x, y))\}_{x, y, \lambda} \\ & \stackrel{c}{=} \\ & \{(\text{view}_1^\pi(x, y, \lambda), \text{output}^\pi(x, y, \lambda))\}_{x, y, \lambda} \end{aligned}$$

, and

$$\begin{aligned} & \{(\mathcal{S}_2(1^\lambda, x, f_2(x, y)), f(x, y))\}_{x, y, \lambda} \\ & \stackrel{c}{=} \\ & \{(\text{view}_2^\pi(x, y, \lambda), \text{output}^\pi(x, y, \lambda))\}_{x, y, \lambda} \end{aligned}$$

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

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

In the context of 2PC, we are interested in computing a functionality $\mathcal{F}_{\hat{f}}$ which takes in secret shares of some fixed-point input $x \in \mathbb{F}$ and outputs a secret-sharing of $\hat{f}(x)$. In our paper, we describe protocols $\Pi_{\hat{f}}$ for securely computing such fixed-point functionalities where the description of \hat{f} is sometimes implicit in the protocol description. The approximate nature of \hat{f} w.r.t. to f comes from the fact that: 1) we are only allowed to perform operations on fixed point values, 2) the intermediate computations in $\Pi_{\hat{f}}$ might have some error. The latter case is true in our protocols (such as Secure Powers, Secure Polynomial, Secure Sigmoid) as it relies on the non-interactive approximate truncation protocol from prior work [58]⁹ as a subprotocol.

Security proof (sketch). Our final protocol for secure logistic regression invokes subprotocols for secure matrix multiplication and secure sigmoid. The protocol for secure sigmoid in turn invokes additional subprotocols such as secure spline, secure exponentiation, secure polynomial evaluation and secure MIC (Multiple Interval Containment). Let $\Pi_{\hat{f}}$ be a particular subprotocol where \hat{f} is the approximate function that it is computing. For example, $\Pi_{\hat{f}}$ can be our secure polynomial protocol from Section 5.3 where \hat{f} would be the (implicit) approximate polynomial evaluation function. All of our protocols are designed in the preprocessing model where

⁹Although it is possible to use exact truncation protocols from the literature at the cost of increased rounds of interaction, we made a choice to stick with the non-interactive approximate truncation protocol [58] for efficiency reasons and the fact that this approximation only introduces atmost 1 bit of error in the least significant bit of the fractional part which can be offset by increasing the precision scale.

the protocol Π depends on some correlated randomness available from the offline phase. Let $\mathcal{F}_{\hat{f}}$ denote a functionality which takes as input secret shares of some fixed-point value $x \in \mathbb{F}$ and outputs a secret-sharing of $\hat{f}(x)$. Let $\mathcal{F}_{\hat{f}}^{\text{offline}}$ denote the setup functionality which generates the preprocessing material required for securely computing \hat{f} and sends it to the parties.

We can argue that $\Pi_{\hat{f}}$ securely computes $\mathcal{F}_{\hat{f}}$ in the $\mathcal{F}_{\hat{f}}^{\text{offline}}$ hybrid model. We note that all of our subprotocols in this paper, except for secure comparison, make use of the non-interactive secure truncation protocol from SecureML[58] for performing fixed point multiplication. Therefore, for proving security of such subprotocols, we will abstract away the truncation protocol as an ideal functionality $\mathcal{F}_{\text{truncate}}$ and prove security in a hybrid model where $\mathcal{F}_{\text{truncate}}$ is available in addition to $\mathcal{F}_{\hat{f}}^{\text{offline}}$. The exact function implemented by $\mathcal{F}_{\text{truncate}}$ is implicit in the protocol description of SecureML[58].

For proving security, we need to construct a simulator $\mathcal{S}_{\hat{f}}$ which simulates the protocol in the ideal world. This needs to be done in two steps - first, we need to simulate the preprocessing material, and secondly, we need to simulate the online protocol interaction. For subprotocols which are not based on FSS (such as secure matrix multiplication, secure polynomial evaluation and secure exponentiation), the preprocessing material can be simulated honestly by having the $\mathcal{S}_{\hat{f}}$ internally execute $\mathcal{F}_{\hat{f}}^{\text{offline}}$ and sending the adversary its part of the preprocessing material. For simulating the preprocessing material of FSS-based protocols (such as secure spline, secure comparison, secure MIC), we can execute a simulator \mathcal{S}_{FSS} for generating fake FSS keys. Such a simulator is already provided in prior works for DPF [17], DCF [15] and iDPF [13] and its security relies on the assumption of One Way Function (OWF). We note that in all of our subprotocols, simulating the view of the adversary in the online protocol is straightforward as parties only exchange uniformly random elements which can be perfectly simulated. In the overall protocol for secure logistic regression, we can invoke the simulator for all of the underlying subprotocols in order to simulate the view of adversary. For technical reasons, as pointed out in [10], a resharing step needs to be added in the end of secure logistic regression protocol to upgrade from view-indistinguishability to indistinguishability of the joint distribution of adversary's view and honest party's output. Practically, this can be cheaply done by having P_0 and P_1 locally sample a random value r_0 and r_1 respectively from the domain of the weight vector, sending it across to the other party, and then adding $r_0 + r_1$ locally to their output shares.

M SPLINE DETAILS

We approximate sigmoid on $[0, 1)$ by splitting the interval into m equally sized intervals. To do so, we define a series of $m + 1$ points $\{\alpha_i\}_{i \in [m+1]}$ where $\alpha_1 = 0 < \alpha_2 < \dots < \alpha_m < \alpha_{m+1} = 1$ and $\alpha_{i+1} - \alpha_i = \frac{1}{m}$ for all $i \in [m]$. Then we define m linear univariate polynomials whose coefficients we denote as $\{a_i, b_i\}_{i \in [m]}$. The sigmoid approximation is then computed as follows:

$$\text{Sigmoid}(x) = \begin{cases} a_1x + b_1 & \alpha_1 \leq x < \alpha_2 \\ a_2x + b_2 & \alpha_2 \leq x < \alpha_3 \\ \dots & \dots \\ a_{m-1}x + b_{m-1} & \alpha_{m-1} \leq x < \alpha_m \\ a_mx + b_m & \alpha_m \leq x < \alpha_{m+1} \end{cases}$$

The coefficient values for the i^{th} interval are computed by interpolating a line between the coordinates $(\alpha_i, \sigma(\alpha_i))$ and $(\alpha_{i+1}, \sigma(\alpha_{i+1}))$, where $\sigma(\cdot)$ denotes the exact sigmoid function. Table 7 describes the specific values for the coefficients that were used in our experiments for $m = 10$ intervals. Note that the values of the coefficients in Table 7 are the exact real number values which are converted into a fixed point representation when performing the secure computation.

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

Table 7: Spline parameters for instantiating sigmoid approximation

The value $m = 10$ was decided by evaluating the spline approximation for different values of m and checking the average ULP error (defined w.r.t. a fixed fractional scale of $s = 20$ bits), as used in [62], for sigmoid inputs drawn uniformly at random in $[0, 1)$. For $m = 10$, we obtained an average ULP error of 46 which corresponds to an absolute error of less than 0.00005. This seemed to be a reasonable cutoff for the accuracy of logistic regression hence we made the choice of $m = 10$. Note that the online communication cost and rounds of our secure sigmoid protocol is independent of m .

As mentioned in Section 4.1, we use the spline-based approximation only on the interval $[0, 1)$ (which automatically also provides an approximation on $(-1, 0]$ due to the symmetric nature of sigmoid curve) and use exponentiation combined with Taylor-series based approximation on the interval $[1, \infty)$ (and symmetrically on $(-\infty, -1]$). We chose this approach rather than using only splines to approximate sigmoid on the entire $(-\infty, \infty)$ interval, as has been done in prior works [58], as it would require a large number of spline intervals¹⁰ and potentially higher degree splines for the approximation to work well, thus increasing the cost of protocol.

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

In the v1 setting (i.e. secure sigmoid with trusted offline setup), this increase in cost would be reflected in the amount of FSS pre-processing material that parties need to store. Specifically, for m intervals and d degree splines, the cost would be proportional to md without accounting for fixed-point related issues [15].

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