Secure Poisson Regression

Mahimna Kelkar* Phi Hung Le† Mariana Raykova‡ Karn Seth§

Abstract

We introduce the first construction for secure two-party computation of Poisson regression, which enables two parties who hold shares of the input samples to learn only the resulting Poisson model while protecting the privacy of the inputs.

Our construction relies on new protocols for secure fixed-point exponentiation and correlated matrix multiplications. Our secure exponentiation construction avoids expensive bit decomposition and achieves orders of magnitude improvement in both online and offline costs over state of the art works. As a result, the dominant cost for our secure Poisson regression are matrix multiplications with one fixed matrix. We introduce a new technique, called correlated Beaver triples, which enables many such multiplications at the cost of roughly one matrix multiplication. This further brings down the cost of secure Poisson regression.

We implement our constructions and show their extreme efficiency. Our secure exponentiation for 20-bit fractional precision takes less than 0.07ms. One iteration of Poisson regression on a dataset with $10,000$ samples with 1000 binary features, requires 16.47s offline time, 23.73s online computation and 7.279MB communication. For several real datasets this translates into training that takes seconds and only a couple of MB communication.

1 Introduction

Privacy preserving computation technologies aspire to enable a wide range of modern computations used to analyze data, while providing strong privacy guarantees for the input data, which is often partitioned across multiple parties. Approaches based on cryptographic techniques for secure multiparty computation (MPC) have maintained the invariant of strong privacy guarantees while progressively supporting more complex functionality. In recent years, such approaches have taken on some of the most powerful available tools for data analysis which come from machine learning (ML). These tools bring functionalities with new levels of complexity to be supported in secure computation.

Existing MPC systems that support ML computations have mostly considered algorithms that aim to solve classification tasks, the most prominent of which are neural networks [4, 23, 27, 32]. In this work, we focus on a different type of computation: modeling Poisson processes. These processes are used to represent counts of rare independent events which happen at random but at a fixed rate.

*Cornell University and Cornell Tech. mahimna@cs.cornell.edu. Part of the work was done while the author was an intern at Google LLC.

†George Mason University. ple13@gmu.edu. Part of the work was done while the author was an intern at Google LLC.

‡Google LLC. marianar@google.com

§Google LLC. karn@google.com

In such a process, the rate of events can be characterized by an underlying Poisson distribution. Poisson distributions are used to describe processes across many life and social sciences. Examples include the number of bacteria over time in a petri dish, the number of mutations of a strand of DNA of a certain length, the number of photons arriving at a telescope, the number of losses and claims in insurance policies in a certain period of time, the number of calls arriving at a call center per minute, and the number of purchases a user makes after being shown an online advertisements.

It is common to model response variables that follow the Poisson distribution by assuming their dependence on a set of explanatory (predictor) variables. Specifically, it is often assumed that the logarithm of the response variable is some linear combination of the explanatory variables. In this setting, the relationship between a response variable and the corresponding explanatory variables can be learned using Poisson regression. When the explanatory variables represent features which are conjectured to affect the counts, the regression model can be interpreted as uncovering the statistical significance of the effect of different features on the response variable. For example, Poisson regression has been used to model the dependence of the mortality rate from lung cancer on the age and smoking habits of people [16], the frequency at which voters engage in political discussion as a function of the method they use for voting (e.g., in person or by mail), their demographics, political affiliations, news exposure and others [28], the number of truck accidents depending on the geometrical characteristics of the road and the traffic patterns [31], the effect of age, gender, preexisting conditions such as diabetes and obesity on the mortality rate from COVID-19 [30], predicting the number of payment defaults in credit scoring based on age, income, number of dependents, home ownership, type of employment [21], and the number of purchases that users make influenced by online advertisements they have been shown [29].

Traditionally, Poisson regression is performed by collecting all the examples (observed response variable counts together with the observed explanatory variable values), and performing training. However, in many of the above examples the information reflected in the predictor variables comes from different sources that hold health and financial data, which is highly sensitive information that is often subject to privacy regulations. Thus, while the final output model could be a useful tool for drawing insights about the underlying processes and events, providing the input data in the clear for the training is not an option. In this paper, we propose a solution that enables the computation while keeping all the inputs hidden from the parties performing the computation, revealing only the final Poisson regression model.

We introduce a system for secure computation that enables two parties who hold different parts of the training samples for Poisson regression to compute the final Poisson model. We assume the most general setting where the two computation parties hold cryptographic shares of the input training data, and obtain cryptographic shares of the resulting model. This representation can capture any partition of the input among the parties and also enables computation with the output model that does not reveal the model parameters to either party. Our new two party computation construction for Poisson regression leverages several new constructions for its building block components that offer improved efficiency. These functionalities have numerous uses beyond Poisson regression and thus are of independent interest as tools for secure computation.

Secure Exponentiation. One of the most challenging steps in the training of Poisson regression is the exponentiation computation. This step constitutes the nonlinear portion of Poisson regression, and is not part of existing MPC implementations for ML functionalities. Nonlinear computations have traditionally been very challenging for secure computation techniques, since

such techniques are generally better suited for evaluating linear functions or low-degree polynomials. Indeed, in existing MPC frameworks for ML functionalities [4, 23], the nonlinear components of the computation (e.g., the logistic function or the RELU function) are the core challenge that these works solve, and they contribute the most significant part of the cost of the final constructions. Adding to the challenge is the fact these nonlinear functions work on real numbers, which are quite difficult to support in MPC. Most approaches replace the nonlinear function with an approximation such as a low-degree polynomial or a piecewise linear function, which is easier to evaluate in MPC. However, such approximations could lead to significant degradation in the quality of the learned model (i.e. higher model error compared to training in the clear), and thus, the evaluation of the resulting constructions needs to consider jointly efficiency and accuracy.

In our work, we present a new construction for secure fixed-point exponentiation. It leverages a close approximation of the exact function with high precision that enables a significant efficiency improvement compared to existing constructions. In particular, all existing secure exponentiation approaches rely either on inaccurate polynomial approximations, or on bit decomposition of the exponent, which comes with a significant computation and communication cost. Our techniques avoid this multi-round computation step by leveraging ideas that enable the parties to obtain approximate multiplicative shares of the output only with local operations. We can control the accuracy and failure probability by appropriate parameter adjustment, only assuming knowledge of bounds on the input range. These bounds arise naturally in the context of Poisson regression. We introduce a new way to split the computation of the exponentiation into computation that depends only on the integer part of the exponent and computation that depends only on the fractional part of the exponent. Furthermore, we provide a novel way to combine the two computations with only local operations to obtain multiplicative shares of the output. Our only communication requirement is to transform the multiplicative shares of the output of the exponentiation into additive shares, which can be used for any further computation. For this, we leverage an existing protocol from Ghodosi et al. [17] that relies on a small amount of offline precomputation and a single round of online computation.

Our resulting protocol achieves orders of magnitude improvement on both the online throughput and the offline cost compared with state of the art works [8, 10]. In terms of accuracy, we can tune the parameters of our construction so that the output is arbitrarily close to the "true" exponentiation on the values in the clear without significant efficiency penalty (for example we can go from error 0.006% to error 0.0002% with 5 additional bits of precision). Our construction is so efficient that the nonlinear component of our Poisson regression protocol is no longer the cost bottleneck, and no longer degrades the quality of the computation, which stands in stark contrast to other works in the area of secure ML.

Optimized Secure Matrix Multiplication. The computation of Poisson regression extensively uses matrix multiplication. Current state of the art techniques for secure matrix multiplication was proposed by Mohassel and Zhang [23]. The authors generalized the addition and multiplication operations on shared values to shared matrices and proposed a way to generate Beaver triples for matrix addition and multiplication. Their technique also relies on the precomputation of Beaver triples [12] (for matrices) which optimizes the online cost for the multiplication. For each secure matrix multiplication, one random Beaver triple is generated and consumed during the online phase.

We make the observation that the matrix multiplication operations used in the Poisson regression training have a specific structure that can be exploited to further optimized the cost of the

matrix multiplications: the same matrix is used in many multiplications with many different matrices. While we can use independently generated Beaver triples for each multiplication, we show a more efficient way to precompute multiplication triples which takes advantage of the structure of the online matrix multiplications. We call these correlated Beaver triples, and they enable multiple online multiplications with the same matrix. Using correlated Beaver triples results in improvements in both the offline cost for their precomputation as well as the cost for the online matrix multiplications. In particular, K correlated matrix multiplications of dimensions $n \times m$ and $m \times k$ require $nm + K(mk + 2nk)$ ring elements for offline precomputation and $2(nm + Kmk)$ for online work. For the same setting, SecureML [23] needs $K(nm + mk + 2nk)$ ring elements for offline computation and $2K(nm + mk)$ for online phase. In the setting of Poisson regression, the most significant factor in these costs is $\Theta(nm)$ as $k = 1$ typically, which results in $\frac{mn+m+2n}{mn/K+m+2n}$ and $\frac{n+1}{n/K+1}$ factors improvements for the offline and online phase respectively. When $m+2n \ll mn$ and $1 \ll n$, the gain is very close to K, or in other words, the cost to perform K secure multiplications is almost the same as the cost to perform one. Thanks to our very efficient secure exponentiation, the dominant cost (both computation and communications) in the protocol for secure Poisson regression comes from secure matrix multiplication operations (more than 90% of the cost). Consequently, the use of correlated Beaver triples translates directly to a significant overall improvement of the cost of the whole secure Poisson regression protocol.

Experimental Results. We implemented all our constructions and provide detailed benchmarking. Our secure exponentiation construction achieves significant efficiency improvements over existing approaches. Our implementation uses 127-bit modulus for the computation field, which suffices for our Poisson regression evaluation. For this modulus, secure exponentiation with shared exponent that has 20-bit precision takes only 0.055ms online time with 0.013ms preprocessing offline time. In comparison, the constructions of Alisari et al. [8] requires 96ms per exponentiation when batching 100,000 exponentiations together. The more recent implementation in SCALE-MAMBA [10] uses a larger 245-bit modulus and a 40-bit precision, partly motivated by numerical instability for smaller sizes. Our construction does not have such instabilities and achieves online throughput that is 200x more efficient. The improvement in the offline computation is even greater, where our protocol has 2000x improvement in offline communication and $500,000$x improvement in offline computation.

We evaluate our secure Poisson regression implementation using three real datasets: Somoza's data on infant and child survival in Colombia, time to Ph.D. data, and data on the three-year survival status of breast-cancer patients [1]. We further evaluate the scalability of our system using larger synthetic datasets. The accuracy of our secure regression is essentially identical to that of plaintext computation of the regression. The training for each of the three datasets takes less than 0.55s online time and 0.35s offline time, and the communication is less than 2MB. The computation and communication overhead for our construction scales roughly linearly with the size of the training data. For a dataset with $10,000$ samples with 1000 binary features, the training requires 16.47s offline time, 23.73s online computation and 7.279MB communication per iteration for a 127-bit modulus, and 20 bits for fractional precision. We also estimate the efficiency for secure Poisson regression for datasets used to predict COVID-19 case fatality rate, credit default rates and ad campaign conversion rates (see Section 8).

2 Preliminaries and Background

We start with some preliminaries and introduce standard background techniques on regression, and secure computation.

Basic notation. $\mathbb{Z}$ denotes the integers and $\mathbb{R}$ denotes the real numbers. $\mathbb{Z}_N$ denotes the ring of integers modulo N. For a prime q, $\mathbb{F}_q$ denotes the field with q elements, and $\mathbb{F}_q^\times$ denotes its multiplicative group. We use bold uppercase letters (e.g., $\mathbf{M}$) to denote matrices and bold lowercase letters (e.g., $\mathbf{u}, \mathbf{v}$) to denote (row) vectors. Throughout the paper, e denotes Euler's constant. In some places, we abuse function notation slightly, and write $f(\mathbf{u})$ to denote the vector resultant from applying f to each element in $\mathbf{u}$ separately.

2.1 Poisson Regression and Gradient Descent

In this section, we provide a brief overview of the gradient descent technique in the context of Poisson regression.

Poisson regression. Regression is a common statistical technique to learn a function $g(\mathbf{x}_i) \approx y_i$, given n training samples $\mathbf{x}_i$ (each with m features), and corresponding output labels y_i. Different forms of regression model different classes of functions g. For example, machine learning has extensively used linear regression (to model linear outputs) and logistic regression (to model binary outputs).

When the response variable y is count or rate-based (rather than continuous), using Poisson regression makes more sense. For Poisson regression, the response variable is modeled as a Poisson distribution, and therefore, $g(\mathbf{x}_i) = e^{\langle \boldsymbol{\theta}, \mathbf{x}_i \rangle}$, where $\boldsymbol{\theta}$ is the coefficient or weights vector, and $\langle \cdot, \cdot \rangle$ is the dot product. Rate-data can be modeled by an extra multiplicative factor t_i denoting the time "exposure" for each sample over which the response variable was computed.

Gradient descent. Gradient descent is a standard machine learning technique used to train a model iteratively. A model can be defined by a set of parameters $\boldsymbol{\theta} = (\theta_1, \cdots, \theta_m)$. To learn the model parameters from data, the gradient descent algorithm iteratively attempts to minimize a predetermined loss function $L(\theta|data)$. At each step, the model parameters are updated based on the gradient of the loss function.

In this paper, we focus specifically on Poisson regression with exposure. For this, training data is provided as $(\mathbf{X}, \mathbf{Y}, \mathbf{T})$, where $\mathbf{X} \in \mathbb{R}^{n \times m}$ contains data for the explanatory variables, $\mathbf{Y} \in (\mathbb{R}^+)^{n \times 1}$ contains data for the response variable, and $\mathbf{T} \in (\mathbb{R}^+)^{n \times 1}$ is the exposure data. n is the number of training samples and m is the number of features (or explanatory variables). $\mathbf{T}$, also called the "exposure", allows the modeling of rate-based data. Henceforth, unless specified, we use Poisson regression with exposure. Poisson regression attempts to learn model parameters $\boldsymbol{\theta}$, by computing the gradient of the loss function as:

$$\frac{\partial L(\boldsymbol{\theta}|\mathbf{X}, \mathbf{Y}, \mathbf{T})}{\partial \boldsymbol{\theta}} = \sum_{i=1}^{n} \mathbf{x}_i (y_i - t_i e^{\langle \boldsymbol{\theta}, \mathbf{x}_i \rangle})$$

where $(\mathbf{x}_i, y_i, t_i)$ is the i^{th} data point. The training process updates the parameters iteratively. For the k^{th} iteration, $\boldsymbol{\theta}^{(k)}$ is computed as follows:

$$\begin{aligned}\boldsymbol{\theta}^{(k+1)} &= (1-\beta)\boldsymbol{\theta}^{(k)} + \alpha \sum_{i=1}^{n} \mathbf{x}_i (y_i - t_i \cdot e^{\langle \boldsymbol{\theta}, \mathbf{x}_i \rangle}) \\ &= (1-\beta)\boldsymbol{\theta}^{(k)} + \alpha \mathbf{X}^T \left(\mathbf{Y} - \mathbf{T} \circ e^{\mathbf{X}\boldsymbol{\theta}^{(k)}} \right)\end{aligned}$$

where the exponential function is applied to each element in $\mathbf{X}\boldsymbol{\theta}^{(k)}$, $\circ$ is the Hadamard (element-wise) product and the constants α and β denote the learning rate and the regularization parameter respectively. Usually $\boldsymbol{\theta}$ is initialized, as the zero vector, or with random weights.

2.2 Secure Computation Functionalities

Secure computation protocols enable functionalities where parties can compute a function on their joint private inputs in a way that is guaranteed only the output of the computation. Our protocol constructions are in a two-party setting and provide semi-honest security [18], i.e. the parties are assumed to follow the prescribed protocol. We denote the two parties by P_0 and P_1. We use $[\![x]\!]^{\mathbb{Z}_N}$ to denote an (additive) sharing of x over $\mathbb{Z}_N$. We drop the superscript when it is clear from context. We write $[\![x]\!] = ([\![x]\!]_0, [\![x]\!]_1)$ where P_0 holds $[\![x]\!]_0$ and P_1 holds $[\![x]\!]_1$ such that $[\![x]\!]_0 + [\![x]\!]_1 = x \bmod N$. The sharing is chosen randomly, for example by first choosing $[\![x]\!]_0$ uniformly at random in $\mathbb{Z}_N$ and then assigning $[\![x]\!]_1 = x - [\![x]\!]_0 \bmod N$.

We will occasionally use the notation $F([\![x]\!], [\![y]\!])$, omitting the subscript for the share. This means that P_0 and P_1 engage in a computation of some functionality F, with P_0 contributing $[\![x]\!]_0$ and $[\![y]\!]_0$ as input, and P_1 contributing $[\![x]\!]_1$ and $[\![y]\!]_1$ as input, with each party receiving its corresponding secret shares of the result as output.

Next we overview some secure computation techniques that we use in our protocols.

Multiplication using Beaver triples. Suppose that P_0 and P_1 are given shares $[\![x]\!]$, and $[\![y]\!]$, over $\mathbb{Z}_N$. To compute $[\![z]\!] = [\![xy]\!]$, a common technique in the preprocessing model is to use Beaver's multiplication trick [12]. For this, a randomly sampled tuple $([\![a]\!], [\![b]\!], [\![c]\!])$, also called a "Beaver triple", such that $c = ab \bmod N$ is provided to the two parties. Now, to compute $[\![z]\!]$, P_0 and P_1 start by locally computing $[\![u]\!] = [\![x]\!] - [\![a]\!]$ and $[\![v]\!] = [\![y]\!] - [\![b]\!]$. Next, they reconstruct u and v by communicating their share to the other party. Finally, P_i can compute its share $[\![z]\!]_i = i \cdot uv + u\,[\![b]\!]_i + v\,[\![a]\!]_i + [\![c]\!]_i$. Note now that $[\![z]\!]_0 + [\![z]\!]_1 = (x-a)(y-b) + (x-a)(b) + (y-b)(a) + c = xy - ab + c = xy \bmod N$. Note that the same technique works for multiplication in a fixed-point ring. For a protocol, a different Beaver triple is needed for each secure multiplication to be performed. Each secure multiplication needs a preprocessing of 3 ring elements per party, and has an online communication of 2 elements per party. We use $\mathcal{F}_{\mathsf{mult}}([\![x]\!], [\![y]\!])$ to denote executing the secure multiplication functionality.

Vectorization for Beaver triples. Beaver triples also work for secure matrix multiplication, where the two matrices to be multiplied are secret shared between P_0 and P_1 [23]. Consider a matrix multiplication between an $n \times m$ size matrix $\mathbf{X}$ and an $m \times k$ size matrix $\mathbf{Y}$. Naively, the matrix multiplication $\mathbf{XY}$ requires nmk multiplications, and therefore nmk Beaver triples would

be necessary. However, as [23] notes, this can be optimized by sharing a matrix Beaver triple $(\llbracket \mathbf{A} \rrbracket, \llbracket \mathbf{B} \rrbracket, \llbracket \mathbf{C} \rrbracket)$, where $\mathbf{A}$ and $\mathbf{B}$ are matrices with the same dimension as $\mathbf{X}$ and $\mathbf{Y}$ respectively, and $\mathbf{C} = \mathbf{XY} \bmod N$. Here, only $nm + mk + nk$ ring elements are needed as preprocessing per party instead of $3nmk$ in the naive method. The online cost is similarly improved $(2(nm + mk)$ vs $2nmk)$ per party. We use $\mathcal{F}_{\mathsf{matMult}}(\llbracket \mathbf{X} \rrbracket, \llbracket \mathbf{Y} \rrbracket)$ to denote executing the secure matrix multiplication functionality.

We use standard Ring-LWE based techniques to generate the Beaver triples, and compress the real number of bits required for preprocessing and communication. We provide a background on Ring-LWE in Section 2.2.1.

2.2.1 Ring-LWE-based Encryption

Ring-Learning-With-Errors [20] (RLWE) is a hardness assumption based on which one can construct efficient homomorphic encryption schemes. We use the leveled encryption scheme proposed by Brakersky et al. [13], based on RLWE, to generate the Beaver triples in our preprocessing phase. For a positive integer N, the scheme is defined over the ring $R = \mathbb{Z}[X]/\Phi_N(X)$ where $\Phi_N(X)$ is an N^{th} cyclotomic polynomial of degree $\phi(N)$ ($\phi(\cdot)$ is the Euler's totient function). We define the ring $R_t = R/tR$, and use p, q to denote the plaintext and ciphertext modulus respectively. Choosing p and q carefully allows us to pack $\phi(N)$ plaintexts $(m_1, \cdots, m_{\phi(N)})$ into a single ring element $m \in R_p$ and enables SIMD operations (addition, multiplication) over the packed plaintexts.

Basic definition. We provide a background on Ring-LWE key generation, encryption, and decryption below.

- *Key Generation.* In the two party setting, one party samples a key pair (sk, pk) such that $sk = (1, -s)$, where $s \in R$ with coefficients in $\{-1, 0, 1\}$ and s has low Hamming weight (e.g., H(s) = 64) and $pk = (a, b)$, where $a \leftarrow R_q$ and $b = as + t\epsilon \in R_q$ with ϵ drawn from a small noise distribution χ.
- *Encryption.* Given a packed plaintext $m \in R_p$, its fresh ciphertext can be given by (c_0, c_1) where $c_0 = m + bv + p\epsilon_0$ and $c_1 = av + p\epsilon_1$ (where $v, \epsilon_0, \epsilon_1$ are drawn from the noise distribution).
- *Decryption.* The party that holds the secret key can decrypt the ciphertext to recover the underlying plaintext. Given a ciphertext $c \equiv (c_0, c_1) \in R_q^2$, the plaintext can be computed as $\mathrm{Dec}_{sk}(c) = c_0 + c_1 s \mod p$.

Matrix-vector multiplication. Suppose that $\mathbf{A} \in \mathbb{Z}_p^{n \times k}$ is a matrix by P_0, and $\mathbf{x} \in \mathbb{Z}_p^{n \times 1}$ is held by P_1. The parties want to compute $z = \mathbf{Ax}$ securely and reveal the outcome to P_0. The BGV scheme allows us to do so. First, P_0 generates the secret-public key pair (sk, pk) for the BGV scheme and distributes the public key pk to P_1. P_0 encrypts each column of the matrix $\mathbf{A}$ separately and sends them to P_1. Let a_i denote the encryption of the i^{th} column, and x_i the i^{th} element of $\mathbf{x}$. P_1 can now compute $\mathrm{Enc}_{pk}(z) = \sum_{i=1}^{k} a_i * x_i$ using the additive homomorphic property of the encryption scheme, adding a large noise to the resulting ciphertext to hide the homomorphic operation performed. Here, $*$ represents scalar multiplication, which has the effect of multiplying each element encrypted in a_i by x_i. P_1 sends the new ciphertext back to P_0, who can use the secret key to decrypt it to $\mathbf{z} = \mathbf{Ax}$. In practice, the vector $\mathbf{z}$ is usually masked with a

random vector $\mathbf{r}$ so that P_0 obtains $\mathbf{z} = \mathbf{Ax} + \mathbf{r}$ as a share of $\mathbf{Ax}$ while P_1 holds $\mathbf{r}$ as the remaining share.

Matrix-matrix multiplication. The previous technique [26] can easily be extended to handle matrix multiplication. Let $\mathbf{X} \in \mathbb{Z}_p^{k \times t}$ be the matrix held by P_1. P_1 repeats the matrix-vector multiplication procedure between a_i and every column of $\mathbf{X}$. Upon receiving the new ciphertexts from P_1, P_0 decrypts them to obtain the output matrix $\mathbf{Z} = \mathbf{AX}$.

Packing multiple multiplications. When $\phi(N)$ is much larger than n (the number of rows of the matrix $\mathbf{A}$), we can pack many multiplications into one ciphertext. Suppose that P_0 and P_1 want to compute $\mathbf{z}^j = \mathbf{A}^j \mathbf{x}^j$ where $\mathbf{A}^j \in \mathbb{Z}_p^{n \times k}$, $\mathbf{x}^j \in \mathbb{Z}_p^{k \times 1}$ and $N = t \cdot n$. P_0 can pack the i^{th} column of t different matrices $\mathbf{A}^j$ $(j \in [t])$ into one ciphertext $\mathbf{a}_i = \text{Enc}_{pk}(a_i^1, \cdots, a_i^t)$. P_1 prepares the vector $\mathbf{x}_i = (x_i^1, \cdots, x_i^1, \cdots, x_i^t, \cdots, x_i^t) \in \mathbb{Z}^{tn \times 1}$ and computes $\text{Enc}_{pk}(\mathbf{z}) = \sum_{i=1}^{k} \mathbf{a}_i * x_i$ using the additive homomorphic property of the encryption scheme (note that $*$ is the element wise product between a ciphertext and plaintext using the SIMD multiplication). A large noise is added to the resulting ciphertext to hide the homomorphic operation performed. P_0 can now decrypt the ciphertext to obtain $\mathbf{z} = (z^1, \cdots, z^t)$.

Choosing parameters for Ring-LWE. Following the parameters suggested by [7], we use plaintext prime $p = 63$ bits and ciphertext prime $q = 167$ bits for our RLWE scheme when generating Beaver triples for rings less than 64 bits, and $p = 127$ bits and $q = 295$ bits when generating triples for rings between 64 and 128 bits. In both cases, we use a polynomial modulus of degree 16384. This is sufficient for security of at least 128 bits.

3 Secure Computation over Fixed-Point Rings

Poisson regression operates over the real numbers. When the computation is done in the clear, one can leverage floating point representation to achieve high precision. Secure computation protocol, however, are restricted to fixed-point representation. While there are techniques to emulate floating point representation in MPC [8], this is usually very expensive and the more efficient approaches are to adapt the actual computation to work with fixed-point representation while preserving accuracy. We adopt this approach in our work as well and similarly to other works [23], we will compute over fixed-point numbers mapped onto an integer ring.

We define a fixed-point ring that will be used to represent our fixed-point numbers.

Fixed-point ring. A fixed-point ring $\mathcal{R}$ is a tuple $(\mathbb{Z}_{2^l}, l_x, l_f)$ where $\mathbb{Z}_{2^l}$ is the ring of integers modulo 2^l, and l_x, l_f are positive integers with $l_f \leq l_x \leq l - 1$. $\mathcal{R}$ will be used to represent fixed-point numbers with at most l_f (binary) fractional bits, and whose absolute value is less than $2^{l_x - l_f}$. Non-negative numbers will be in the range $[0, 2^{l_x})$ and negative numbers will be in the range $(2^l - 2^{l_x}, 2^l)$ in their two's complement representation. We use $\mathcal{R}_*$ to denote this part of $\mathcal{R}$, i.e., where the fixed-point numbers are represented. $\mathcal{R}_*^+ = [0, 2^{l_x})$ and $\mathcal{R}_*^- = (2^l - 2^{l_x}, 2^l)$ denote the positive and negative parts respectively.

For a real number r, with $|r| < 2^{l_x - l_f}$, we will use the hat operator, as in $\widehat{r}$, to denote its representation in the ring $\mathcal{R}$. Note that $\widehat{r} = \lfloor 2^{l_f} \cdot r \rfloor$ when $r \geq 0$ and $\widehat{r} = 2^l - \lfloor 2^{l_f} \cdot |r| \rfloor$ when $r < 0$. For example, in $\mathcal{R} = (\mathbb{Z}_{2^{10}}, 3, 2)$, a real number $x = 1.25$ will be represented in $\mathcal{R}$ by

$\widehat{x} = \lfloor 2^2 \cdot 1.25 \rfloor = 5$, and $y = -1.25$ will be represented by $\widehat{y} = 2^{10} - \lfloor 2^2 \cdot 1.25 \rfloor = 1019$. Note that something like $z = 1.26$ will also be represented by $\widehat{z} = 5$ due to truncation.

Similarly, for a ring element $x \in \mathcal{R}_*$, i.e., $x \in [0, 2^{l_x}) \cup (2^l - 2^{l_x}, 2^l)$, we will use the under-tilde operator, as in $\underset{\sim}{x}$, to denote its canonical real number representation. By canonical, we mean the real number which gives no truncation error when represented in the ring. For instance, in the previous example, $\underset{\sim}{5} = 1.25$ and not 1.26.

Secure operations. We define secure arithmetic operations on values that have been secret shared between the two parties, P_0 and P_1, in our protocol. We distinguish between two types of operations: (1) Basic ring operations are operations over shares in the ring $\mathbb{Z}_{2^l}$ treating elements as integers; (2) Fixed-point or FP operations, on the other hand, are operations that manipulate shares in the ring $\mathbb{Z}_{2^l}$, treating the underlying elements as fixed-point numbers. For a given $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, we will use $[\![x]\!]^{\mathcal{R}}$ or $[\![x]\!]^{\mathbb{Z}_{2^l}}$ to denote additive shares of $x \in \mathbb{Z}_{2^l}$. P_0 and P_1 will hold the shares $[\![x]\!]_0$ and $[\![x]\!]_1$ respectively. We will drop the superscript $\mathcal{R}$ when it is clear from context. With this notation, we now define some basic useful secure ring operations.

1. Basic operations:

   - (Addition - Add). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{Add}([\![x]\!], [\![y]\!])$ outputs $[\![x + y]\!]$.
   - (Multiplication - Mult). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{Mult}([\![x]\!], [\![y]\!])$ outputs $[\![xy]\!]$.

   Addition can be done non-interactively by each party locally adding its shares modulo 2^l. Multiplication is modulo 2^l and can be done via Beaver's multiplication trick (also known as Beaver triples [12]) in one interactive round.

2. Fixed-Point operations: These operations are for over elements in $\mathcal{R}_*$. Intuitively, the functionality for these operations can be thought of as first retrieving the real numbers corresponding to the ring elements (using the under-tilde operator), then computing the result in real numbers, and finally casting back into the fixed-point ring (using the hat operator).

   - (FP Addition - FPAdd). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{FPAdd}([\![x]\!], [\![y]\!])$ outputs $\left[\!\!\left[\widehat{(\underset{\sim}{x}) + (\underset{\sim}{y})}\right]\!\!\right]$.
   - (Public FP Multiplication - $\mathsf{PubFPMult}$). Given $[\![x]\!]$ and a public element $c \in \mathcal{R}_*$, $\mathsf{PubFPMult}([\![x]\!], c)$ outputs $\left[\!\!\left[\widehat{(\underset{\sim}{c})(\underset{\sim}{x})}\right]\!\!\right]$.
   - (FP Multiplication - FPMult). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{FPMult}([\![x]\!], [\![y]\!])$ outputs $\left[\!\!\left[\widehat{(\underset{\sim}{x})(\underset{\sim}{y})}\right]\!\!\right]$.
   - (Public FP Division - $\mathsf{PubFPDiv}$). Given $[\![x]\!]$ and a public positive integer $c \in \mathbb{Z}^+$, $\mathsf{PubFPMult}([\![x]\!], c)$ outputs $\left[\!\!\left[\widehat{(\underset{\sim}{x})/c}\right]\!\!\right]$.
   - (FP Exponentiation - FPExp). Given a public positive base element $b \in [0, 2^{l_x})$, and a shared exponent $[\![x]\!]$, $\mathsf{FPExp}(b, [\![x]\!])$ outputs $\left[\!\!\left[\widehat{(\underset{\sim}{b})^{(\underset{\sim}{x})}}\right]\!\!\right]$.

Note that the basic addition and multiplication operations are over $\mathbb{Z}_{2^l}$ but for FP operations, they are over real numbers. It is easy to see though that Add and FPAdd provide the same functionality when the underlying shares represent valid fixed-point elements. To avoid overflow for FP

operations, we will require that the underlying real numbers represented by any FP operation will still be smaller in absolute value than the $2^{l_x - l_f}$. In practice, this can be done be choosing a large enough ring to handle the range of values necessary for any computation.

Similar to the basic operations, fixed-point addition can be done non-interactively, and multiplication can be done using Beaver triples. Due to truncation, secure fixed-point multiplication can have an error of at most 2^{-l_f} in the underlying computation. Public fixed-point multiplication and division can both be done non-interactively, and we provide protocols to do so in Section 3.1that have an error of at most 2^{-l_f}. The exponentiation protocol is a novel contribution of our paper and we provide the full details in Section 6.

We can also use a prime modulus q for our fixed-point ring (instead of 2^l), embed fixed-point numbers into $[0, 2^{l_x}) \cup (q - 2^{l_x}, q)$ in $\mathbb{F}_q$, and define all of the above operations similarly over $\mathbb{F}_q$.

Approximation and rounding errors. The secure computation of FP operations may come inbuilt with errors, as a result of truncation. To bound the errors of our techniques, we define an ϵ-approximate computation (for FP operations) to denote an error of at most ϵ in the underlying fixed-point computation.

Ring change. A final useful operation we introduce is to switch between rings with different moduli. Given N and N', and a shared value $[\![x]\!]^{\mathbb{Z}_N}$, the operation $\mathsf{RingChange}([\![x]\!]^{\mathbb{Z}_N}, \mathbb{Z}_{N'})$ will output $[\![x]\!]^{\mathbb{Z}_{N'}}$, a sharing of $x \pmod{N'}$ in $\mathbb{Z}_{N'}$. We will only require the operation for $N' > N$ and when x is small ($x < 2^{l_x}$) which allows us to do this without any interaction. We detail a non-interactive protocol for this in Section 3.1.

3.1 Detailed secure functionalities

We provide more details on the public fixed-point multiplication, and division functionalities, as well as the RingChange operation, and RLWE encryption.

Public fixed-point division. Consider a sharing $[\![x]\!]$ over $\mathcal{R}_*^+$ with modulus N, and a public positive divisor $c \in \mathbb{Z}^+$. Recall that except with probability $2^{l_x}/N$, the sharing is such that $[\![x]\!]_0 + [\![x]\!]_1 = x + N$. Now, to compute the fixed-point division by c, P_0 computes $[\![z]\!]_0 = N - \left\lfloor \frac{N - [\![x]\!]_0}{c} \right\rfloor$ and P_1 computes $[\![z]\!]_1 = \left\lfloor \frac{[\![x]\!]_1}{c} \right\rfloor$. Notice now that, $\frac{x}{c} + N - 1 \leq [\![z]\!]_0 + [\![z]\!]_1 \leq \frac{x}{c} + N + 1$. Therefore, $([\![z]\!]_0, [\![z]\!]_1)$ is a sharing of the representation of $\underline{x}/c$ in $\mathcal{R}$, with an error at most 2^{-l_f}.

Public fixed-point multiplication. Consider a sharing $[\![x]\!]$ over $\mathcal{R}_*$ with modulus N, and a positive public element $c \in \mathcal{R}_*^+$. Let $[\![x]\!] = (r, (x - r) \bmod N)$. Let $[\![z]\!]_0 = \left\lfloor \frac{c[\![x]\!]_0 - cN}{2^{l_f}} \right\rfloor \bmod N$ and $[\![z]\!]_1 = \left\lfloor \frac{c[\![x]\!]_1}{2^{l_f}} \right\rfloor \bmod N$. Let $\frac{c[\![x]\!]_0 - cN}{2^{l_f}} = w_0 - d_0$, and $\frac{c[\![x]\!]_1}{2^{l_f}} = w_1 + d_1$, where w_i are the integer parts and $0 \leq d_i < 1$ are the fractional parts. Note the negative sign on d_0 since $N > [\![x]\!]_0$. We show that $([\![z]\!]_0, [\![z]\!]_1)$ form a sharing of $\widehat{(c)(\underline{x})}$. Recall that this is $\frac{cx}{2^{l_f}}$ when $x \in \mathcal{R}_*^+$ and $N - \frac{c(N-x)}{2^{l_f}}$ when $x \in \mathcal{R}_*^-$.

Case 1) $x \in \mathcal{R}_*^+$. Then, when $r \in [2^{l_x}, N)$, the sharing is such that $[\![x]\!]_0 + [\![x]\!]_1 = x + N$. Now, $[\![z]\!]_0 + [\![z]\!]_1 \bmod N \equiv w_0 + w_1 \equiv (w_0 - d_0 + w_1 + d_1) + (d_0 - d_1) \equiv \frac{cx}{2^{l_f}} + (d_0 - d_1)$. Therefore,

$(cx)/2^{l_f} - 1 \leq [\![z]\!]_0 + [\![z]\!]_1 \bmod N \leq (cx)/2^{l_f} + 1$.

Case 2) $x \in \mathcal{R}_*^-$. Then, when $r \in [0, N - 2^{l_x}]$, the sharing is such that $[\![x]\!]_0 + [\![x]\!]_1 = x$ (without the modulo). Now, $[\![z]\!]_0 + [\![z]\!]_1 \bmod N \equiv w_0 + w_1 \equiv (w_0 - d_0 + w_1 + d_1) + (d_0 - d_1) \equiv \frac{-c(N-x)}{2^{l_f}} + (d_0 - d_1)$. Therefore, $(N - \frac{c(N-x)}{2^{l_f}}) - 1 \leq [\![z]\!]_0 + [\![z]\!]_1 \bmod N \leq (N - \frac{c(N-x)}{2^{l_f}}) + 1$.

Consequently, when $r \in [2^{l_x}, N - 2^{l_x}]$, i.e., except with probability less than 2^{l_x+1}, this results in a sharing of the representation of $(\underline{x})(\underline{c})$, with an error of at most 2^{-l_f}.

Ring change. We only require the RingChange operation to switch rings between $\mathbb{Z}_N$ and $\mathbb{Z}_{N'}$ where $N' > N$, and only for positive fixed-point numbers. Consider a random sharing of $x \in [0, 2^{l_x})$ in $\mathbb{Z}_N$ and denote the two shares by $[\![x]\!]_0 = r$ and $[\![x]\!]_1 = x - r \bmod N$. Note that when $r \in [0, 2^{l_x})$, $[\![x]\!]_0 + [\![x]\!]_1 = x$ (even without a mod N). For any other r, $[\![x]\!]_0 + [\![x]\!]_1 = x + N$. This means that for a random sharing of x, the addition "wraps around" N with probability $1 - \frac{2^{l_x}}{N}$. Now, if we set $[\![x]\!]_0^{\mathbb{Z}_{N'}} = [\![x]\!]_0^{\mathbb{Z}_N} + N' - N$ and $[\![x]\!]_1^{\mathbb{Z}_{N'}} = [\![x]\!]_1^{\mathbb{Z}_N}$, then $([\![x]\!]_0^{\mathbb{Z}_{N'}}, [\![x]\!]_1^{\mathbb{Z}_{N'}})$ forms a sharing of x in $\mathbb{Z}_{N'}$ and wraps around N'. Consequently, except for a failure probability of at most $2^{l_x}/N$, the above protocol switches the sharing of x from $\mathbb{Z}_N$ to $\mathbb{Z}_{N'}$ with no error.

If necessary, the range for both the shares of both parties can be expanded to all of $\mathbb{Z}_{N'}$ by using a PRG. Specifically, both parties can agree on a PRG G the outputs values in $\mathbb{Z}_{N'}$, and a seed $g_0 = s$. For the j^{th} RingChange, they can compute the next PRG value g_j. Then P_0 adds g_j modulo N' to its share, and P_1 subtracts g_j modulo N' to its share.

4 Technical Overview

We start with an overview of the main building blocks for our secure Poisson regression training.

Gradient descent training. We consider a general setting where all training data is secret shared between two servers, P_0 and P_1. Our operations are all done over these shares; that is, each operation should take shares as input, and output shares of the resultant computation. We detail our secure Poisson regression protocol that uses gradient descent in Section 5.

Notice that the formula to update the weights vector in the Poisson regression (from Section 2.1), requires 2 matrix multiplication operations ($\mathbf{X}\boldsymbol{\theta}^{(k)}$ and $\mathbf{X}^T\mathbf{Z}$, where $\mathbf{Z} = \mathbf{Y} - (\mathbf{T} \circ e^{\mathbf{X}\boldsymbol{\theta}^{(k)}})$), one element-wise multiplication for a pair of vectors, and n exponentiations (one for each element in $\mathbf{X}\boldsymbol{\theta}^{(k)}$). Each of these operations is performed on shared inputs, and takes one communication round, resulting in a total of 4 rounds for a single iteration of our protocol. The other computations can all be done locally.

Batched optimization for matrix multiplications. While standard techniques exist for secure matrix multiplication using Beaver triples (for example, SecureML, ABY3 [22, 23]), we observe one of the multiplicands in both matrix multiplications stays constant across all iterations ($\mathbf{X}$ for the first multiplication, and $\mathbf{X}^T$ for the second). Equipped with this observation, we define a new "batched" optimization for multiplication. For this, we generate correlated Beaver triples, instead of a fresh independent triple for each multiplication. Essentially, to compute many multiplications of the form $\mathbf{X}\mathbf{M}_j$, instead of using independent Beaver triples $([\![\mathbf{A}_j]\!], [\![\mathbf{B}_j]\!], [\![\mathbf{C}_j]\!])$ for each multiplication, we prove that the same $[\![\mathbf{A}]\!]$ can be used to mask $\mathbf{X}$ for all multiplications. This also

means that the matrix $\mathbf{X} - \mathbf{A}$ needs to be reconstructed only once. Especially when the matrix $\mathbf{X}$ is large (compared to the $\mathbf{M}_j$), as is the case in our regression protocol, this greatly improves the amortized cost per iteration for both the offline and online phase. We detail this optimization, and show how to efficiently generate correlated Beaver triples in Sections 5.2 and 5.3.

Secure fixed-point exponentiation. A major challenge for secure Poisson regression is the computation of the exponential (e^x) function, where each party holds additive shares of x in some ring, and should receive additive shares of e^x in the same ring. The primary pre-existing work to compute this functionality is the SCALE-MAMBA package [10, 11], but the authors note high numerical instability in their proposed protocol. Furthermore, their protocol requires a full bit decomposition, several rounds, and has a massive offline cost. We found this to be unsuitable for our purpose. We provide a more detailed comparison to prior work in Section 6.4.

To make our Poisson regression protocol practical, we construct a novel 1-round protocol for 2-party secure fixed-point exponentiation (for a public base). Our construction is highly efficient: It does not need a bit decomposition, requires a *single* round of communication, transmits a single field element per party in the online phase, and uses just two field elements of offline preprocessing per party. We found that our protocol has a 1000x better throughput than state-of-the-art prior work for secure exponentiation and a massive 500,000x improvement in the offline cost. In turn, this makes our overall regression extremely efficient, both in terms of communication, and offline cost.

Roughly, our protocol works as follows: First, we reduce the problem to exponentiation with base 2. Then, we separate the integer and fractional parts of the exponent and exponentiate them separately. We provide a novel way to combine both results locally to end up with multiplicative shares of the exponentiation. For this, we make a key observation, that since the fractional part is already small, its shares can be multiplied without wrapping around the modulus. This means that our fractional exponentiation can be done locally in real numbers, and we avoid the expensive usual technique of polynomial approximation by a high degree polynomial. Finally, we detail a protocol that converts from multiplicative shares to additive shares of the same value. This can be done in a single round, and is the only point in our protocol that requires any communication. The maximum error of our exponentiation protocol, is in spirit similar to chaining two secure multiplications. We provide a detailed description of our protocol, as well as an analysis of its error in Section 6.

5 Secure Poisson Regression Protocol

We detail our full secure Poisson regression protocol in this section.

Protocol input. Recall that for Poisson regression (with exposure), each of the n training samples is of the form $(\mathbf{x}_i, t_i, y_i)$ where $\mathbf{x}_i$ contains m features, t_i is the exposure value, and y_i is the response output. We use $\mathbf{X}$ to denote the $n \times m$ matrix of training samples, $\mathbf{T}$ to denote the $n \times 1$ vector of exposures, and $\mathbf{Y}$ to denote the $n \times 1$ vector of response values. We assume that all entries are already represented as fixed-point elements and shared between the two protocol parties. We use $[\![\mathbf{X}]\!], [\![\mathbf{Y}]\!], [\![\mathbf{T}]\!]$ to denote the sharings.

Protocol parameters. Prior to the protocol, we require P_0 and P_1 to agree on the following parameters: (1) A fixed-point ring $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$; (2) An l bit prime q, and an exponent bound (for the exponentiation protocol); (3) The regression parameters α (learning rate), β (regularization term), and the number of iterations K.

5.1 Basic Design

The goal of the regression protocol is to output a sharing of a weights vector $\boldsymbol{\theta}$. For this, we use gradient descent, which updates the weights at every iteration. Three variants are commonly used, which differ in the way the weights are updated: (1) Standard, where the entire dataset is used for each iteration; (2) Mini-batch, where a small random sample is used for each iteration; and (3) Stochastic, where a single random sample is used for each iteration. For this paper, we use the standard gradient descent for our secure Poisson regression, but note that our protocol can be adapted for any variant. We chose to go with the standard gradient descent int this paper, but discuss the alternative mini-batch variant in Section 5.5.

Recall that in the update step of our gradient descent, the weights for the k^{th} iteration are updated as follows:

$$\boldsymbol{\theta}^{(k+1)} = (1-\beta)\boldsymbol{\theta}^{(k)} + \alpha \mathbf{X}^T \left(\mathbf{Y} - \mathbf{T} \circ e^{\mathbf{X}\boldsymbol{\theta}^{(k)}}\right)$$

Let $[\![\boldsymbol{\theta}^{(k)}]\!]$ denote a sharing of the weights vector after the k^{th} iteration. Parties start with $[\![\boldsymbol{\theta}^{(0)}]\!]$ initialized randomly or as shares of 0. Now, each iteration of our regression proceeds as follows: (1) First, P_0 and P_1 compute the (fixed-point) matrix multiplication $[\![\mathbf{U}]\!] = \left[\!\left[\mathbf{X}\boldsymbol{\theta}^{(k)}\right]\!\right]$. (2) Next, each element in $\mathbf{Z}$ is exponentiated (n exponentiations in total). Let $[\![\mathbf{V}]\!]$ be the sharing of the result after each term in $[\![\mathbf{U}]\!]$ is exponentiated; (3) Then, P_0 and P_1 compute an element-wise product $[\![\mathbf{W}]\!] = [\![\mathbf{T} \circ \mathbf{V}]\!]$; (4) Next, P_0 and P_1 compute the (fixed-point) matrix multiplication $[\![\mathbf{Z}]\!] = [\![\mathbf{X}^T(\mathbf{Y} - \mathbf{W})]\!]$; (5) The remaining computations (public multiplication by α), and addition by $(1-\beta)\boldsymbol{\theta}^{(k)}$ can be computed locally, to end up with shares of the updated weights $\boldsymbol{\theta}^{(k+1)}$. Our protocol requires 4 rounds, one for each of the first four steps. Figure 1 contains a detailed description of our protocol. The element-wise product and matrix multiplications, can be computed using the functionality $\mathcal{F}_{\mathsf{mult}}$ and $\mathcal{F}_{\mathsf{matMult}}$ respectively, and implemented using matrix Beaver triples as preprocessing. The fixed-point exponentiations are computed using the functionality $\mathcal{F}_{\mathsf{FPexp}}$, which we describe in detail in Section 6.

Basic protocol cost. From the previous description, we note that each gradient descent iteration computes 2 matrix computations (of sizes $(n \times m, m \times 1)$ and $(m \times n, n \times 1)$, 1 element-wise product for n size vectors, and n secure exponentiations. By using the matrix Beaver triples optimization from [23], a total of $2nm+n$ triples are enough in the preprocessing stage (per iteration). In addition to this, we utilize further optimizations for batched multiplication that substantially improve the performance of our protocol, when amortized over multiple iterations. Our key observation for this optimization is that the matrix multiplications in each iteration have $\mathbf{X}$, or $\mathbf{X}^T$ as one of the multiplicands. In other words, for K iterations, we have K multiplications of the form $(\mathbf{X}, \cdot)$ and K of the form $(\mathbf{X}^T, \cdot)$. This allows us to batch together the multiplications in separate iterations using correlated randomness where one of the matrices in the Beaver triple is reused. We detail this optimization in Section 5.2, and show that it does not leak any extra information about the multiplicands.

Secure Poisson Regression

Setup. P_0 and P_1 agree on a fixed-point ring $\mathcal{R}$, a prime q, and parameters for the Poisson regression: learning rate α, a regularization term β, number of iterations K.
Input. Two parties have shares $(\llbracket \mathbf{X} \rrbracket_i, \llbracket \mathbf{Y} \rrbracket_i, \llbracket \mathbf{T} \rrbracket_i)$ over $\mathcal{R}$. $\mathbf{X} \in \mathcal{R}^{n \times m}$ is the feature matrix where n is the number of samples and m is the number of explanatory variables, $\mathbf{Y} \in \mathcal{R}^n$ is the label vector, $\mathbf{T} \in \mathcal{R}^n$ is the exposure vector.

Protocol.

1. Both parties initialize shares $\llbracket \boldsymbol{\theta}^{(0)} \rrbracket$ to 0^m.
2. For $k = 1$ to K do:
   (a) The parties make a call to $\mathcal{F}_{\mathsf{matMult}}$, and set $\llbracket \mathbf{U} \rrbracket \leftarrow \mathcal{F}_{\mathsf{matMult}}(\llbracket \mathbf{X} \rrbracket, \llbracket \boldsymbol{\theta}^{(k-1)} \rrbracket)$.
   (b) The parties make a call to $\mathcal{F}_{\mathsf{FPexp}}$ on each element of $\llbracket \mathbf{U} \rrbracket$. Let $\llbracket \mathbf{V} \rrbracket \leftarrow \mathcal{F}_{\mathsf{FPexp}}(e, \llbracket \mathbf{U} \rrbracket)$.
   (c) The parties make calls to $\mathcal{F}_{\mathsf{mult}}$ on corresponding element of the vectors $\llbracket \mathbf{T} \rrbracket$ and $\llbracket \mathbf{V} \rrbracket$. Let $\llbracket \mathbf{W} \rrbracket \leftarrow \mathcal{F}_{\mathsf{mult}}(\llbracket \mathbf{T} \rrbracket, \llbracket \mathbf{V} \rrbracket)$.
   (d) The parties compute $\llbracket \mathbf{S} \rrbracket \leftarrow \llbracket \mathbf{Y} - \mathbf{W} \rrbracket$ locally.
   (e) The parties make a call to $\mathcal{F}_{\mathsf{matMult}}$, and set $\llbracket \mathbf{Z} \rrbracket \leftarrow \mathcal{F}_{\mathsf{matMult}}(\llbracket \mathbf{X}^T \rrbracket, \llbracket \mathbf{S} \rrbracket)$.
   (f) The parties update their share for $\boldsymbol{\theta}$ locally:

$$\llbracket \boldsymbol{\theta}^{(k)} \rrbracket \leftarrow (1 - \beta) \cdot \llbracket \boldsymbol{\theta}^{(k-1)} \rrbracket + \alpha \cdot \llbracket \mathbf{Z} \rrbracket$$

Output. Party P_i outputs its share $\llbracket \boldsymbol{\theta}^{(K)} \rrbracket_i$.

Figure 1: 2PC protocol for Secure Poisson Regression.

The n secure exponentiations in each iteration require a total preprocessing of $2n$ field elements per party, and a communication of n field elements per party (see Section 6). Note that all of the exponentiations are independent and can be done in parallel in a single round.

Failure probability. The fixed-point multiplication, and exponentiation operations have a small failure probability, which depends on the chosen parameters. We compute the overall failure probability for our regression protocol, which will be helpful to choose appropriate parameters for a given acceptable failure probability.

Consider $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, and $\mathbb{F}_q$ as parameters for our regression protocol. Each fixed-point multiplication has a failure probability of at most 2^{l_x+1-l} due to truncation. For matrix multiplication between a $(n \times m)$, and a $(m \times k)$ matrix, the failure probability is at most $nk \cdot 2^{l_x+1-l}$ (see [23]).

For each iteration of the regression, there are a total of $2(n+m)$ truncations for the multiplication steps (n each from steps 2a and 2c, and m each for steps 2e and 2f), which add up to a failure probability of $(2n + 2m) \cdot 2^{l_x+1-l}$. Additionally, there are n exponentiations in step 2c, each of which has a failure probability of at most $2^{l_x+1}/q$ (see Section 6 for details). Therefore, by the union bound, the total failure probability of our regression protocol for K iterations is at most $K(2(n+m) \cdot 2^{l_x+1-l} + n \cdot 2^{l_x+1}/q)$. This dictates the parameter choices for the fixed-point ring and the prime field required for an acceptable failure probability, say $p_{\mathsf{fail}} < 2^{-40}$. Note that the failure

Secure Batched Matrix Multiplication Functionality $\mathcal{F}_{\mathsf{batchMult}}$

Input. P_i has shares $(\llbracket \mathbf{X} \rrbracket_i, \llbracket \mathbf{Y}_1 \rrbracket_i, \cdots, \llbracket \mathbf{Y}_K \rrbracket_i)$.
Functionality.

1. Wait for shares $(\llbracket \mathbf{X} \rrbracket_i, \llbracket \mathbf{Y}_1 \rrbracket_i, \cdots, \llbracket \mathbf{Y}_K \rrbracket_i)$ from party P_i, for $i \in \{0, 1\}$.
2. Reconstruct $\mathbf{X}, \mathbf{Y}_1, \cdots, \mathbf{Y}_K$ and compute $\mathbf{Z}_i = \mathbf{X} \cdot \mathbf{Y}_i$.
3. Secret share $\mathbf{Z}_j = \llbracket \mathbf{Z}_j \rrbracket_0 + \llbracket \mathbf{Z}_j \rrbracket_1$ for $j \in [1, K]$ where $\llbracket \mathbf{Z}_j \rrbracket_0$ is sampled uniformly at random and give $\llbracket \mathbf{Z}_j \rrbracket_i$ to P_i.

Output. Party P_i outputs shares $\llbracket \mathbf{Z}_1 \rrbracket_i, \cdots, \llbracket \mathbf{Z}_K \rrbracket_i$ where $\mathbf{Z}_j = \mathbf{X} \cdot \mathbf{Y}_j$.

Figure 2: Ideal functionality for batched matrix multiplication

probability can be made arbitrarily small by increasing l and q.

Standard Poisson regression. The secure regression protocol we described so far is for the general version of Poisson regression with exposure. Standard Poisson regression does not contain the exposure data ($\mathbf{T}$). This means that for standard Poisson regression, the element-wise product between $\mathbf{T}$ and $e^{\mathbf{X}\boldsymbol{\theta}^{(k)}}$ is no longer necessary. Therefore, we can reduce one communication round, resulting in a 3-round protocol. The other steps of our protocol remain exactly the same.

5.2 Optimized Batched Multiplication

We show our optimized batched multiplication protocol for efficient computation of many multiplications where one of the multiplicands stays the same. More specifically, we want to compute K multiplications of the form $\mathbf{X}\mathbf{Y}_j$ for secret shared matrices. P_i is provided shares $\llbracket \mathbf{X} \rrbracket_i, \llbracket \mathbf{Y}_1 \rrbracket_i, \cdots, \llbracket \mathbf{Y}_K \rrbracket_i$, and the goal now is to compute shares of the multiplications $\llbracket \mathbf{Z}_j \rrbracket = \llbracket \mathbf{X}\mathbf{Y}_j \rrbracket$ (for $j \in [1, K]$) more efficiently. We provide the ideal functionality $\mathcal{F}_{\mathsf{batchMult}}$ for this in Figure 2.

To realize this functionality, instead of using independent beaver triples $(\llbracket \mathbf{A}_j \rrbracket, \llbracket \mathbf{B}_j \rrbracket, \llbracket \mathbf{C}_j \rrbracket)$ (one for each multiplication), we prove (in Theorem 1) that we can use correlated randomness across the multiplications and therefore need a single matrix sharing $\llbracket \mathbf{A} \rrbracket$ for the $\mathbf{X}$ multiplicand. Formally, our preprocessing requirement is now the shares $\llbracket \mathbf{A} \rrbracket, \llbracket \mathbf{B}_1 \rrbracket, \cdots, \llbracket \mathbf{B}_K \rrbracket, \llbracket \mathbf{C}_1 \rrbracket, \cdots, \llbracket \mathbf{C}_1 \rrbracket$. We detail our batched multiplication protocol in Figure 3.

If $\mathbf{X}$ is large compared to the $\mathbf{Y}_j$ (as is the case in Poisson regression), this optimization is significant since we only need one matrix to mask $\mathbf{X}$ across all multiplications. Note that we can use the same batch multiplication technique to compute the element-wise product in our protocol.

Theorem 1. *The protocol $\Pi_{\mathsf{batchMult}}$ in Figure 3 securely realizes the ideal functionality $\mathcal{F}_{\mathsf{batchMult}}$ in Figure 2 in the $\mathcal{F}_{\mathsf{batchBeaver}}$-hybrid world and in the presence of a semi-honest adversary.*

Proof. Let P_i be the corrupted party. The simulator $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{batchBeaver}}$ to obtain shares of the Beaver triples and hands them to P_i. This means that the distribution of the Beaver triple shares is identical in both the hybrid and ideal world. Next, the simulator opens $(\mathbf{X} - \mathbf{A})$ to a random matrix, and uses the output from the ideal functionality $\mathcal{F}_{\mathsf{batchMult}}$ to compute the values $(\mathbf{Y}_j - \mathbf{B}_j)$. This allows $\mathcal{S}$ to simulate the shares $\llbracket (\mathbf{Y}_j - \mathbf{B}_j) \rrbracket_{1-i}$ of the other party in the online phase. Note that $\llbracket (\mathbf{Y}_j - \mathbf{B}_j) \rrbracket_{1-i}$ is a function of the Beaver triple shares, the output $\llbracket \mathbf{Z}_j \rrbracket_i$ and

Secure Batched Matrix Multiplication Protocol $\Pi_{\mathsf{batchMult}}$

Input. P_i has shares $(\llbracket \mathbf{X} \rrbracket_i, \llbracket \mathbf{Y}_1 \rrbracket_i, \cdots, \llbracket \mathbf{Y}_K \rrbracket_i)$.
Offline Phase. P_i retrieves correlated Beaver triples $(\llbracket \mathbf{A} \rrbracket_i, \llbracket \mathbf{B}_1 \rrbracket_i, \cdots, \llbracket \mathbf{B}_K \rrbracket_i, \llbracket \mathbf{C}_1 \rrbracket_i, \cdots, \llbracket \mathbf{C}_K \rrbracket_i)$ such that $\mathbf{C}_j = \mathbf{A} \cdot \mathbf{B}_j$ for $j \in [1, K]$. This is done by making a call to $\mathcal{F}_{\mathsf{batchBeaver}}$.

Online Protocol.

1. P_i computes shares $\llbracket \mathbf{X} - \mathbf{A} \rrbracket_i$, $\llbracket \mathbf{Y}_1 - \mathbf{B}_1 \rrbracket_i$, $\cdots$, $\llbracket \mathbf{Y}_K - \mathbf{B}_K \rrbracket_i$ locally. Then, P_0 and P_1 exchange the shares to reconstruct $(\mathbf{X} - \mathbf{A})$, $(\mathbf{Y}_1 - \mathbf{B}_1)$, $\cdots$, $(\mathbf{Y}_K - \mathbf{B}_K)$.
2. For each $j \in [1, K]$, party P_i computes
$\llbracket \mathbf{Z}_j \rrbracket_i = i \cdot (\mathbf{X} - \mathbf{A})(\mathbf{Y}_j - \mathbf{B}_j) + (\mathbf{X} - \mathbf{A}) \llbracket \mathbf{B}_j \rrbracket_i + \llbracket \mathbf{A} \rrbracket_i (\mathbf{Y}_j - \mathbf{B}_j) + \llbracket \mathbf{C}_j \rrbracket$

Output. Party P_i outputs shares $\llbracket \mathbf{Z}_1 \rrbracket_i, \cdots, \llbracket \mathbf{Z}_K \rrbracket_i$

Figure 3: Protocol for batched matrix multiplication

Correlated Beaver Triple Functionality $\mathcal{F}_{\mathsf{batchBeaver}}$

Parameters. Let n, m, k, K, N be functionality parameters, where n, m, k are used to define matrix sizes, K is the number of triples to generate, and N defines the ring $\mathbb{Z}_N$. **Functionality.**

1. Sample uniformly at random $\llbracket \mathbf{A} \rrbracket_i \in \mathbb{Z}_N^{n \times m}$, $\llbracket \mathbf{B}_j \rrbracket_i \in \mathbb{Z}_N^{m \times k}$, $\llbracket \mathbf{C}_j \rrbracket_0 \in \mathbb{Z}_N^{n \times k}$ for $i \in \{0, 1\}, j \in [1, K]$.
2. Compute $\llbracket \mathbf{C}_j \rrbracket_1 = (\llbracket \mathbf{A} \rrbracket_0 + \llbracket \mathbf{A} \rrbracket_1)(\llbracket \mathbf{B}_j \rrbracket_0 + \llbracket \mathbf{B}_j \rrbracket_1) - \llbracket \mathbf{B}_j \rrbracket_0$.
3. Send $(\llbracket \mathbf{A} \rrbracket_i, \llbracket \mathbf{B}_j \rrbracket_i, \llbracket \mathbf{C}_j \rrbracket_i)$ to P_i.

Output.

1. P_0 outputs $(\llbracket \mathbf{A} \rrbracket_0, \llbracket \mathbf{B}_1 \rrbracket_0, \llbracket \mathbf{C}_1 \rrbracket_0, \cdots, \llbracket \mathbf{B}_K \rrbracket_0, \llbracket \mathbf{C}_K \rrbracket_0)$.
2. P_0 outputs $(\llbracket \mathbf{A} \rrbracket_1, \llbracket \mathbf{B}_1 \rrbracket_1, \llbracket \mathbf{C}_1 \rrbracket_1, \cdots, \llbracket \mathbf{B}_K \rrbracket_1, \llbracket \mathbf{C}_K \rrbracket_1)$ where $\llbracket \mathbf{C}_j \rrbracket_1 = -\mathbf{R}_j$.

Figure 4: Ideal functionality for generating correlated Beaver triples

$(\mathbf{X} - \mathbf{A})$. In both worlds, these matrices are uniformly random and independent from one another. Therefore, the joint distributions between the hybrid world and the ideal world are identical. □

5.3 Offline Phase

We now discuss how to generate the correlated Beaver triples used in our Poisson regression protocol. We describe the ideal functionality Figure 4 and our specific protocol in Figure 5, which uses an additively homomorphic encryption (AHE) scheme to generate the triples. To generate K correlated triples, the protocol proceeds as follows: First, P_0 and P_1 sample random matrices to be shares of $\mathbf{A}$ and $\mathbf{B}_1, \cdots, \mathbf{B}_K$. Next, P_0 samples a key for the AHE scheme and sends the encryption of its shared matrices to P_1. For each j, P_1 now selects a random matrix as its share for $\mathbf{C}_j$ and uses the homomorphic property to compute the share for P_0 inside the encryption. Finally, P_1 sends this to P_0, who can retrieve its own share by decrypting the message. Theorem 2 proves the correctness of our triple generation.

Correlated Beaver Triple Protocol $\Pi_{\mathsf{batchBeaver}}$

Setup. P_0 and P_1 agree on a ring $\mathbb{Z}_N$ and the parameters of an additive homomorphic encryption (AHE) scheme. They also agree on the number of Beaver triple matrices to generate.

Protocol.

1. P_0 and P_1 sample shares of random matrices $\mathbf{A}, \mathbf{B}_1, ..., \mathbf{B}_K$ (denoted $(\llbracket \mathbf{A} \rrbracket_0, \llbracket \mathbf{B}_1 \rrbracket_0, \cdots, \llbracket \mathbf{B}_K \rrbracket_0)$ and $(\llbracket \mathbf{A} \rrbracket_1, , \llbracket \mathbf{B}_1 \rrbracket_1, \cdots, \llbracket \mathbf{B}_K \rrbracket_1)$ respectively).
2. P_0 samples a secret key sk of the AHE scheme and computes ($\mathbf{E} \leftarrow \mathrm{Enc}_{sk}(\llbracket \mathbf{A} \rrbracket_0)$, $\mathbf{F}_1 \leftarrow \mathrm{Enc}_{sk}(\llbracket \mathbf{B}_1 \rrbracket_0)$, $\cdots$, $\mathbf{F}_K \leftarrow \mathrm{Enc}_{sk}(\llbracket \mathbf{B}_K \rrbracket_0)$). It then sends $(\mathbf{E}, \mathbf{F}_1, \cdots, \mathbf{F}_K)$ to P_1.
3. Upon receiving $(\mathbf{E}, \mathbf{F}_1, \cdots, \mathbf{F}_K)$ from P_0, P_1 samples random matrices $(\mathbf{R}_1, \cdots, \mathbf{R}_K)$ and uses the additive homomorphic property of the encryption scheme to compute $\mathbf{D}_j = \llbracket \mathbf{A} \rrbracket_1 \cdot \mathbf{F}_j + \mathbf{E} \cdot \llbracket \mathbf{B}_j \rrbracket_1 + \llbracket \mathbf{A} \rrbracket_1 \cdot \llbracket \mathbf{B}_j \rrbracket_1 + \mathbf{R}_j = \mathrm{Enc}_{sk}(\mathbf{A} \cdot \mathbf{B}_j + \mathbf{R}_j - \llbracket \mathbf{A} \rrbracket_0 \cdot \llbracket \mathbf{B}_j \rrbracket_0)$. P_1 sends $\mathbf{D}_1, \cdots, \mathbf{D}_K$ back to P_0.
4. P_0 decrypts $\mathbf{D}_j$ and obtains $\llbracket \mathbf{C}_j \rrbracket_0 = \mathbf{A} \cdot \mathbf{B}_j + \mathbf{R}_j$.

Output.

1. P_0 outputs $(\llbracket \mathbf{A} \rrbracket_0, \llbracket \mathbf{B}_1 \rrbracket_0, \llbracket \mathbf{C}_1 \rrbracket_0, \cdots, \llbracket \mathbf{B}_K \rrbracket_0, \llbracket \mathbf{C}_K \rrbracket_0)$.
2. P_1 outputs $(\llbracket \mathbf{A} \rrbracket_1, \llbracket \mathbf{B}_1 \rrbracket_1, \llbracket \mathbf{C}_1 \rrbracket_1, \cdots, \llbracket \mathbf{B}_K \rrbracket_1, \llbracket \mathbf{C}_K \rrbracket_1)$ where $\llbracket \mathbf{C}_j \rrbracket_1 = -\mathbf{R}_j$.

Figure 5: Protocol to generate correlated triples.

Theorem 2. *The protocol $\Pi_{\mathsf{batchBeaver}}$ in Figure 5 securely realizes the ideal functionality $\mathcal{F}_{\mathsf{batchBeaver}}$ in Figure 4 in the presence of a semi-honest adversary.*

Proof. The proof is a direct consequence of the security of the additive homomorphic scheme. First, suppose that P_0 is the corrupted party. Then, the simulator $\mathcal{S}$ queries the ideal functionality and obtains P_0's output. The simulator then sends back the ciphertexts $\mathbf{D}_j$ which encrypt the messages $\llbracket \mathbf{C}_j \rrbracket_0 - \llbracket \mathbf{A} \rrbracket_0 \cdot \llbracket \mathbf{B}_j \rrbracket_0$. It is easy to see that the joint distributions in both worlds are computationally indistinguishable by a reduction to the security of the additive homomorphic encryption scheme.

Now, suppose that P_1 is the corrupted party. To simulate P_1, it is enough to send P_1 the encryption of random messages. Again, the ability to distinguish the real and ideal execution will imply the ability to break the additive homomorphic encryption scheme.

We conclude that the joint distributions in both worlds are computationally indistinguishable. □

Offline cost improvement. Suppose that we need to compute K secure matrix multiplications of the form $\mathbf{X}\mathbf{Y}_j$ where $\mathbf{X}$ is of size $n \times m$, and each $\mathbf{Y}_j$ is of size $m \times k$. By generating correlated Beaver triples using our protocol, the total communication cost to generate K sets of triples is $nm + Kmk + 2Knk$ field elements, i.e., an amortized cost of $\frac{nm}{K} + mk + 2nk$ for 1 triple. Note that a fresh ciphertext created by P_0 using its secret key has the same length as the plaintext plus one more random seed (for which the cost can be ignored). Without our batched optimization, the amortized communication cost to generate one Beaver triple $(\llbracket \mathbf{A}_i \rrbracket, \llbracket \mathbf{B}_i \rrbracket, \llbracket \mathbf{C}_i \rrbracket)$ for SecureML is $nm + mk + 2nk$ field elements.

The gains are largest when $\mathbf{X}$ is large. This is exactly the case for our Poisson regression multiplications since the second multiplicands are column vectors (i.e., $k = 1$). As a concrete

Secure Inference Protocol

Setup. $\mathrm{P}_0, \mathrm{P}_1$ agree on a fixed-point ring $\mathcal{R}$ and a prime q.
Input. P_i has share $[\![\boldsymbol{\theta}]\!]_i$ of the weights vector.

Protocol.

1. The parties make a call to $\mathcal{F}_{\mathsf{matMult}}$ to compute the dot product between $\boldsymbol{\theta}$ and $\mathbf{x}$. Let $[\![u]\!] \leftarrow \mathcal{F}_{\mathsf{matMult}}([\![\mathbf{x}^T]\!], [\![\boldsymbol{\theta}]\!])$.
2. The parties make a call to $\mathcal{F}_{\mathsf{FPexp}}$ on $[\![u]\!]$. Let $v \leftarrow \mathcal{F}_{\mathsf{FPexp}}(e, [\![u]\!])$.
3. The parties make a call to $\mathcal{F}_{\mathsf{mult}}$ to multiply $[\![t]\!]$ and $[\![u]\!]$. Let $[\![y]\!] \leftarrow \mathcal{F}_{\mathsf{mult}}([\![t]\!], [\![u]\!])$.

Output. Party P_i outputs its share $[\![y]\!]_i$.

Figure 6: Secure inference protocol.

example, when $n = 10000, m = 100, k = 1$, our amortized communication improvement for $K = 100$ is roughly 97%.

Online cost improvement. Using correlated Beaver triples also improves the online cost of our protocol. Since $\mathbf{X} - \mathbf{A}$ only needs to be reconstructed once instead of for each multiplication, the amortized total online communication per multiplication for our technique is $\frac{2nm}{K} + 2mk$ field elements. In contrast, the standard secure matrix multiplication technique has a total online cost of $2nm + 2mk$ per multiplication.

5.4 Secure Inference

A useful functionality, after the regression is complete is to use the learned weights to predict, or infer the value of the response variable for future samples. Formally, suppose that P_0 and P_1 hold a sharing $[\![\boldsymbol{\theta}]\!]$ of the weights. Now, given a new sample $([\![\mathbf{x}]\!], [\![t]\!])$ that is shared between the two parties, the goal is to use the learned weights to compute a sharing $[\![y]\!]$ of the response variable. Note that $[\![y]\!] = \left[\!\left[e^{\mathbf{x}^T\boldsymbol{\theta}}\right]\!\right]$, which can be computed securely along the same lines as our secure Poisson regression protocol. Our protocol is shown in Figure 6.

5.5 Additional Considerations

Learning rate. It is important to choose a good learning rate for the Poisson regression to converge efficiently. A large learning rate may cause the regression to not converge, while a small value can cause it to converge slowly. For regression done in the clear, the learning rate can be adjusted according to the magnitude of the gradient to maximize the efficiency of the training and to avoid divergence. In a secure setting however, this needs to be done carefully in order not to leak information. Testing whether the regression is diverging/converging, or revealing the magnitude of the gradient descent risks exposing sensitive information from the training dataset. All the probes and adjustments for the learning rate must be done securely. For example, we can add a secure function to clip the gradient if its magnitude is larger than certain bound before updating the

Secure Fixed-Point Exponentiation Functionality $\mathcal{F}_{\mathsf{FPexp}}$

Public Parameters. P_0 and P_1 agree on a fixed-point ring $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, an l-bit prime $q < 2^l$, and an exponent bound $A \in \mathbb{Z}^+$.
Input. P_0 and P_1 have shares $[\![x]\!] \in \mathcal{R}_*$, and a public real base $b \in \mathbb{R}^+$, satisfying $x \log_2(b) > 1 - A$

Functionality.

1. Let $[\![x']\!] \leftarrow \mathsf{PubFPMult}([\![x]\!], \widehat{\log_2(b)})$ // Convert to base 2 exponentiation
2. Let $[\![z]\!] \leftarrow \mathsf{FPAdd}([\![x']\!], \widehat{A})$ // Make exponent > 1
3. Let $(z_i^{\text{int}}, z_i^{\text{frac}}) \leftarrow \left(\left\lfloor [\![z]\!]_i / 2^{l_f} \right\rfloor, ([\![z]\!]_i / 2^{l_f}) - z_i^{\text{int}}\right)$ // Split into integer and fractional parts
4. Let $(z_0^{\text{int}}, z_1^{\text{int}}) \leftarrow \mathsf{RingChange}((z_0^{\text{int}}, z_1^{\text{int}}), \mathbb{Z}_{q-1})$ // RingChange from $\mathbb{Z}_{2^{l-l_f}}$ to $\mathbb{Z}_{q-1}$
5. Let $(v_i^{\text{int}}, v_i^{\text{frac}}) \leftarrow (2^{z_i^{\text{int}}} \bmod q, 2^{z_i^{\text{frac}}})$ // Exponentiate both parts
6. Let $v_i \leftarrow \left(v_i^{\text{int}} \cdot \left\lfloor 2^{l_f} v_i^{\text{frac}} \right\rfloor\right) \bmod q$ // Get each party's local share
7. Let $y' \leftarrow v_0 v_1 \bmod q$ // Combine shares of both parties
8. Create a random additive sharing $[\![y']\!]$ in $\mathbb{F}_q$ // Convert to additive shares
9. Let $[\![y]\!]^{\mathbb{F}_q} \leftarrow \mathsf{PubFPDiv}([\![y']\!], 2^{l_f + A})$ // Divide by the remaining factor
10. Let $[\![y]\!]^{\mathbb{Z}_{2^l}} \leftarrow \mathsf{RingChange}([\![y]\!]^{\mathbb{F}_q}, \mathbb{Z}_{2^l})$ // RingChange from $\mathbb{F}_q$ to $\mathbb{Z}_{2^l}$

Figure 7: Functionality $\mathcal{F}_{\mathsf{FPexp}}$

weights. This is useful as the gradient tends to be large at the beginning of the training and could cause divergence even when the learning rate is relatively small.

Mini-batches. Mini-batched gradient descent, where a small random batch of training samples is used for every iteration, is usually more efficient than batched gradient descent. In this paper, however, we do not run experiments with mini-batch gradient descent as our datasets are all small (each has less than 75 features).

Our correlated Beaver triples also work for mini-batched gradient descent, and allow for similar improvements in both the offline and online phase. For iteration j, the parties generate Beaver triples $([\![\mathbf{A}]\!], [\![\mathbf{B}_j]\!], [\![\mathbf{C}_j]\!])$ and sample a random permutation π_j together. The rows of $([\![\mathbf{A}]\!], [\![\mathbf{C}_j]\!])$, and the training data are then shuffled according to π_j. Now, the triples and the training data can be partitioned into mini batches, with each gradient descent iteration run over a different mini-batch.

6 Secure Fixed-Point Exponentiation

In this section, we will detail our novel secure fixed-point exponentiation protocol. To simplify our analysis, our protocol will mirror $\mathcal{F}_{\mathsf{FPexp}}$ functionality (Figure 7) rather than the previously defined FPExp operation. Note that due to truncation errors, the two functionalities are not identical. However, we will show later (in Section 6.3) that the result computed by $\mathcal{F}_{\mathsf{FPexp}}$ is close to the actual fixed-point exponentiation result. Similar to the FPExp operation, the functionality $\mathcal{F}_{\mathsf{FPexp}}$ will take as inputs a public base and a secret shared exponent. Since we are working in a fixed-point ring, we will consider our inputs to be the fixed-point representations rather than the real numbers themselves. Given a fixed-point ring $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, a public base $b \in \mathcal{R}_*$, and a shared exponent

Multiplicative to Additive Conversion: MTA

Public Parameters. A finite field $\mathbb{F}_q$ where q is prime. All operations will be in $\mathbb{F}_q$.
Preprocessing. P_0 is given (α_0, β_0) and P_1 is given (α_1, β_1) satisfying $\alpha_0\alpha_1 + \beta_0\beta_1 = 1$.
Input. P_0 and P_1 have multiplicative shares m_0 and m_1 of s, i.e., $s = m_0 m_1$.
Required Output. P_i outputs a_i such that $a_0 + a_1 = s$.
Protocol Description.

P_0		P_1
$v_0 \leftarrow \beta_0 m_0$		$v_1 \leftarrow \alpha_1 m_1$
	$\xrightarrow{v_0}$	
	$\xleftarrow{v_1}$	
$a_0 \leftarrow \alpha_0 m_0 v_1$		$a_1 \leftarrow \beta_1 m_1 v_0$
Output a_0		Output a_1

Figure 8: Protocol MTA to convert from multiplicative to additive shares

$[\![x]\!]$, $\mathcal{F}_{\mathsf{FPexp}}(b, [\![x]\!])$ will compute a sharing of something "close" to $\widehat{(b)^{(x)}}$. We compare our work to existing techniques in Section 6.4 and benchmark our protocol in Section 7.1.

6.1 Protocol Construction

It is straightforward to construct a protocol that realizes the $\mathcal{F}_{\mathsf{FPexp}}$ functionality. First, we note that the PubFPMult, FPAdd, RingChange, and PubFPDiv operations used in steps 1, 2, 4, 9, and 10 of $\mathcal{F}_{\mathsf{FPexp}}$ can all be computed by locally manipulating the shares. Steps 3, 5 and 6 are also purely local computations. The only point at which communication will be necessary is to retrieve an additive sharing of y' (steps 7, 8). Effectively, here, P_0 and P_1 need to go from a multiplicative sharing of $y' \in \mathbb{F}_q$ to an additive sharing of the same y'.

To accomplish this, we will use a 2-party variant of the efficient MTA (multiplicative to additive) protocol from Ghodosi et al. [17] (also given in Figure 8). Suppose that P_0 and P_1 hold multiplicative shares m_0 and m_1 of a secret s in field $\mathbb{F}_q$. The protocol requires a tuple (α_i, β_i) of preprocessed values (in $\mathbb{F}_q$) such that $\alpha_0\alpha_1 + \beta_0\beta_1 = 1$. Now, the MTA protocol proceeds as follows: First, P_0 and P_1 simultaneously send $v_0 = \beta_0 m_0$ and $v_1 = \alpha_1 m_1$ respectively to the other party. Finally, P_0 and P_1 can compute $a_0 = \alpha_0 m_0 v_1$ and $a_1 = \beta_1 m_1 v_0$. Note that a_0 and a_1 are the required additive shares of s since $a_0 + a_1 = \alpha_0 m_0 \alpha_1 m_1 + \beta_1 m_1 \beta_0 m_0 = m_0 m_1(\alpha_0\alpha_1 + \beta_0\beta_1) = s$. Ghodosi et al. also show that these shares are individually uniformly random.

The source of the preprocessed values is not provided in [17] but they are nevertheless easy to compute even without a trusted dealer. For this, first, P_0 samples u_0, w_0 and P_1 samples α_1, β_1 uniformly at random from $\mathbb{F}_q^\times$. Next, the two parties can securely compute $r = u_0\alpha_1 + w_0\beta_1$, and resample if $r = 0$. The probability that a resample is necessary is at most $1/q - 1$. Finally, P_0 can set $\alpha_0 = u_0 r^{-1}$ and $\beta_0 = w_0 r^{-1}$, where r^{-1} is the multiplicative inverse of r in $\mathbb{F}_q^\times$. Notice now, that $\alpha_0\alpha_1 + \beta_0\beta_1 = rr^{-1} = 1$, as required. Note that since the resample probability is negligible, the distribution of r is negligibly close to uniformly random.

Since the only communication is through the MTA protocol, the security of our protocol securely

realizing the $\mathcal{F}_{\mathsf{FPexp}}$ functionality is a direct consequence of the security of the MTA protocol. In total, our protocol requires only one round, and a single field element sent by each party.

6.2 Protocol Details

We now describe the main technical components of why our protocol is a useful proxy for computing the fixed-point exponentiation. We defer the concrete error analysis of our protocol to Section 6.3. We begin with a simplified version of our protocol where $\underset{\sim}{b} = 2$, and the exponent satisfies $\underset{\sim}{x} > 1$, and handle other exponents and other (positive) bases later.

Our strategy works as follows: (1) First, we split the exponentiation into two parts: an integer part and a fractional part. (2) Next, each part is exponentiated separately (and locally) to get multiplicative shares of the final result (along with an extra factor). (3) We then use a single round of interaction to convert the multiplicative shares to additive shares. (4) Finally, each party can locally remove the extra factor to obtain additive shares of the final result. We detail each of these steps below.

Splitting the exponent. Let $[\![z]\!]$ be a sharing of the fixed-point exponent, where P_0 holds $[\![z]\!]_0$ and P_1 holds $[\![z]\!]_1$. We use z here (instead of x) to follow along with functionality $\mathcal{F}_{\mathsf{FPexp}}$, and standardize the notation for a general base, since the first step there would be to reduce the problem to a base 2 exponentiation. The party P_i first splits its share $[\![z]\!]_i$ as $(z_i^{\mathrm{int}}, z_i^{\mathrm{frac}})$ where $z_i^{\mathrm{int}} = \left\lfloor [\![x]\!]_i / 2^{l_f} \right\rfloor$ and $z_i^{\mathrm{frac}} = [\![z]\!]_i / 2^{l_f} - z_i^{\mathrm{int}} = ([\![z]\!]_i \bmod 2^{l_f})/2^{l_f}$. Notice now that $z = ([\![z]\!]_0 + [\![z]\!]_1 \bmod 2^l)$ $= 2^{l_f}\left((z_0^{\mathrm{int}} + z_1^{\mathrm{int}} \bmod 2^{l-l_f}) + (z_0^{\mathrm{frac}} + z_1^{\mathrm{frac}})\right)$. Therefore,

$$2^{\underset{\sim}{z}} = \left(2^{(z_0^{\mathrm{int}} + z_1^{\mathrm{int}}) \bmod 2^{l-l_f}}\right) \cdot \left(2^{z_0^{\mathrm{frac}} + z_1^{\mathrm{frac}}}\right)$$

This allows us to exponentiate the integer and fractional parts separately and combine them at a later step. Note that the two integer and fractional exponent shares may not always sum up to the actual integer and fractional parts of $\underset{\sim}{z}$ respectively. This is because the two fractional shares could add up to more than 1, leaving the integer shares to sum to $\left\lfloor z/2^{l_f} \right\rfloor - 1 \bmod 2^{l-l_f}$. Furthermore, our integer exponentiation requires the exponent to be positive. This leads to our requirement of $\underset{\sim}{z} > 1$. We will relax this assumption later.

Integer exponentiation. First, we observe that $[\![w]\!]_0 = z_0^{\mathrm{int}}$ and $[\![w]\!]_1 = z_1^{\mathrm{int}}$ form a sharing of $w = (z_0^{\mathrm{int}} + z_1^{\mathrm{int}} \bmod 2^{l-l_f})$ over the ring $\mathfrak{R} = \mathbb{Z}_{2^{l-l_f}}$. Denote this sharing by $[\![w]\!]^{\mathfrak{R}}$. Now, we can use existing integer ring exponentiation techniques (such as [9, 25, 33]) to compute 2^w. These techniques however require a few rounds of communication even for a public base. Instead, here, we will describe an alternative method that can be done locally in a way that will seamlessly combine with the fractional exponentiation part.

For this, we assume that the parties have agreed on an l-bit prime q (i.e., $2^{l-1} < q < 2^l$). We will first convert the sharing of w in $\mathfrak{R}$ to a sharing in $\mathbb{Z}_{q-1}$ using the RingChange operation. Note that the ring size increases if at least 1 fractional bit is present. Recall that since w is positive (from our exponent assumption), with probability $(1 - 2^{l_x}/q)$, the new sharing $[\![w]\!]^{\mathfrak{R}}$ will satisfy $w + (q-1) = [\![w]\!]_0^{\mathfrak{R}} + [\![w]\!]_1^{\mathfrak{R}}$. Now, the two parties can exponentiate their shares locally $(\bmod\ q)$ to directly get a multiplicative sharing of 2^w. This works since,

$$\left(2^{[\![w]\!]_0^{\mathfrak{R}}} \bmod q\right) \cdot \left(2^{[\![w]\!]_1^{\mathfrak{R}}} \bmod q\right) \bmod q \left(2^{w+q-1}\right) \bmod q = 2^w \bmod q$$

where the last step is due to Fermat's little theorem. Let $v_0^{\text{int}} = 2^{[\![w]\!]_0^{\mathfrak{R}}} \bmod q$ and $v_1^{\text{int}} = 2^{[\![w]\!]_1^{\mathfrak{R}}} \bmod q$ be the final multiplicative shares (in $\mathbb{F}_q$) of 2^w held by P_0 and P_1.

Fractional exponentiation. Let z_0^{frac} and z_1^{frac} be the fractional exponents held by P_0 and P_1 respectively. Notice that if both parties locally exponentiate (in $\mathbb{R}$) their shares, they would end up with multiplicative shares (in $\mathbb{R}$) of the fractional exponentiation result. Specifically, if P_i computes $v_i^{\text{frac}} = 2^{z_i^{\text{frac}}}$, then $v_0^{\text{frac}} \cdot v_1^{\text{frac}} = 2^{z_0^{\text{frac}} + z_1^{\text{frac}}}$. To allow for seamless integration with the integer exponentiation part, we have P_i later compute $\left\lfloor 2^{l_f} \cdot v_i^{\text{frac}} \right\rfloor$. A crucial observation here is that since $2^0 \leq v_i^{\text{frac}} < 2^1$, $\left\lfloor 2^{l_f} \cdot v_i^{\text{frac}} \right\rfloor$ is small and positive, and therefore it can also be viewed as an element in $\mathbb{F}_q$. Furthermore, the multiplication (now in $\mathbb{F}_q$), will not wrap around the modulus q. This will allow v_i^{frac} and v_i^{int} to be combined easily. Note that the product will include an extra 2^{l_f} factor (apart from the standard fractional fixed-point multiplier). Due to truncation, the extra factor is necessary when first combining the integer and fractional parts and will be divided out later. This will become evident in our error analysis.

Combining the two parts. At this stage, P_i holds the result of the integer exponentiation v_i^{int}, and the result of the fractional part v_i^{frac}. Let $d_i = \left\lfloor 2^{l_f} \cdot v_i^{\text{frac}} \right\rfloor$. Ignoring errors due to truncation for now, we have:

$$\begin{aligned}
(v_0^{\text{int}} \cdot v_1^{\text{int}} \cdot d_0 \cdot d_1) \bmod q &\approx \left(2^{(z_0^{\text{int}} + z_1^{\text{int}}) \bmod 2^{l - l_f}}\right) \left(2^{2l_f}\right) \left(2^{z_0^{\text{frac}} + z_1^{\text{frac}}}\right) \bmod q \\
&= 2^{2l_f} 2^{z} \bmod q \\
&= 2^{l_f} \widehat{(2^{z})} \bmod q
\end{aligned}$$

This means that barring any truncation errors, if P_i computes $y_i' = v_i^{\text{int}} \cdot d_i \bmod q$, then $y_0' y_1' \bmod q \approx (2^{l_f})\widehat{(2^{z})}$. Now, P_0 and P_1 convert the multiplicative shares of $y' = y_0' y_1'$ to additive ones through the MTA protocol which requires one round of interaction. The leftover 2^{l_f} factor can be divided out through local computation using $\mathsf{PubFPDiv}$. Finally, both parties can locally use the $\mathsf{RingChange}$ protocol to convert their shares back to $\mathbb{Z}_{2^l}$. Note that this conversion is once again from a smaller to a larger ring since $q < 2^l$. We will bound the error resultant from truncation in Section 6.3.

Working with bases other than 2. Our Poisson regression usecase requires secure base e exponentiation, but so far our protocol only works for base 2. To make it work for any positive base b, we first observe that given a real exponent u, $b^u = 2^{u \log_2(b)}$. Consequently, as the first protocol step, the sharing $[\![x]\!]$ of the (base b) exponent in $\mathcal{R}$ will be converted to a sharing $[\![z]\!]$, of the equivalent base 2 exponent, where $(\underset{\sim}{z}) = (\underset{\sim}{x}) \log_2(b)$. This can be computed as $[\![z]\!] = \mathsf{PubFPMult}([\![x]\!], \widehat{\log_2(b)})$ and requires no interaction.

Working with exponents ≤ 1. As previously mentioned, we initially required our fixed-point exponent to be greater than 1 since this guarantees correctness for the integer ring exponentiation. To handle other exponents, we will assume that there is an agreed upon exponent bound $A \in \mathbb{Z}^+$, such that for base b and exponent sharing $[\![x]\!]$, it holds that $(\underset{\sim}{x} \log_2(b)) > 1 - A$, i.e., the most negative exponent for base 2 exponentiation still has an absolute value of less than $A - 1$. Suppose that $[\![x']\!]$ is the sharing of the exponent after converting to a base 2 exponentiation. We now need

to ensure that $\underset{\sim}{x}' > 1$. This can be done by adding A to the exponent, or equivalently, adding $\widehat{A}$ to the sharing using FPAdd to get a new sharing $[\![z]\!]$. At the end of the protocol, the extra 2^A factor will be divided out. We note that since the 2^A factor will be present in intermediate steps, both $\mathcal{R}$ and $\mathbb{F}_q$ will need to be large enough to accommodate it.

Protocol cost and other considerations. Our exponentiation protocol has a total online cost of 2 $\mathbb{F}_q$ elements (1 per party), and a preprocessing cost of 4 $\mathbb{F}_q$ elements (2 per party). We note that our protocol can easily be adapted to working solely in the field $\mathbb{F}_q$ (with appropriately defined fixed-point representation), rather than switching between our defined fixed-point ring and $\mathbb{F}_q$. This design is simpler but usually much slower since common operations like multiplication, truncation etc., are much faster over a ring $\mathbb{Z}_{2^l}$, as compared to a field. Therefore, for our purpose, it is far more cost efficient to work mostly in $\mathbb{Z}_{2^l}$ (and $\mathbb{Z}_{2^{l-l_f}}$), and only switch to $\mathbb{F}_q$ inside of the exponentiation subprotocol.

Assumption on the exponent bound. We emphasize that our assumption of a minimum allowable exponent is not unreasonable in the context of fixed-point exponentiation. Given l_f fractional bits, $2^{(-z)}$ where $z > l_f$ is already not representable in the fixed-point ring. Consequently, this gives us a natural bound of $2^{(-z)}$ on how negative the exponent can be for the computation to even make sense. Of course, a tighter bound A can be chosen if appropriate. This observation allows our protocol to be orders of magnitude faster than prior work, since it does not require an expensive bit decomposition to first detect whether an exponent is negative; we can simply add the exponent bound to all exponents to always work with positive exponents for the main protocol. One caveat is that we lose the ability to detect if our predefined bound has been violated without resorting to a bit decomposition, and our protocol may produce incorrect results when the bound is incorrectly defined or is exceeded during protocol execution. We point out though, that this assumption is not unlike a standard assumption of a large enough ring modulus to hold the fixed-point computations, and similar assumptions appear in [5, 23].

Alternate 2-round protocol. We also describe a alternate 2-round variant of our exponentiation protocol. Here, instead of combining the integer and fractional exponentiation shares locally first, the MTA protocol is used to retrieve additive shares of the integer and fractional result separately. Note that this can be done simultaneously in 1 round. Finally, in the second round, shares of both results can be combined through a single secure multiplication. In total, 8 $\mathbb{F}_q$ elements are transmitted in the online phase, and 14 $\mathbb{F}_q$ elements are required for preprocessing. While the communication cost is larger than the previously described 1-round protocol, one upshot of this construction is that it can tolerate a smaller ring size. Recall that in the 1-round protocol, the full result along with an extra 2^{l_f} factor needs to fit in the ring. This is no longer necessary for the 2-round protocol and depending on the usecase and the number of fractional bits used, the trade-off may be acceptable. For our regression usecase however, there are other constraints that increase the size of the fixed point ring. Furthermore, in practice, the computational gain as a result of a smaller ring size (in the order of microseconds for our construction), will almost certainly be overshadowed by the extra communication round (usually in the order of milliseconds). Therefore, we use the 1-round protocol that optimizes for communication cost.

Failure probability. We analyze the total failure probability of our base 2 exponentiation protocol. Note that this is different from our error analysis in Section 6.3. In particular, we say that our exponentiation protocol has failure probability p_{fail} and error ε if, except with probability p_{fail}, the protocol error is bounded by ε. To compute the failure probability, first suppose that the (positive) base 2 exponent z is secret shared as $(\llbracket z \rrbracket_0, \llbracket z \rrbracket_1)$. With probability at least $1 - 2^{l_x - l}$, we have $\llbracket z \rrbracket_0 + \llbracket z \rrbracket_1 = z + 2^l$, i.e., the two shares wrap around $\mathbb{Z}_{2^l}$. When this happens, the integer components will also wrap around $\mathbb{Z}_{2^{l-l_f}}$, and after the RingChange to $\mathbb{Z}_{q-1}$, z_0^{int} and z_1^{int} will wrap around $\mathbb{Z}_{q-1}$.

Next, after the integer and fractional parts are exponentiated combined, and converted from multiplicative to additive shares, the random additive sharing of y' in $\mathbb{F}_q$ will also wrap around $\mathbb{F}_q$ with probability at least $1 - \frac{2^{l_x}}{q}$. Finally, the later PubFPDiv and RingChange back to $\mathbb{Z}_{2^l}$ steps will work smoothly when the sharing of y' wraps around $\mathbb{F}_q$.

Therefore, using the union bound, we can bound the total failure probability of the exponentiation protocol as $2^{l_x - l} + 2^{l_x}/q < 2^{l_x+1}/q$, since we use $q < 2^l$. Given the exponent bound A, choosing $l_x = 2A + 2l_f$ is sufficient, and therefore, we can rewrite the bound as $2^{2A+2l_f+1}/q$. Note that the failure probability can easily made as small as necessary by increasing the size of $\mathbb{F}_q$, and our fixed-point ring. For example, to achieve $p_{\text{fail}} < 2^{-40}$, with $l_f = 15$ bits of precision, and $A = 5$, roughly an 81-bit modulus will be required.

6.3 Error analysis

We will now compute a bound on the error of our exponentiation protocol for base 2. For this, we will compute the difference between the result computed by $\mathcal{F}_{\mathsf{FPexp}}$ and the actual exponentiation (in real numbers).

Let $\llbracket z \rrbracket$ be a sharing of the (base 2) exponent in the fixed-point ring $\mathcal{R} = (\mathbb{Z}_{2^l}, l_x, l_f)$, that computes the exponentiation $2^{(z)}$ (in $\mathbb{R}$). First, we note that the integer exponentiation produces no error; the only error results from the truncation in the fractional part and its subsequent combination with the exponentiation of the integer part. Let z_i^{int} and z_i^{frac} denote the integer and fractional parts of the underlying fixed-point of the share $\llbracket z \rrbracket_i$, after $(z_0^{\text{int}}, z_1^{\text{int}})$ has undergone a RingChange to become a sharing in $\mathbb{Z}_{q-1}$. Note that no error is added by the RingChange. It is easy to see that the true computation 2^z can be written as $2^{(z_0^{\text{int}} + z_1^{\text{int}} \bmod q-1)} 2^{z_0^{\text{frac}}} 2^{z_1^{\text{frac}}}$.

Following $\mathcal{F}_{\mathsf{FPexp}}$, we first compute $v_i^{\text{int}} = 2^{z_i^{\text{int}}} \bmod q$, and $v_i^{\text{frac}} = 2^{z_i^{\text{frac}}}$, and combine them to get $v_i = (v_i^{\text{int}} \cdot \lfloor 2^{l_f} v_i^{\text{frac}} \rfloor) \bmod q$. Since v_i^{frac} is a positive real, suppose that $v_i^{\text{frac}} = d_i + \varepsilon_i$, where $0 \le \varepsilon_i < 2^{-l_f}$. In other words, ε_i is the part not representable in l_f fractional bits. Now, $v_i = \left(v_i^{\text{int}} \cdot 2^{l_f} \cdot (v_i^{\text{frac}} - \varepsilon_i)\right) \bmod q$. Consequently,

$$
\begin{aligned}
y' &= 2^{2l_f} \cdot v_0^{\text{int}} \cdot v_1^{\text{int}} \cdot (v_0^{\text{frac}} - \varepsilon_0) \cdot (v_1^{\text{frac}} - \varepsilon_1) \bmod q \\
&= 2^{2l_f} \cdot 2^{(z_0^{\text{int}} + z_1^{\text{int}} \bmod q-1)} \cdot (v_0^{\text{frac}} - \varepsilon_0) \cdot (v_1^{\text{frac}} - \varepsilon_1) \\
&= 2^{2l_f} \left[2^{(z_0^{\text{int}} + z_1^{\text{int}} \bmod q-1)} (v_0^{\text{frac}} v_1^{\text{frac}} - \varepsilon_0 v_1^{\text{frac}} - \varepsilon_1 v_0^{\text{frac}} + \varepsilon_0 \varepsilon_1) \right] \\
&= 2^{2l_f} \left[2^{(z)} + 2^{(z_0^{\text{int}} + z_1^{\text{int}} \bmod q-1)} (-\varepsilon_0 v_1^{\text{frac}} - \varepsilon_1 v_0^{\text{frac}} + \varepsilon_0 \varepsilon_1) \right]
\end{aligned}
$$

where the mod q can be removed from step 2 onwards, since $\mathbb{F}_q$ is large enough to accommodate the entire intermediate result. Now, $2^{(z_0^{\text{int}} + z_1^{\text{int}} \bmod q-1)} = 2^{(z)} / (v_0^{\text{frac}} \cdot v_1^{\text{frac}})$, and $1 \le v_i^{\text{frac}} < 2$ and

therefore,

$$2^{2l_f}\left[2^{(z)} - 2^{(z)} \cdot 2^{-l_f} \frac{(v_0^{\text{frac}} + v_1^{\text{frac}})}{v_0^{\text{frac}} v_1^{\text{frac}}}\right] < y' < 2^{2l_f}\left[2^{(z)} + 2^{(z)} \cdot 2^{-2l_f}\right]$$

This gives,

$$2^{l_f} 2^{(z)}(2^{l_f} - 2) < y' < 2^{l_f} 2^{(z)}(2^{l_f} + 2^{-l_f})$$

Now, $y \leftarrow \mathsf{PubFPDiv}(\llbracket y' \rrbracket, 2^{l_f})$ results in an additional potential error of at most ± 1. That is,

$$-1 + 2^{(z)}(2^{l_f} - 2) < y < 1 + 2^{(z)}(2^{l_f} + 2^{-l_f})$$

In other words, the computed fixed-point number $\underset{\sim}{y} = y/2^{l_f}$ differs from the real value $2^{(z)}$ as,

$$\left|\underset{\sim}{y} - 2^{(z)}\right| < 2^{-l_f}(2 \cdot 2^{(z)} + 1)$$

To put this in perspective, a computation of $2^{10.125} \approx 1116.68$ in a fixed point ring with $l_f = 15$, will result in a maximum possible error of 0.068, or at most 0.006%. With $l_f = 20$, the maximum error reduces to 0.0002%. This should be more than reasonable for most practical settings, and indeed fits our regression usecase well, since regression is resistant to small errors. Furthermore, we emphasize that the error can always be made arbitrarily small by increasing the number of fractional bits available for the computation. Also note that this error is achieved for the worst possible sharing of the exponent, and may be much smaller for a random sharing.

Error dependence on actual value. The astute reader might observe that the above computed error (in the fixed-point ring) is bounded by a small multiple of the actual real number result 2^z. We highlight that this is not unlike the error of chaining two truncated secure multiplications. For example, suppose that $\llbracket \widehat{a} \rrbracket, \llbracket \widehat{b} \rrbracket, \llbracket \widehat{c} \rrbracket, \llbracket \widehat{d} \rrbracket$ are sharings held by P_0 and P_1 of fixed-point numbers a, b, c, d. Recall that secure multiplication can result in an error of at most ± 1 in the fixed-point ring. This means that the secure multiplication of a, b can result in a sharing of $\widehat{ab} + 1$, while the secure multiplication of c, d can result in a sharing of $\widehat{cd} + 1$. At this point, if the two resultant shares are also multiplied, the complete result can be at most $\widehat{abcd} + \widehat{ab} + \widehat{cd} + 2$. In other words, the error here can also depend on the actual numbers involved in the computation.

Malicious security. Although our secure fixed-point exponentiation protocol operates exclusively in the semi-honest setting, we comment briefly on the challenges of extending it to a maliciously secure version. One possible technique is for the protocol parties to operate on authenticated shares [24] and use generic zero-knowledge proofs to prove that each party performs their steps correctly. However, doing so would likely reduce the efficiency gains of our protocol substantially. In particular, a key step in our protocol is separating the exponentiation into integer and fractional parts, following which the fractional part can be exponentiated locally in real numbers (or floating point) and still be seamlessly combined with the integer exponentiation part. In the malicious setting, it is expensive to prove that these steps were performed correctly, and it may be more efficient to use a polynomial approximation instead for the fractional exponentiation, together with cut-and-choose or ZK techniques to prove correctness. We leave these explorations for future work.

6.4 Comparison to existing exponentiation techniques

To highlight the strong improvements of our secure exponentiation protocol, we provide a comparison to existing techniques in literature.

Modular and integer exponentiation. There is a long line of work [9, 15, 25, 33] on secure integer and modular exponentiation. Here, the goal is to compute $b^a \bmod N$ where both a, b are secret shared in $\mathbb{Z}_N$. Other variants have also been studied; for example, where b is assumed to be public, or when N is prime. Classical approaches, such as the one put forth by Damgård et al. [15], required a full decomposition of the shared values, which is computationally expensive. Later approaches [9, 25, 33] are more practical and did not require decomposition. Still, for semi-honest adversaries, even when the base b is public, previous approaches require more than 1 round of communication (e.g., [9] describes a 2-round protocol, and [33] describes a 3-round protocol).

A byproduct of our protocol, is a modular/integer exponentiation protocol that works for a public base, and when the exponent is small. In the online phase, it requires a single (simultaneous) communication round, and an exchange of a single field element in either direction. In other words, for exponentiation modulo a prime q, our protocol has a total communication cost of $2 \log q$ bits, and a total preprocessing cost of $4 \log q$ bits.

Fixed-point exponentiation. There are substantially fewer works tackling fixed-point exponentiation. [14] provides several common fixed-point operations but does not describe exponentiation. [8] contains exponentiation operations for floating-point numbers as well as techniques to convert between floating and fixed-point numbers. However, both the floating-point exponentiation and the conversions between the two representation types require a full bit decomposition.

The most relevant comparison is to the protocol from the SCALE-MAMBA package [11]. The authors mention that they could not find a secure fixed-point exponentiation protocol in academic literature and resorted to constructing their own protocol for base 2 exponentiation. Their technique is as follows: First, the protocol compares the exponent to zero and proceeds differently depending on positive and negative exponents. Next, each exponent share is split into its integer and fractional parts and the exponentiation is done separately. For the integer part, earlier known techniques are used. The fractional computation is approximated using a degree-8 polynomial $P_{1045}(x)$, described by Hart [19]. Finally, the two computations are combined using a single multiplication, and a division is performed if the exponent was negative.

The overall structure is similar to our technique of splitting the computation into an integer and a fractional part. However, this approach has some notable drawbacks compared to our protocol. First, as far as we are aware, a comparison to zero requires a full bit decomposition of each parties shares, which as mentioned earlier, is expensive. In contrast, our work does not need to compare to zero to be able to handle negative exponents. Instead, here, we assume that the exponent is bounded from below, which allows us to convert the exponents to a suitable positive value beforehand. The extra multiplicative factor (2^A if A is the exponent bound) is later divided out. This assumption is suitable for our application, and results in a much faster protocol. Second, for the fractional exponentiation, SCALE-MAMBA uses a polynomial approximation, using a degree-8 polynomial, which will require several rounds of communication and/or a large communication cost. Here, for our protocol, we make a crucial observation that the fractional exponentiation (in real numbers) is small enough, such that the multiplicative shares of the product do not wrap around the fixed-

(l_f, l)	64-bit BaseInt			128-bit BaseInt		
	Offline	Online	ops/s	Offline	Online	ops/s
$(5, 32)$	3.09 μs	0.004 μs	323K	7.68 μs	1.01 μs	115K
$(10, 63)$	3.22 μs	0.99 μs	237K	9.24 μs	14.9 μs	41K
$(15, 63)$	3.22 μs	1.01 μs	236K	9.24 μs	16.0 μs	40K
$(20, 100)$	-	-		11.15 μs	33.2 μs	23K
$(20, 127)$	-	-		12.9 μs	54.9 μs	15K

Table 1: Benchmarks for exponentiation protocol, for base 64-bit and 128-bit int sizes. Offline and online phase times are averaged over 1 million runs. Throughput is given as average operations per second, rounded to the nearest thousand.

point ring. This allows us to directly compute shares of the fractional approximation without any polynomial approximation. Finally, we also realize that the integer and fractional parts can be combined locally first, giving rise to our 1-round protocol. For this, we do require a slightly larger ring (l_f more bits) to ensure that our intermediate computations can be appropriately represented, but we think that this tradeoff is appropriate. Furthermore, as we point out in Section 7.1, SCALE-MAMBA chooses a larger ring to reduce numerical instability that was observed, which already results in a larger ring size requirement than our protocol.

7 Experimental Evaluation

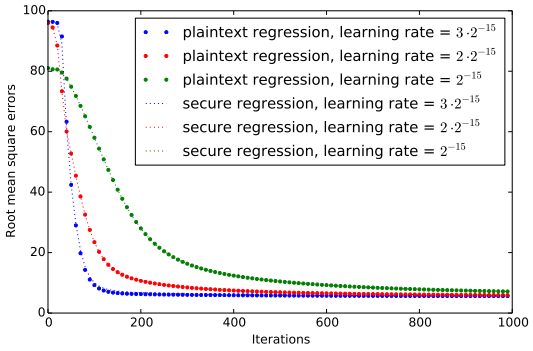
Implementation details. We implement our protocols in C++, and compile our code using the open-source Bazel [2] build tool. We support moduli up to 127-bit for both the fixed-point ring and the field. For the operations, we use the native C++ `uint64_t` type for moduli smaller than 64-bits, and `uint128` from Google's abseil library [3] for larger moduli. We give users the option to decide the base integer size (64-bit or 128-bit) and provide experimental results for both.

Experimental setup. We run all of our experiments on a compute-optimized c2-standard-8 Google cloud instance with 32 GB RAM. Our code is single-threaded and only uses a single core of the instance.

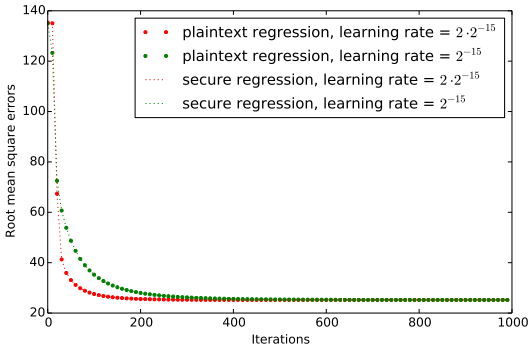
Result methodology. We provide both offline and online timing results for our overall Poisson regression protocol, as well as our secure exponentiation sub-protocol. For online timings, we report the maximum execution time between the two parties P_0 and P_1.

7.1 Secure Exponentiation Experiments

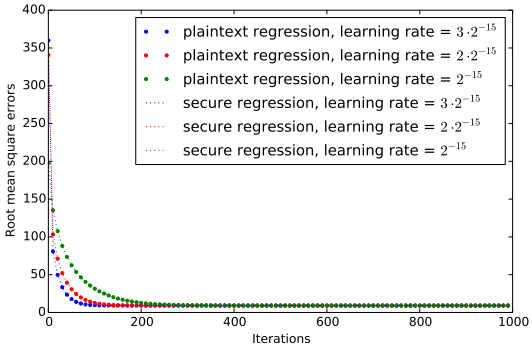
We benchmark our secure exponentiation protocol separately and compare its performance to the protocol from SCALE-MAMBA [11]. For our benchmarks, we choose several different parameters for (l, l_f) and use an l-bit prime q. We compute offline and online timings for both 64-bit and 128-bit base integer sizes. Our results are provided in Table 1. Offline time includes the time to generate all the preprocessed data. For online time, we time both P_0 and P_1 separately and and report the maximum running time between the two. All times are averaged over 1 million runs of



(a) Somoza



(b) Time to PhD

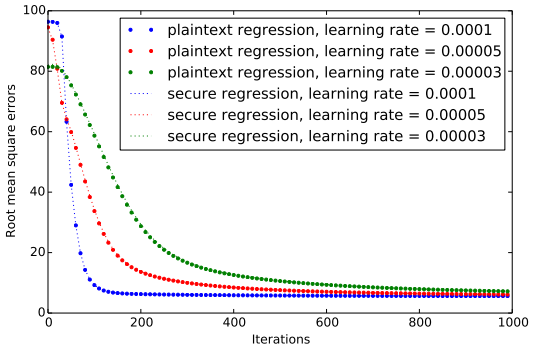


(c) Smoking and Cancer

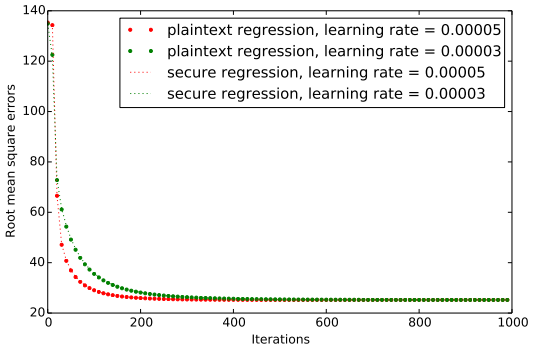
Figure 9: Convergence of the RMSE for plaintext regression versus Secure Poisson regression with 15-bit fixed-point precision.

the protocol. We also provide the throughout of our protocol, which is the number of operations that can be done sequentially in one second.

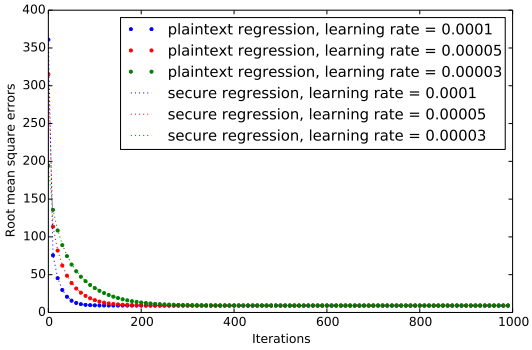
Comparison to related work. Aly and Smart [10] provide some benchmarks for the SCALE-MAMBA package. We only compare our protocol in the 2-party setting, but note that SCALE-MAMBA details a general n-party actively-secure protocol with abort and [10] also includes benchmarks for the 3-party full threshold, and honest-majority Shamir sharing settings. For fixed-point exponentiation with $l = 245$ and $l_f = 40$, [10] has an online runtime of 15 ms, an offline runtime of 18000 ms, and an offline cost of 1337 Beaver triples, 1 square tuple, and 7688 shared bits, which comes out to $\sim$2MB per exponentiation. In contrast, for those parameters, our total offline cost is only 980 bits, i.e., a 2000x improvement. Our implementation only supports a maximum of $l = 127$, and therefore our comparison is not direct, but for $(l_f, l) = (20, 127)$, our online runtime was just 0.055 ms, and our offline time was 0.013 ms. For comparable parameters, the offline time for the arctan operation in [10], which requires fewer preprocessed bits than exponentiation, is still around 7000 ms, which implies a more than 500,000x improvement. [10] notes that the reason large parameters were chosen specifically for exponentiation (as opposed to $(20, 128)$ for other functions like square-root, sine, cosine etc.), was the high numerical instability. This is not observed in our



(a) Somoza



(b) Time to PhD



(c) Smoking and Cancer

Figure 10: Convergence of the RMSE for plaintext regression versus Secure Poisson regression with 20-bit fixed-point precision.

protocol for the parameters $(20, 127)$. Our protocol also has a massive throughput gain. While [10] reports 76 ops/s for exponentiation when 50 invocations are run in parallel, we have ∼15,000 ops/s run sequentially for our 127-bit modulus. We also emphasize that given the low preprocessing size of our protocol when compared to others, it will be particularly efficient in the online-only setting.

For a 114-bit field, the semi-honest floating-point exponentiation protocol from [8] takes 96 ms. This does not account for the conversion between fixed-point and floating-point representations. Still, for a larger 127-bit modulus, our protocol takes 0.055 ms, which is a 1700x improvement.

7.2 Poisson Regression Experiments

We measure the performance of our secure Poisson regression protocol by comparing it with plaintext Poisson regression, where the data is provided without encryption. Our secure regression is implemented with fixed-point numbers, while the C++ `double` type is used in the plaintext version.

Datasets. We run our regression experiments on three datasets (detailed below) from the Princeton University course on Generalized Linear Models.

1. *Somoza.* This dataset contains infant and child survival rates in Colombia, based on the

Dataset	n	m	Dataset size	$(l_f, l) = (15, 63)$ Standard [23] Offline	Online	Comm.	Correlated Triples Offline	Online	Comm.
Replicated	100	10	9.6 KB	0.004	0.0007	0.04 MB	0.0009	0.0002	0.004 MB
		100	81.6 KB	0.032	0.004	0.36 MB	0.002	0.001	0.009 MB
		1000	0.8 MB	0.316	0.043	3.53 MB	0.0152	0.009	0.054 MB
	1000	10	96 KB	0.039	0.007	0.39 MB	0.009	0.002	0.043 MB
		100	0.82 MB	0.316	0.045	3.54 MB	0.020	0.011	0.076 MB
		1000	8.02 MB	3.046	0.493	35.0 MB	0.127	0.091	0.404 MB
	10000	10	0.96 MB	0.399	0.075	3.89 MB	0.090	0.027	0.433 MB
		100	8.16 MB	3.144	0.500	35.3 MB	0.195	0.108	0.749 MB
		1000	80.16 MB	30.797	5.47	349.7 MB	1.415	1.014	3.91 MB
Somoza	21	11	2.18 KB	0.0009	0.00007	9.51 KB	0.0002	0.00006	1.11 KB
PhD	73	17	11.1 KB	0.0045	0.0003	48.0 KB	0.0007	0.0002	3.69 KB
Cancer	36	14	4.6 KB	0.002	0.0013	20.0 KB	0.0004	0.0011	1.87 KB

Dataset	n	m	Dataset size	$(l_f, l) = (20, 127)$ Standard [23] Offline	Online	Comm.	Correlated Triples Offline	Online	Comm.
Replicated	100	10	9.6 KB	0.010	0.008	0.07 MB	0.004	0.008	0.008 MB
		100	81.6 KB	0.077	0.029	0.69 MB	0.019	0.028	0.017 MB
		1000	0.8 MB	0.761	0.243	6.62 MB	0.175	0.236	0.103 MB
	1000	10	96 KB	0.099	0.080	0.73 MB	0.038	0.080	0.077 MB
		100	0.82 MB	0.777	0.295	6.62 MB	0.191	0.287	0.139 MB
		1000	8.02 MB	7.550	2.444	65.6 MB	1.682	2.360	0.755 MB
	10000	10	0.96 MB	0.992	0.809	7.25 MB	0.388	0.802	0.769 MB
		100	8.16 MB	7.816	2.972	66.2 MB	1.886	2.876	1.361 MB
		1000	80.16 MB	76.13	25.01	655.2 MB	16.47	23.73	7.279 MB
Somoza	21	11	2.18 KB	0.0025	0.0017	17.7 KB	0.0010	0.0017	2.01 KB
PhD	73	17	11.1 KB	0.0115	0.0071	89.6 KB	0.0038	0.0070	6.62 KB
Cancer	36	14	4.6 KB	0.0050	0.0033	37.3 KB	0.0018	0.0032	3.37 KB

Table 2: Benchmarks for Poisson regression protocol, for different datasets for both $(l_f, l) = (15, 63)$ and $(l_f, l) = (20, 127)$ parameters. n is the number of examples and m is the number of features. For larger values of n and m, the Somoza dataset was replicated. Times (in seconds) are given per iteration of gradient descent over the entire dataset. For correlated triples, the time and communication is amortized over 100 iterations. All of our code is single threaded.

World Fertility Survey. Here, the survival is modeled as a function of the sex, cohort and age range. The dataset tracks infants over several years, and contains the exposure for each feature. Data for 2000 infants is present, and collapsed into combined exposures and counts for 21 distinct features.

2. *Time to PhD.* This dataset predicts PhD graduation as a function of years in graduate school, university, and residence status. We encode this dataset as 17 binary features. There are 73 distinct combinations of features in the dataset, and data from 35,000 PhD students is used to calculate the aggregate exposure period and graduation counts for each feature combination.

3. *Smoking and Cancer.* This dataset contains information from a Canadian study of mortality by age and smoking status. There are 14 different binary features, corresponding different

age buckets and smoking statuses. There are 73 distinct feature combinations, containing counts and exposure periods from a total of 92,000 respondents.

All of the datasets are publicly available at [1].

Performance evaluation. In terms of accuracy, we benchmark our secure Poisson regression protocol against plaintext poisson regression with different learning rates and fixed point precision. See Figure 9 and 10. We see that our secure protocol performs almost exactly as well as the plaintext regression: the lines plotted for model error versus number of iterations are nearly coincident.

When we take a closer look at the learned parameter $\boldsymbol{\theta}$, we find that the actual weights learned by the secure protocol are also nearly exactly the same as those from plaintext learning. See Table 3: the root mean square error between the secure weights and the plaintext weights is very small regardless of the dataset being tested on.

In terms of computation and communication efficiency, we benchmark our secure Poisson regression protocol on real and synthesized datasets. As there is no previous work done on secure Poisson regression, it is not possible for us to compare efficiency of our protocol with other work. Instead, we compare our protocol with a "basic" version that does not use correlated Beaver triples.

We benchmark our protocol for all our datasets in Table 2 for 2 different parameter choices: $(l_f, l) = (15, 63)$ and $(20, 127)$. The table contains the offline time, the online time, and the communication required by our protocol for each iteration of the gradient descent. The offline time denotes the total time to generate the Beaver triples. For the online time, we time the two parties P_0 and P_1 separately, and report the maximum time of the two.

We also run the same experiments for our batched multiplication optimization using correlated Beaver triples. Here, since the gain is only when multiple gradient descent iterations are run, for our timing values, we run 100 iterations, and report the amortized time for 1 iteration.

In addition to benchmarking our three datasets, we run all of our experiments for larger values of n and m. For this, we replicate the Somoza dataset to obtain a new dataset of the appropriate size $(n \times m)$. We report our timing results for this under the "Replicated" dataset header in Table 2.

We find that our protocols are very practical, even for larger datasets. For example, for a dataset with 10,000 elements and 100 features, our protocol has an amortized cost of 1.886 seconds of offline computation, 2.876 seconds of online computation, and 1.361 MB of communication. Over 100 iterations, the cost is approximately 3 minutes of offline computation, 5 minutes of online computation, and 136 MB of communication.

8 Applications

In this section, we give several concrete applications for Secure Poisson Regression, and discuss performance of our protocol in each of these scenarios.

8.1 COVID-19 case fatality rate

Recent work [30] performs an analysis of COVID-19 case fatality using Poisson Regression. They measure the effect of 9 binary variables on the counts of COVID-19 fatalities, using 2070 cases as training examples. Variables include age-range ($\geq$ 60 years), presence of cardiovascular disease, and presence of neurologic diseases The regression model is used to compute the incidence rate

Learning rate	Iterations	RMSE between plaintext weights and secure weights		
		Somoza	Time to PhD	Smoking and Lung Cancer
0.0001	100	0.00064	-	0.00016
	500	0.00259	-	0.00048
	1000	0.00456	-	0.00097
0.00005	100	0.00034	0.00031	0.00021
	500	0.00160	0.00123	0.00057
	1000	0.00346	0.00200	0.00150
0.00003	100	0.00029	0.00030	0.00023
	500	0.00131	0.00126	0.00060
	1000	0.00294	0.00228	0.00107

Table 3: Root mean square errors between the weights obtained from secure regression and those from plaintext regression. This table shows that the learned weights from secure regression are nearly the same as those obtained from plaintext regression.

ratio (IRR) for each variable, that is, the ratio between predicted fatalities when that variable is present versus not.

This case provides a good example for health-related data, where multiple hospitals may hold slices of the data, and may not want it to be centralized in the clear. To compute over this data privately, hospitals could send shares of the data to two servers who could perform Poisson Regression securely, and compute shares of the model parameters. The model could then be sent to each hospital which would individually compute the IRR for each variable on its examples, and release the aggregate IRRs.

On this dataset we estimate that Secure Poisson Regression would take 0.076 seconds of offline computation and 0.16 seconds of online computation per iteration of gradient descent, with a communication cost of 0.154 MB. Assuming 100 iterations of gradient descent are needed in order to converge, our protocol would require 7 seconds of offline computation and 16 seconds of online computation, with a communication cost of 15.4 MB.

8.2 Predicting credit default rates

[21] use Poisson Regression to model the rate of default payments by borrowers. They measure the effect of 6 variables, including income, age, monthly credit card expenditure, and home-ownership on the monthly rate of defaulted loan payments using a sample of 1002 individuals. After regression, the authors propose using the model inference to data of loan applicants to compute predicted defaults, and thereby characterize risk level.

This case involves training on sensitive financial data, which may be distributed across several institutions. Securely computing regression on these values would then consist of two phases: combining the records from multiple institutions, followed by performing secure regression on the joint data. The former task can be handled using techniques like privacy-preserving record linkage [6]. Our secure protocol is a good fit for the latter part, as well as the subsequent inference.

We estimate our protocol would incur 1.9 seconds of offline time and 4 seconds of online time to perform 100 iterations of secure gradient descent on this dataset, with a total communication cost of 3.8 MB. Each iteration would incur 0.019 seconds and 0.04 seconds of offline and online time,

with 0.038 MB of communication.

8.3 Modeling Ad campaign conversion rates

Google researchers [29] describe a system for measuring ad campaign conversion rates using Poisson regression. A "conversion" corresponds to an individual buying an item after seeing one or more ads. [29] give several ways to model multiple ad channels having a combined effect on an individual, with the ad effects decaying over time. One is to use a"step" decay: assigning each ad channel 3 binary attributes, corresponding to whether an individual was exposed to the ad in the short term (1 day prior), medium term (2-7 days prior) or long term (7-30 days prior). The conversion rate is then learned via Poisson regression using such attributes for some combination of ad channels. Credit for a conversion is proportionally distributed to each ad channel according to the relative change in predicted conversion rate when that ad channel is switched from exposed to unexposed. The total credit per ad channel is computed as the sum of its proportional credit across all conversions in the dataset.

This problem is an excellent case for the use of secure computation techniques, since it involves sensitive business and user data that may be held by different ad companies and transaction data providers. A secure solution would require privately joining the records, securely performing regression, and then securely computing the aggregate credit for each ad channel. The private join could be achieved using privacy-preserving record linkage techniques [6]. Our work is well-suited to performing the regression and the subsequent inference.

On a dataset with 5 ad channels and 3 binary attributes per channel for a total of 15 binary attributes, and assuming 100,000 training points, we estimate that our regression would take 5.82 seconds of offline time and 12.03 seconds of online time per iteration of gradient descent, with 11 MB of communication. For 100 iterations of gradient descent, we would incur 9.7 minutes of offline computation and 20 minutes of online computation, with 1.1 GB of communication.

9 Conclusion

Poisson regression is a widely used technique for modeling Poisson processes that occur across the life and social sciences. In many settings, the inputs for training Poisson models are sensitive health or financial data held by different parties. The secure Poisson regression protocol introduced in this paper enables computation on private data which reveals only the output Poisson model while protecting the inputs. Our construction achieves this with great efficiency while preserving accuracy comparable to computation in the clear. For several real datasets, this means execution in just a few seconds with a couple MB of communication. At the crux of our protocol is a new construction for secure fixed-point exponentiation and a new technique for correlated matrix multiplication, both of which are of independent interest with applications far beyond Poisson regression.

References

[1] `https://data.princeton.edu/wws509/datasets`.

[2] `https://github.com/bazelbuild/bazel`.

[3] `https://github.com/abseil/abseil-cpp`.

[4] Nitin Agrawal et al. “QUOTIENT: Two-Party Secure Neural Network Training and Prediction”. In: CCS. 2019, pp. 1231–1247.

[5] Nitin Agrawal et al. “QUOTIENT: Two-Party Secure Neural Network Training and Prediction”. In: *CCS*. 2019, 1231–1247.

[6] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. “Information sharing across private databases”. In: *SIGMOD*. 2003, pp. 86–97.

[7] Martin R Albrecht, Rachel Player, and Sam Scott. “On the concrete hardness of learning with errors”. In: *Journal of Mathematical Cryptology* 9.3 (2015), pp. 169–203.

[8] Mehrdad Aliasgari et al. “Secure Computation on Floating Point Numbers”. In: *NDSS*. 2013.

[9] Abdelrahaman Aly, Aysajan Abidin, and Svetla Nikova. “Practically Efficient Secure Distributed Exponentiation without Bit-Decomposition”. In: *FC*. 2018, pp. 291–309.

[10] Abdelrahaman Aly and Nigel P. Smart. *Benchmarking Privacy Preserving Scientific Operations*. Cryptology ePrint Archive, Report 2019/354. 2019.

[11] Abdelrahaman Aly et al. *SCALE-MAMBA v1.10: Documentation*. `https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf`. 2020.

[12] Donald Beaver. “Efficient Multiparty Protocols Using Circuit Randomization”. In: *CRYPTO*. Ed. by Joan Feigenbaum. 1992, pp. 420–432.

[13] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) Fully Homomorphic Encryption without Bootstrapping”. In: *ACM Trans. Comput. Theory* 6.3 (2014).

[14] Octavian Catrina and Amitabh Saxena. “Secure Computation with Fixed-Point Numbers”. In: *FC*. 2010, pp. 35–50.

[15] Ivan Damgård et al. “Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation”. In: *TCC*. 2006, pp. 285–304.

[16] E. L. Frome. “The Analysis of Rates Using Poisson Regression Models”. In: *Biometrics* 39.3 (1983), pp. 665–674.

[17] Hossein Ghodosi, Josef Pieprzyk, and Ron Steinfeld. “Multi-party computation with conversion of secret sharing”. In: *Des. Codes Cryptogr.* 62.3 (2012), pp. 259–272.

[18] Oded Goldreich. *Foundations of Cryptography: Basic Applications*. Vol. 2. Cambridge University Press, 2004.

[19] John F. Hart. *Computer Approximations*. Krieger Publishing Co., Inc., 1978.

[20] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “On ideal lattices and learning with errors over rings”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2010, pp. 1–23.

[21] Sami Mestiri and Abdeljelil Farhat. “Using Non-Parametric Count Model for Credit Scoring”. In: *SSRN Electronic Journal* (Oct. 2019).

[22] Payman Mohassel and Peter Rindal. “ABY^3: A Mixed Protocol Framework for Machine Learning”. In: *CCS*. 2018, pp. 35–52.

[23] Payman Mohassel and Yupeng Zhang. “SecureML: A System for Scalable Privacy-Preserving Machine Learning”. In: *IEEE SP*. 2017, pp. 19–38.

[24] Jesper Buus Nielsen et al. “A new approach to practical active-secure two-party computation”. In: *CRYPTO*. 2012, pp. 681–700.

[25] Chao Ning and Qiuliang Xu. “Constant-Rounds, Linear Multi-party Computation for Exponentiation and Modulo Reduction with Perfect Security”. In: *ASIACRYPT*. 2011, pp. 572–589.

[26] Deevashwer Rathee, Thomas Schneider, and K. K. Shukla. “Improved Multiplication Triple Generation over Rings via RLWE-Based AHE”. In: *CANS*. 2019, pp. 347–359.

[27] M. Sadegh Riazi et al. “XONN: XNOR-based Oblivious Deep Neural Network Inference”. In: *USENIX Security*. 2019, pp. 1501–1518.

[28] Sean Richey. “Who votes alone? The impact of voting by mail on political discussion”. In: *Australian Journal of Political Science* 40.3 (2005), pp. 435–442.

[29] Dinah Shender et al. *A Time To Event Framework For Multi-touch Attribution*. 2020. arXiv: 2009.08432.

[30] G. J. B. Sousa et al. “Mortality and survival of COVID-19”. In: *Epidemiology and Infection* 148 (2020).

[31] “The relationship between truck accidents and geometric design of road sections: Poisson versus negative binomial regressions”. In: *Accident Analysis and Prevention* 26.4 (1994), pp. 471–482.

[32] Sameer Wagh, Divya Gupta, and Nishanth Chandran. “SecureNN: 3-Party Secure Computation for Neural Network Training”. In: *Proc. Priv. Enhancing Technol.* 2019.3 (2019), pp. 26–49.

[33] Ching-Hua Yu et al. “Efficient Secure Two-Party Exponentiation”. In: *CT-RSA*. 2011, pp. 17–32.