Secure Poisson Regression

Mahimna Kelkar, *Cornell Tech;* Phi Hung Le, Mariana Raykova, and Karn Seth, *Google*

https://www.usenix.org/conference/usenixsecurity22/presentation/kelkar

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

Secure Poisson Regression

Mahimna Kelkar* *Cornell Tech*
Phi Hung Le* *Google*
Mariana Raykova *Google*
Karn Seth *Google*

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. In a LAN setting, our secure exponentiation for 20-bit fractional precision takes less than 0.07ms with a batch-size of 100,000. One iteration of secure Poisson regression on a dataset with $10,000$ samples with 1000 binary features needs about 65.82s in the offline phase, 55.14s in the online phase and 17MB total 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, 19, 24, 28]. 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. 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. Some 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 losses and claims in insurance policies in a certain period of time, and the number of purchases a user makes after being shown 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 expected response 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 [13], 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 [25], the effect of age, gender, preexisting conditions such as diabetes and obesity on the mortality rate from COVID-19 [27], predicting the number of payment defaults in credit scoring based on socio-economic characteristics [18], and the

*Part of this work was done during an internship at Google.

number of purchases that users make influenced by online advertisements they have been shown [26].

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 to compute the final Poisson model. We assume the most general setting where the two 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. A key component of Poisson regression involves the exponential (e^x) function. 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, 19], 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. [14] that relies on a small amount of offline precomputation and a single round of online computation.

Since there are no prior works that consider (fixed-point) exponentiation in the two-party semi-honest model, as comparison points, we consider state-of-the-art works that achieve a similar functionality in the malicious setting [9, 10] or in the semi-honest setting for $n \geq 3$ parties using a floating-point representation [7]. Although the comparison is not direct (see Section 7.1 for details), our protocol achieves orders of magnitude improvement on both the online throughput and the offline cost which indicates the possibility of substantial gains even when comparing in the same setting. 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. Poisson regression makes extensive use of matrix multiplications. For secure multiplication on shared values, a well-known work [11] uses precomputed random *Beaver* triples followed by a single online communication round. In similar fashion, state of the art techniques for secure matrix multiplication [19] generalize Beaver triples to matrices and optimize the online communication and amount of preprocessing required; only one (matrix) Beaver triple is required for each matrix multiplication.

We make the observation that the matrix multiplication operations used in the Poisson regression training have a spe-

cific structure that can be exploited to further optimized the communication cost of the matrix multiplications: the same matrix $\mathbf{X}$ is used in many multiplications with many different matrices $\mathbf{Y}_i$. 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 the online phase: the communication cost is reduced by up to a factor of $(n+1)$ (where n is the number of training samples; see Section 5). Thanks to our very efficient secure exponentiation, the dominant cost (more than 90% for both computation and communication) in the secure Poisson regression protocol comes from secure matrix multiplication operations. 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 protocol 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, in a LAN setting with 1.5GB/s bandwidth, secure exponentiation for shared exponents with 20-bit precision takes less than 0.07ms when 100K evaluations are batched for communication. SCALE-MAMBA [10], which offers malicious security but is our most relevant point of comparison, 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 phase is even greater, where our protocol requires 2000x less offline preprocessing and has a $500,000$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. In our LAN setting, the total training (with 1000 iterations) for each of the three datasets takes less than 8s in the online phase, 120s in the offline phase, and 121MB total communication. 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, and evaluation with a 127-bit modulus and 20-bit fractional precision, one training iteration requires 65.82s in the offline phase, 23.73s in the online phase and 17MB total communication. 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

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

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 expected response is modeled as a Poisson distribution, and therefore, $g(\mathbf{x}_i) = e^{\langle \theta, \mathbf{x}_i \rangle}$, where θ 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 techniques used to train a model iteratively. A model can be defined by a set of parameters $\theta = (\theta_1, \cdots, \theta_m)$. To learn the model parameters from data, the algorithm iteratively attempts to minimize a predetermined convex function. At each step, the parameters are updated based on the gradient.

In this paper, we focus specifically on Poisson regression with *exposure*, which allows for modeling of rate-based data. 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). Poisson regression attempts to learn model parameters θ, by minimizing $-L(\theta|\mathbf{X}, \mathbf{Y}, \mathbf{T})$ where $L(\cdot)$ is the log likelihood function. For this, the gradient will be computed as:

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

where $(\mathbf{x}_i, y_i, t_i)$ is the i^{th} data point. The training will now update the parameters iteratively. For the $(k+1)^{\text{th}}$ iteration,

$\theta^{(k+1)}$ is computed as follows:

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

where the exponential function is applied to each element in $\mathbf{X}\theta^{(k)}$, $\circ$ is the Hadamard (element-wise) product and the constants α and β denote the learning rate and the regularization parameter respectively. $\theta^{(0)}$ is usually initialized either 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 only the final output is revealed to them. Our protocol constructions are in a two-party setting and provide semi-honest security [15], 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 use the notation $F([\![x]\!], [\![y]\!])$ to denote 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.

Multiplication using Beaver triples. Given $[\![x]\!]$ and $[\![y]\!]$ (over $\mathbb{Z}_N$), a common technique to compute $[\![z]\!] = [\![xy]\!]$ is Beaver's multiplication trick [11]. For this, a randomly sampled *Beaver triple* $([\![a]\!], [\![b]\!], [\![c]\!])$, where $c = ab \bmod N$ is provided to the two parties. Now, P_0 and P_1 first locally compute $[\![u]\!] = [\![x]\!] - [\![a]\!]$ and $[\![v]\!] = [\![y]\!] - [\![b]\!]$. Next, they reconstruct u and v by communicating their share to the other party. Finally, P_i can compute $[\![z]\!]_i = i \cdot uv + u[\![b]\!]_i + v[\![a]\!]_i + [\![c]\!]_i$. Note that the same technique works for multiplying fixed-point numbers. 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.

An optimization in [19] shows that for secure matrix multiplication, given $[\![\mathbf{X}]\!]$ and $[\![\mathbf{Y}]\!]$ where $\mathbf{X}$ has dimension $n \times m$ and $\mathbf{Y}$ has dimension $m \times k$, the preprocessing and online costs per party are $(nm + mk)$ rings and $(n+m)k$ group elements respectively. This involves sharing a *matrix* Beaver triple $([\![\mathbf{A}]\!], [\![\mathbf{B}]\!], [\![\mathbf{C}]\!])$ where $\mathbf{A}$ and $\mathbf{B}$ are matrices with the same dimension as $\mathbf{X}$ and $\mathbf{Y}$ respectively, and $\mathbf{C} = \mathbf{AB} \bmod N$. We use $\mathcal{F}_{\mathsf{matMult}}([\![\mathbf{X}]\!], [\![\mathbf{Y}]\!])$ 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. A background on Ring-LWE is provided in Appendix B.

3 Secure Computation over FP 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. For secure computation, while there are techniques that emulate floating point representation [7], they are often expensive. A more efficient approach that is commonly used is 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 [19], we will compute over fixed-point numbers mapped onto an integer ring.

Fixed-point ring. A fixed-point ring $\mathcal{R}$ is a tuple $(\mathbb{Z}_{2^l}, l_x, l_f)$ where 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 $\mathcal{R}_*^+ = [0, 2^{l_x})$ and negative numbers will be in the range $\mathcal{R}_*^-(2^l - 2^{l_x}, 2^l)$ in their two's complement representation. $\mathcal{R}_* = \mathcal{R}_*^+ \cup \mathcal{R}_*^-$ is the total part of $\mathcal{R}$ wherein the fixed-point numbers are represented.

For a real r, with $|r| < 2^{l_x - l_f}$, we 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 $\mathcal{R} = (\mathbb{Z}_{2^{10}}, 3, 2)$, $x = 1.25$ 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 $x \in \mathcal{R}_*$, 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 P_0 and P_1. 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 $x \in \mathbb{Z}_{2^l}$. With this notation, we now define some basic useful secure ring operations.

1. Basic operations:
   - (Addition). Given shared values $[\![x]\!]$ and $[\![y]\!]$, $\mathsf{Add}([\![x]\!], [\![y]\!])$ outputs $[\![x+y]\!]$.
   - (Multiplication). 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 in one round using Beaver triples.
2. Fixed-Point operations: These operations are for elements in $\mathcal{R}_*$. Intuitively, the functionality here 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). 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). 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). 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). Given $[\![x]\!]$ and a public positive integer $c \in \mathbb{Z}^+$, $\mathsf{PubFPDiv}([\![x]\!],c)$ outputs $\left[\!\left[\widehat{(\underset{\sim}{x})/c}\right]\!\right]$.
- (FP Exponentiation). 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 reals. 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, FPAdd can be done non-interactively, and FPMult can be done using Beaver triples. Due to truncation, FPMult 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 with an error of at most 2^{-l_f}, and we provide protocols to do so in Appendix A. 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 FP 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 $\mathbb{F}_q$.

Failure probability and approximation errors. The secure computation of FP operations may come inbuilt with some probability of failure as well as errors as a result of truncation. We say that a protocol has failure probability p_{fail} and error ε if, except with probability p_{fail}, the error in the underlying fixed-point computation is bounded by ε. The failure probability, similar to e.g., [19] can be made arbitrarily small by increasing the gap between l and l_x (see Appendix A).

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. The protocol is detailed in Appendix A.

4 Secure Poisson Regression Protocol

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.

4.1 Basic Design

The goal of the regression protocol is to output a sharing of a weights vector θ. 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. We chose to go with standard gradient descent in this paper, but note that our protocol can be adapted for any variant.

Recall that in the update step of our gradient descent, the weights for the $(k+1)^{\text{th}}$ iteration are updated as follows:

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

Let $[\![\theta^{(k)}]\!]$ denote a sharing of the weights vector after the k^{th} iteration. Parties start with $[\![\theta^{(0)}]\!]$ 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 $[\![\mathbf{U}]\!] = \left[\!\left[\mathbf{X}\theta^{(k)}\right]\!\right]$. (2) Next, each element in $\mathbf{U}$ 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 $[\![\mathbf{Z}]\!] = \left[\!\left[\mathbf{X}^T(\mathbf{Y}-\mathbf{W})\right]\!\right]$; (5) The remaining computations (public multiplication by α), and addition by $(1-\beta)\theta^{(k)}$ can be computed locally, to end up with shares of the updated weights $\theta^{(k+1)}$. 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

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

$$[\![\theta^{(k)}]\!] \leftarrow (1-\beta) \cdot [\![\theta^{(k-1)}]\!] + \alpha \cdot [\![\mathbf{Z}]\!]$$

Output. Party P_i outputs its share $[\![\theta^{(K)}]\!]_i$.

Figure 1: 2PC protocol for Secure Poisson Regression.

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 [19], 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, and defer the security proof that it does not leak any extra information about the multiplicands to the full version [16].

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.

We discuss additional considerations on efficiency like choosing the appropriate learning rate and using mini-batches for training in the full version [16].

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 [19]).

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 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 $e^{\mathbf{X}\theta^{(k)}}$ 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.

Secure Inference. A useful functionality, after the regression is complete, is to predict, or infer the value of the response variable for future samples. Formally, given a sharing $[\![\theta]\!]$ of the learned weights, the goal is evaluate a new sample $([\![\mathbf{x}]\!], [\![t]\!])$, i.e., compute a sharing of the response $y = e^{\mathbf{x}^T\theta}$. We defer the protocol to the full version [16].

5 Optimized Batched Multiplication

We now describe our optimization 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 where $\mathbf{X}$ has size $n \times m$, and all $\mathbf{Y}_i$ have size $m \times k$. P_i is provided shares $[\![\mathbf{X}]\!]_i, [\![\mathbf{Y}_1]\!]_i, \cdots, [\![\mathbf{Y}_K]\!]_i$, and the goal now is to compute shares

of the multiplications $[\![\mathbf{Z}_j]\!] = [\![\mathbf{X}\mathbf{Y}_j]\!]$ (for $j \in [1,K]$) more efficiently. For this, we will use correlated randomness across the multiplications and therefore need a single matrix sharing $[\![\mathbf{A}]\!]$ for the $\mathbf{X}$ multiplicand. We prove that this can be done securely in the full version [16]. Formally, our preprocessing requirement is now $[\![\mathbf{A}]\!], [\![\mathbf{B}_1]\!], \cdots, [\![\mathbf{B}_K]\!], [\![\mathbf{C}_1]\!], \cdots, [\![\mathbf{C}_K]\!]$. 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.

The structure of the correlated Beaver triples allows them to be efficiently generated in the offline phase via the use of Ring-LWE. We detail the generation protocol in Section 5.1.

Online cost improvement. Correlated Beaver triples improve the cost of our protocol significantly. In the online phase, since $\mathbf{X}-\mathbf{A}$ only needs to be reconstructed once instead of for each multiplication, the amortized online communication per multiplication for our technique is $\frac{2nm}{K} + 2mk$ ring elements, compared to $2nm + 2mk$ using standard matrix Beaver triples from [19]. In the setting of Poisson regression, since $k = 1$ typically, this results in a $\frac{n+1}{(n/K)+1}$ factor improvement for the online phase, which is very close to $(n+1)$ when K is large. For example, if the number of training samples $n = 1000$ and the model is trained over $K = 1000$ iterations, the communication cost of the online phase is reduced by 500 times if correlated Beaver triples are used.

5.1 Improving The Offline Phase

Without any need for optimization, the correlated Beaver triples can be generated in the offline phase using the two approaches from SecureML [19]: OT-based and additive homomorphic encryption (AHE) based. The latter, which uses Paillier encryption, requires 190x less communication than the former, but is more expensive computationally. Experiments from [19] show that the AHE-based approach is better in WAN network, while the OT-based is 20-30x faster in LAN setting (See Table 2 in [19]).

We show how to significantly improve triple generation via the use of Ring-LWE. Our approach works for any ring $\mathbb{Z}_N$ ($N = 2^l$ in our case) and does not rely on packing techniques for $\mathbb{Z}_{2^l}$ as in [22] (where the number of slots is only $\phi(m)/5$, resulting in 80% space being wasted) or on the embedding of plaintext values in a larger prime field of length $2 \cdot l + \sigma + 2$ as in [23] (which increases the communication and computation cost by at least $(2 + (\sigma+2)/l)$ times).

In more detail, to generate K correlated Beaver triples, our protocol proceeds as follows: First, P_0 and P_1 sample random matrices to be shares of $\mathbf{A}$ and $\mathbf{B}_1, \cdots, \mathbf{B}_K$. Let the shares held by P_i be $[\![\mathbf{A}]\!]_i, [\![\mathbf{B}_1]\!]_i, \cdots, [\![\mathbf{B}_K]\!]_i$. In order to obtain the shares of $\mathbf{C}_j = \mathbf{A}\mathbf{B}_j$, the parties need to compute the shares of $[\![\mathbf{A}]\!]_i [\![\mathbf{B}_j]\!]_{1-i}$ as $[\![\mathbf{A}]\!]_i [\![\mathbf{B}_j]\!]_i$ can then be computed locally by each party P_i (since $[\![\mathbf{C}_j]\!] = [\![\mathbf{A}\mathbf{B}_j]\!] = [\![[\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_0 + [\![\mathbf{A}]\!]_1 [\![\mathbf{B}_j]\!]_1 + [\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1 + [\![\mathbf{A}]\!]_1 [\![\mathbf{B}_j]\!]_0]\!]$). We propose two different ways to compute the shares of $[\![\mathbf{A}]\!]_i [\![\mathbf{B}_j]\!]_{1-i}$. The first approach works better when the number of training samples n is large while the second approach works better when the number of explanatory variables m is small. We benchmark the cost to generate triples using both approaches in Table 3 (Section 7.2), and also compare to the Paillier encryption based approach used in [19]. Overall, both of our approaches are significantly better than Paillier encryption in terms of both communication and computation.

Approach I. P_0 encrypts each column of the the matrix $[\![\mathbf{A}]\!]_0$ separately using Ring-LWE and sends the encrypted columns to P_1. Define $[\![\mathbf{B}_j]\!]_1 = (b_{1j}, \cdots, b_{mj})^T$, $\mathbf{A}_i$ as the i^{th} column of $[\![\mathbf{A}]\!]_0$, and $\mathbf{E}_i$ as $\mathrm{Enc}_{sk}(\mathbf{A}_i)$ for $1 \le i \le m$. P_1 uses the additive homomorphic properties of Ring-LWE to compute the encryption $\mathbf{D}_j = \sum_{i=1}^{m} b_{ij}\mathbf{E}_i + \mathbf{R}_j = \mathrm{Enc}([\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1 + \mathbf{R}_j)$. P_1 sends the ciphertexts to P_0 who decrypts them to obtain $[\![[\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1]\!]_0 = [\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1 + \mathbf{R}_j$ while P_1 has $[\![[\![\mathbf{A}]\!]_0 [\![\mathbf{B}_j]\!]_1]\!]_1 = -\mathbf{R}_j$. If the number of training samples n is much smaller than the length of the ciphertext (say N, the degree of the cyclotomic polynomial used in the Ring-LWE scheme), P_1 can pack multiple $\mathbf{D}_j$ into a single ciphertext to optimize communication. Assuming $N = 2n$, $\mathbf{D}_0 = \mathrm{Enc}(d_1, \cdots, d_n, 0, \cdots, 0)$, and $\mathbf{D}_1 = \mathrm{Enc}(d'_1, \cdots, d'_n, 0, \cdots, 0)$, we can produce the ciphertext $\mathbf{D}'_1 = \mathrm{Enc}(0, \cdots, 0, d'_1, \cdots, d'_n)$ by multiplying $\mathbf{D}_1$ with the plaintext message $(0, \cdots, 0, 1, 0, \cdots, 0)$ which is zero everywhere except for the i^{th} position. Now, $\mathbf{D}_{01} = \mathbf{D}_0 + \mathbf{D}'_1$ is the ciphertext containing $(b_1, \cdots, b_n, b'_1, \cdots, b'_n)$. Similarly, if $t = N/n$, we can pack t ciphertexts $\mathbf{D}_j$ into one ciphertext. The parties now reverse roles to compute shares of $[\![\mathbf{A}]\!]_1 [\![\mathbf{B}_j]\!]_0$, and finally shares of $\mathbf{A}\mathbf{B}_j$.

To analyze the efficiency of this approach, first notice that each party sends m ciphertexts and receives $K \cdot n/N$ ciphertexts. Now, if $\log(q)$ is the bitlength of the Ring-LWE ciphertext modulus, then the cost to generate K correlated Beaver triples is $2(m + Kn/N)N(2\log(q)) = 4(mN + Kn)\log(q)$ bits. When K is very large ($\gg mN$), the amortized cost per triple is roughly $4n\log(q)$ bits. For 127-bit input, we use a ciphertext modulus with length $\log(q) = 295$. For Paillier encryption, the amortized cost per triple is $2(m+n)\log(q')$ bits where $q' = 6144$ is the length of the Paillier ciphertext. Since $m < n$ is typical for training data, our protocol uses at least 10x less bandwidth than the AHE approach from [19]. Besides the smaller communication cost, our approach also provides significant gains in the computation time. To multiply a constant with a ciphertext in Ring-LWE we only need to perform multiplications over field of size 295 bits. However, if Paillier encryption is used, an exponentiation in group of size 6144 bits needs to be computed which is much more expensive.

In terms of computational cost, our protocol requires $m \times K$ multiplications between a scalar and a ciphertext and $K(1 - n/N)$ shift operations.

Approach II. While the previous approach is efficient in terms of communication, it results in a lot of wasteful computation if the number of training samples n is much smaller than the degree N of the cyclotomic polynomial used for Ring-LWE. Our second approach therefore, will be geared towards settings when $n \ll N$.

For this, P_0 first encrypts each row of the matrix $\mathbf{B} = (\llbracket \mathbf{B}_1 \rrbracket_0, \cdots, \llbracket \mathbf{B}_K \rrbracket_0)$ separately and sends the ciphertexts $\mathbf{E}^j \leftarrow \mathrm{Enc}_{sk}(\mathbf{B}^j)$ for $1 \le j \le m$ to P_1. Now, P_1 uses the additive property of Ring-LWE to compute $\mathbf{D}^i = \sum_{j=1}^{m} a_{ij}\mathbf{E}^j + \mathbf{R}^i$ for $1 \le i \le n$, where $\mathbf{D}^i$ is the encryption of the i^{th} row of the matrix $\llbracket \mathbf{A} \rrbracket_1 \mathbf{B} + \mathbf{R}$ and $\mathbf{R}$ is a random matrix sampled by P_1, and sends the ciphertexts to P_0. Note that when K is much smaller than N, P_1 can pack multiple ciphertexts into one before sending them back to P_0 to reduce the communication cost. The packing is done by simply shifting the ciphertexts as described in the first approach. P_0 now decrypts the ciphertexts to obtain $\llbracket \llbracket \mathbf{A} \rrbracket_1 \llbracket \mathbf{B}_j \rrbracket_0 \rrbracket_0 = \llbracket \mathbf{A} \rrbracket_1 \llbracket \mathbf{B}_j \rrbracket_0 + \mathbf{R}_j$, while P_1 sets $\llbracket \llbracket \mathbf{A} \rrbracket_1 \llbracket \mathbf{B}_j \rrbracket_0 \rrbracket_1 = -\mathbf{R}_j$. Similar to the first approach, the two parties now reverse roles to compute shares of $\llbracket \mathbf{A} \rrbracket_1 \llbracket \mathbf{B}_j \rrbracket_0$, and finally shares of $\mathbf{A}\mathbf{B}_j$.

Assume $K \ll N$ (in our experiments, $K = 1000$ and $N = 2^{14}$). To analyze the efficiency of this approach, first notice that P_1 sends m ciphertexts to P_1 and receives $n \cdot K/N$ ciphertexts. The communication cost to generate K correlated Beaver triples is therefore $2(m + n \cdot K/N)N(2\log(q)) = 4(mN + nK)\log(q)$ bits. When $mN < nK$ (for example, $n = 1000, m = 10, N = 2^{14}, K = 1000$), the amortized cost for one triple is less than $8n\log(q)$ bits, which is around 5x cheaper than the AHE-based approach from [19]. In terms of computation, our protocol requires $n \times m$ multiplications between a scalar and a ciphertext and $n(1 - K/N)$ shift operations. The second approach is faster than the first one when $n < K$.

In our secure Poisson regression protocol, we also need to generate the correlated Beaver triples for the multiplication between shares of scalars $\mathbf{T}_i$ and $\mathbf{V}_i$ where $\mathbf{T}_i$ is fixed during the training process. This is equivalent to having $n = m = 1$, so the second approach will be used to generate these correlated Beaver triples.

6 Secure Fixed-Point Exponentiation

In this section, we detail our novel secure fixed-point exponentiation protocol. To simplify our analysis, our protocol will mirror $\mathcal{F}_{\mathsf{FPexp}}$ functionality (Figure 2) 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 6.4) 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 $\llbracket x \rrbracket$, $\mathcal{F}_{\mathsf{FPexp}}(b, \llbracket x \rrbracket)$ will compute a sharing of something "close" to $\widehat{(b)^{(x)}}$. We benchmark our protocol and compare it to existing works in Section 7.1. A more in-depth comparison to related techniques is also provided in the full version [16].

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 use a 2-party variant of the efficient MTA (multiplicative to additive) protocol from Ghodosi et al. [14]. Suppose that P_0 and P_1 hold multiplicative shares m_0 and m_1 of a secret s in $\mathbb{F}_q$. 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. Then, 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 $a_0 + a_1 = \alpha_0 m_0 \alpha_1 m_1 + \beta_1 m_1 \beta_0 m_0 = m_0 m_1(\alpha_0\alpha_1 + \beta_0\beta_1) = s$. [14] also shows that the shares are individually uniformly random.

The source of the preprocessed values is not provided in [14] 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 to Section 6.4. We begin with a simplified version of our protocol where

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 $\underset{\sim}{x}\log_2(b) > 1 - A$

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 2: Functionality $\mathcal{F}_{\mathsf{FPexp}}$

$\underset{\sim}{b} = 2$, and the exponent satisfies $\underset{\sim}{x} > 1$, and handle other exponents and other (positive) bases later.

Our strategy is 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 step 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 two steps there reduce the problem to a base 2 exponentiation (of a positive exponent). The party P_i first splits its share $[\![z]\!]_i$ as $(z_i^{\text{int}}, z_i^{\text{frac}})$ where $z_i^{\text{int}} = \left\lfloor [\![z]\!]_i / 2^{l_f} \right\rfloor$ and $z_i^{\text{frac}} = [\![z]\!]_i / 2^{l_f} - z_i^{\text{int}} = ([\![z]\!]_i \bmod 2^{l_f})/2^{l_f}$. Notice now that $z = ([\![z]\!]_0 + [\![z]\!]_1 \bmod 2^l) = 2^{l_f}\left((z_0^{\text{int}} + z_1^{\text{int}} \bmod 2^{l-l_f}) + (z_0^{\text{frac}} + z_1^{\text{frac}})\right)$. Therefore,

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

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 $[\![w]\!]_0 = z_0^{\text{int}}$ and $[\![w]\!]_1 = z_1^{\text{int}}$ form a sharing of $w = (z_0^{\text{int}} + z_1^{\text{int}} \bmod 2^{l-l_f})$ 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 [8, 21, 29]) 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 $\mathsf{RingChange}$ 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 $[\![w]\!]^{\mathbb{Z}_{q-1}}$ will satisfy $w + (q-1) = [\![w]\!]_0^{\mathbb{Z}_{q-1}} + [\![w]\!]_1^{\mathbb{Z}_{q-1}}$. Now, the two parties can exponentiate their shares locally $(\bmod\ q)$ to directly get a multiplicative sharing of 2^w. This works since,

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

where the last step is due to Fermat's little theorem. Let $v_0^{\text{int}} = 2^{[\![w]\!]_0^{\mathbb{Z}_{q-1}}} \bmod q$ and $v_1^{\text{int}} = 2^{[\![w]\!]_1^{\mathbb{Z}_{q-1}}} \bmod q$ 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 $\lfloor 2^{l_f} \cdot v_i^{\text{frac}} \rfloor$. A crucial observation here is that since $2^0 \leq v_i^{\text{frac}} < 2^1$, $\lfloor 2^{l_f} \cdot v_i^{\text{frac}} \rfloor$ 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 $d_i = \lfloor 2^{l_f} \cdot v_i^{\text{frac}} \rfloor$. 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 PubFPDiv. Finally, both parties can locally use the 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 6.4.

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. 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 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 the exponent is negative; we can simply add the 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 [4, 19].

Failure probability. We analyze the total failure probability of our base 2 exponentiation protocol. First, suppose that the (positive) base 2 exponent z is secret shared as $([\![z]\!]_0, [\![z]\!]_1)$. With probability at least $1 - 2^{l_x - l}$, we have $[\![z]\!]_0 + [\![z]\!]_1 = z + 2^l$, 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 $\mathbb{Z}_{q-1}$ (see Appendix A). 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 Other Considerations

Alternate 2-round protocol. We also describe an 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.

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 [20] 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 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 $[\![z]\!]$ 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 $[\![z]\!]_i$, 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,

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

where the $\bmod q$ 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,

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

This gives,

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

Now, $y \leftarrow \mathsf{PubFPDiv}([\![y']\!], 2^{l_f})$ 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 $[\![\widehat{a}]\!], [\![\widehat{b}]\!], [\![\widehat{c}]\!], [\![\widehat{d}]\!]$ 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.

7 Experimental Evaluation

Implementation details. We implemented our protocols in C++, and compiled the 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 ran all of our experiments on two c2-standard-8 Google cloud instances with 3.1 GHz base frequency and 32 GB RAM. Our code is single-threaded and only uses a single core. For the LAN setting, both instances were deployed in the us-central1 region where the mean network latency was 0.15ms and the bandwidth was about 1.5GB/s. For the WAN setting, one instance was in us-central1 while the other was in us-west2; the mean network latency was 49ms and the bandwidth was about 50MB/s.

7.1 Secure Exponentiation Experiments

We benchmark our secure exponentiation protocol separately and present our results and comparisons here.

Timing experiments. We provide the offline and online computation times as well as end-to-end benchmarks (both LAN and WAN) for several (l, l_f) parameters and for both 64-bit and 128-bit base integer sizes. The results are shown in Table 1. We find that especially when batching the communication for several exponentiations together, the impact of the network is quite minimal, primarily due to the small amount of communication our protocol requires.

(l_f, l)	64-bit BASEINT				128-bit BASEINT			
	Offline	Online	End-to-End		Offline	Online	End-to-End	
			LAN	WAN			LAN	WAN
(5,32)	3.09	0.004	4.21	11.06	7.68	1.01	9.56	16.29
(10,63)	3.22	0.99	5.43	12.07	9.24	14.9	25.02	32.17
(15,63)	3.22	1.01	5.49	12.11	9.24	16.0	26.29	32.95
(20,100)	-	-	-	-	11.15	33.2	44.91	52.35
(20,127)	-	-	-	-	12.9	54.9	68.63	75.91

Table 1: Timing benchmarks (in μs) for the exponentiation protocol, for base 64-bit and 128-bit int sizes. Exponents in the range $[-5, 5]$ were randomly sampled and shared in the fixed-point ring. Offline and online phase computation times (in μs) are averaged over 1 million runs, and don't include communication. End-to-end times per exponentiation are given in the LAN and WAN settings where 100K exponentiations are batched for communication. End-to-end times include computation and communication costs for both the online and offline phases.

l_f	Our approach		Polynomial Approx. [9]	
	μ	σ	μ	σ
5	0.0286	0.011	0.0932	0.0202
10	0.0009	0.0003	0.0051	0.0019
20	9.2×10^{-7}	3.5×10^{-7}	6.6×10^{-6}	5×10^{-6}
30	9.1×10^{-10}	3.5×10^{-10}	1.3×10^{-8}	2×10^{-8}
40	8.9×10^{-13}	3.4×10^{-13}	1.8×10^{-9}	7.4×10^{-9}

Table 2: Mean (μ) and standard deviation (σ) of the fractional exponentiation error as a ratio of the actual result for both our approach and polynomial approximation (as in [9]). Exponents are sampled and shared randomly. The error is averaged over 1 million runs.

Accuracy experiments. As mentioned earlier, our exponentiation has smaller error than standard techniques. Since, the error comes only from the fractional part, we implement both our fractional exponentiation as well as a degree-9 polynomial approximation used in [9] and compare the errors in Table 2. Our errors are smaller by 1 to 2 orders of magnitude and the difference gets wider with more fractional bits.

Apart from the smaller errors, we also note that our technique only requires a single round of communication while a degree-d polynomial approximation usually takes d rounds when implemented using Horner's method (as in done in [9] to reduce total communication). One shortcoming however, is that while it is straightforward to extend polynomial approximation to the malicious setting, it is not obvious how to efficiently do the same for our technique.

Comparison to related work. We did not find any prior work on fixed-point exponentiation that targets the same

Dataset	n	m	Paillier [19]			Correlated Triples (Approach I)			Correlated Triples (Approach II)		
			LAN (s)	WAN (s)	Comm.	LAN (s)	WAN (s)	Comm.	LAN (s)	WAN (s)	Comm.
Replicated	1	1	0.06563	0.0752	3 KB	0.0048	0.0051	0.89 KB	0.00007	0.00024	0.89 KB
	100	10	5.0371	5.3857	0.17 MB	0.0180	0.0192	17.8 KB	0.0024	0.0037	17.8 KB
		100	20.088	20.676	0.30 MB	0.1488	0.1495	44.4 KB	0.0179	0.0195	44.4 KB
		1000	171.18	171.85	1.65 MB	1.4597	1.4620	311 KB	0.1759	0.1814	311 KB
	1000	10	47.739	48.342	1.52 MB	0.0229	0.0263	151 KB	0.0227	0.0265	151 KB
		100	173.58	174.84	1.65 MB	0.1602	0.1642	178 KB	0.1601	0.1640	178 KB
		1000	1433.1	1434.4	3.00 MB	1.5889	1.6150	444 KB	1.6003	1.6006	444 KB
	10000	10	474.36	476.59	15.0 MB	0.1077	0.1331	1.49 MB	0.2234	0.2506	1.49 MB
		100	1709.5	1710.3	15.2 MB	0.5737	0.6008	1.57 MB	1.5840	1.6158	1.57 MB
		1000	14053	14056	16.5 MB	5.9126	6.0368	2.36 MB	15.920	15.956	2.36 MB
Somoza	21	11	1.3610	1.4202	48 KB	0.0192	0.0200	6.81 KB	0.0007	0.0012	6.81 KB
PhD	73	17	4.6749	4.7559	135 KB	0.0282	0.0296	15.7 KB	0.0028	0.0042	16.3 KB
Cancer	36	14	2.3283	2.4085	75 KB	0.0233	0.0245	9.47 KB	0.0013	0.0021	9.47 KB

Table 3: Micro benchmarks for generation of correlated Beaver triples $([\mathbf{A}], [\mathbf{B}_i], [\mathbf{C}_i] = [\mathbf{AB}_i])$ in the offline phase for $l = 127$ bits. $\mathbf{A}$ has dimension $n \times m$; the $\mathbf{B}_i$ have dimension $m \times 1$. The plaintext modulus used is 2^l. Times (in seconds) and communication cost for correlated triples are amortized for one triple over 1000 iterations. The baseline cost for triple generation via Paillier encryption (with a 3072-bit keysize) is averaged over 5 iterations. All of our code is single threaded and is run in the LAN setting.

setting we do. The most relevant protocols are the ones in SCALE-MAMBA [9] (benchmarked in [10]) and Aliasgari et al. [7]. The protocol from [9] uses fixed-points but focuses primarily on active security; the one from [7] is in the semi-honest setting but uses floating-points, is described only for $n \geq 3$ parties and only for Shamir shares. Consequently, while we provide some comparison points, our comparison is not direct and comes with significant caveats. We intend the comparison to be primarily directional, and to highlight the difference in broader protocol approaches. Specifically, we believe the comparison shows the simplicity of our design in the 2-party semi-honest setting, and the corresponding performance gains (often μs vs ms or s).

First, we compare to the 2-party protocol from [9, 10]. As noted earlier, this protocol targets the active-security setting, while we target the semi-honest setting, so the comparison is not direct. [9] requires a full bit decomposition and uses a polynomial approximation for the fractional part, which incurs a larger error than our approach. For fixed-point exponentiation with $l = 245$ and $l_f = 40$, [10] shows 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 ~2MB per exponentiation. In contrast, for those parameters, our total offline cost is 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 0.055 ms, and our offline time was 0.013 ms. [10] notes that large parameters were chosen specifically for exponentiation (as opposed to $(20, 128)$ for other functions like square-root, sine, cosine etc.), due to high numerical instability. This is not observed in our protocol for the parameters $(20, 127)$, largely due to exponentiating the fractional component in $\mathbb{R}$ rather than using polynomial approximation.

Our protocol also has a large throughput advantage. While [10] reports 76 ops/s when 50 invocations are run in parallel, we achieve ~15,000 ops/s run sequentially for our 127-bit modulus. We also note that the implementation from [10] leverages multiple threads while all our code is single threaded and could potentially be optimized further.

Aliasgari et al [7] provide a secure exponentiation protocol in the semi honest setting. They consider floating-point exponentiation in the 3-party setting with Shamir shares. This is significantly different from our setting, since we target fixed-point exponentiation in the 2-party setting. The comparison therefore comes with significant caveats, but we provide a brief analytical comparison here to highlight the differences in techniques, and therefore efficiency. In particular, the protocol from [7] requires a full bit decomposition, 4 comparison tests, and l_f floating-point multiplications (where l_f is number of significand bits). For a l_f-bit comparable precision (for their best setting where l_f is more than the number of exponent bits k), it requires at least $16 + 12\log l_f + \log\log l_f$ rounds and $\mathcal{O}(k) + \mathcal{O}(l_f \log l_f)$ interactive operations (involving exchange of a secret share) taking preprocessing and parallel computation into account. In comparison, our protocol requires a single round and only one interactive operation (i.e., only one secret share is exchanged) regardless of l and l_f.

7.2 Offline Phase Experiments

We provide micro benchmarks for the offline phase generation of correlated Beaver triples in both the LAN and WAN settings in Table 3. As a baseline, we also compare to the cost when using Paillier encryption. Both our approaches (see Section 5.1) have 5x-10x less communication cost and

are 520x-4200x faster than Paillier AHE-based approaches. Experiments in [19] suggest that an OT-based approach is 20x-30x faster than the Paillier AHE-based one in the LAN setting which highlights that our protocols would also be faster than OT-based triple generation.

7.3 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.

We use the parameters $(l_f, l) = (20, 127)$ here but we also provide results for $(l_f, l) = (15, 63)$ in the full version [16].

Datasets. We run our regression experiments on three datasets (detailed next) from the Princeton University course on Generalized Linear Models [1].

1. *Somoza*. This dataset contains infant and child survival rates in Colombia. Survival is modeled as a function of sex, cohort, and age range. The dataset tracks 2000 infants over several years, and provides aggregate exposures and counts over 21 distinct feature combinations.
2. *Time to PhD.* This dataset predicts PhD graduation as a function of years in graduate school, university, and residence status. We encode the explanatory variables into 17 binary features. Data from 35,000 PhD students is used to calculate the aggregate exposure period and graduation counts for 73 distinct feature combinations.
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 to different age buckets and smoking statuses. There are 36 distinct feature combinations, containing counts and exposure periods from a total of 92,000 respondents.

Accuracy evaluation. To quantify accuracy, we benchmark our secure Poisson regression protocol against a plaintext regression baseline for different learning rates and fixed-point precision. See Figure 3. We observe 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 θ, we find that the actual weights learned by the secure protocol are also nearly exactly the same as those from plaintext learning. See Table 4: the root mean square error between the secure weights and the plaintext weights is very small regardless of the dataset being tested on.

Performance evaluation. We also benchmark the computation (offline and online) and online communication efficiency of our end-to-end protocol in Table 5. In addition to the earlier datasets, we also run our experiments on larger synthetic datasets. 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.

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 4: RMSE 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.

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 still use our exponentiation protocol. For correlated triples, since the gain is only when multiple gradient descent iterations are run, for our timing values, we run 1000 iterations, and report the amortized time for 1 iteration.

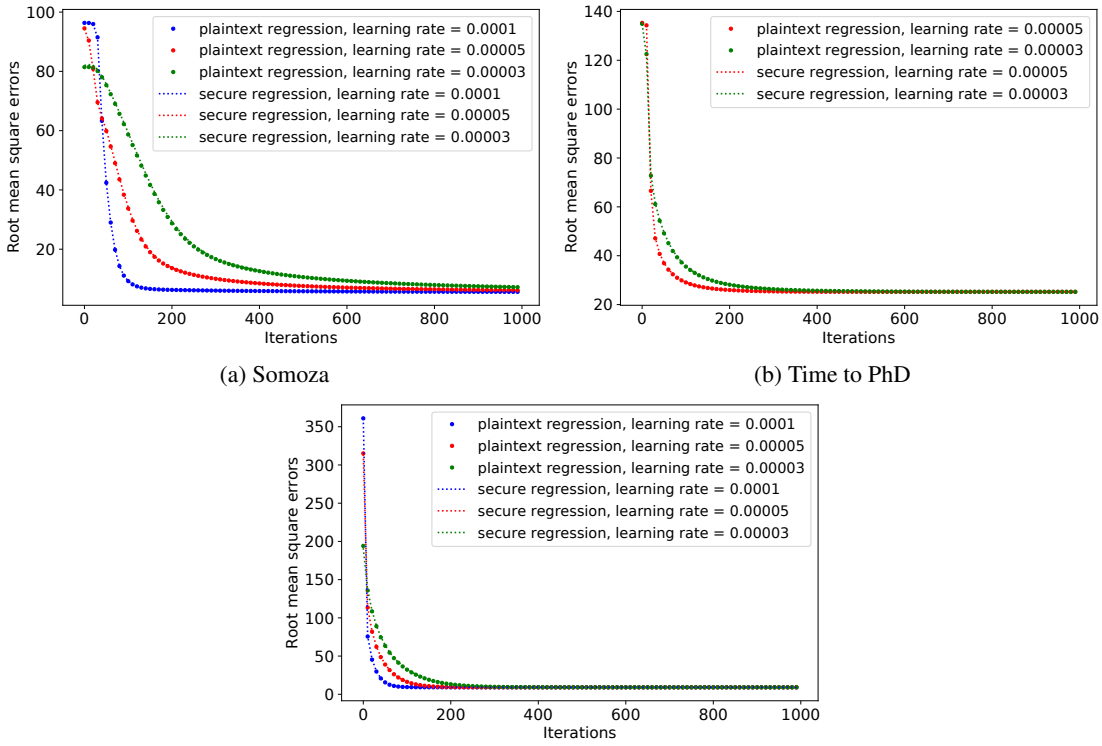
We find that our protocol performs well, even for larger datasets. For example, in the LAN setting, for a dataset with 10,000 elements and 100 features, it has an amortized cost of 3.116 seconds of offline time, 5.501 seconds of online time, and 14.8 MB of communication. Over 100 iterations, the cost is about 5 minutes of offline time, 9 minutes of online time, and 1.48 GB 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.

COVID-19 case fatality rate. Recent work [27] 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 (≥ 60 years), presence of cardiovascular disease, and presence of neurologic diseases. The regression model is used to compute the incidence rate 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 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



(a) Somoza

(b) Time to PhD

(c) Smoking and Cancer

Figure 3: Convergence of the RMSE for plaintext regression versus Secure Poisson regression with 20-bit fixed-point precision.

Dataset	n	m	Standard						Correlated Triples					
			Offline			Online			Offline			Online		
			LAN (s)	WAN (s)	Comm. (MB)	LAN (s)	WAN (s)	Comm. (MB)	LAN (s)	WAN (s)	Comm. (MB)	LAN (s)	WAN (s)	Comm. (MB)
Repl.	100	10	16.700	18.006	0.644	0.008	0.201	0.077	0.028	0.034	0.139	0.006	0.200	0.013
		100	46.740	48.877	0.914	0.065	0.271	0.629	0.044	0.065	0.192	0.062	0.254	0.016
		1000	351.32	354.22	3.61	0.653	1.172	6.149	0.344	0.371	0.725	0.618	0.963	0.049
	1000	10	158.52	168.74	6.17	0.059	0.260	0.763	0.277	0.324	1.33	0.056	0.245	0.123
		100	410.40	421.94	6.44	0.593	1.035	6.259	0.414	0.604	1.39	0.558	0.846	0.131
		1000	2931.8	2944.1	9.14	5.888	8.474	61.22	3.256	3.460	1.92	5.562	6.045	0.214
	10000	10	1582.2	1680.5	61.5	0.584	1.023	7.630	2.650	3.107	13.3	0.549	1.001	1.227
		100	4077.7	4181.3	61.7	5.825	8.342	62.56	3.116	4.997	13.5	5.501	5.989	1.285
		1000	29040	29154	64.4	59.745	72.234	611.9	65.82	70.61	15.1	55.144	55.862	1.862
Sozoma	21	11	4.067	4.421	0.162	0.002	0.199	0.016	0.003	0.008	0.037	0.002	0.197	0.002
PhD	73	17	13.960	15.005	0.500	0.009	0.202	0.081	0.012	0.028	0.116	0.008	0.201	0.004
Cancer	36	14	6.950	7.526	0.263	0.004	0.199	0.034	0.006	0.014	0.060	0.004	0.199	0.003

Table 5: Benchmarks for secure Poisson regression with the parameters $(l_f, l) = (20, 127)$ for different datasets. 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 the "Standard" column, we use standard Beaver triples generated via Paillier encryption (e.g., as in [19]) along with our fixed-point exponentiation protocol. For correlated triples, we use the best performing of our two approaches given the specific (n, m) for offline generation. Both the offline and online phases are amortized over 1000 iterations. All of our code is single threaded.

to each hospital which would individually compute the IRR for each variable, and release the aggregate IRRs.

On a synthetic dataset with similar shape, in the LAN setting, our protocol takes 0.268 seconds in the offline phase and 0.103 seconds in the online phase per iteration of gradient descent, with a total communication cost of 3.52 MB. Assuming 100 iterations of gradient descent are needed for convergence, this results in 26.8 seconds in the offline phase, 10.3 seconds

in the online phase, and a communication of 352 MB.

Predicting credit default rates. [18] 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 [5]. Our secure protocol is a good fit for the latter part, as well as the subsequent inference.

On a synthetic dataset with similar shape, in the LAN setting, our protocol incurs 5.8 seconds of offline time and 3.4 seconds of online time to perform 100 iterations of secure gradient descent, with a total communication cost of 157 MB. Each iteration would incur 0.058 seconds and 0.034 seconds of offline and online time, with 1.57 MB of communication.

Modeling Ad campaign conversion rates. Google researchers [26] 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. [26] 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 [5]. Our work is well-suited for regression as well as the subsequent inference.

On a synthetic dataset with 5 ad channels and 3 binary attributes per channel for a total of 15 binary attributes, and assuming 100,000 training points, our regression takes 6.90 seconds of offline time and 8.197 seconds of online time per iteration of gradient descent, with 156.7 MB of total communication. For 100 iterations of gradient descent, we incur 11.5 minutes of offline time and about 14 minutes of online time, with 15.67 GB of total 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.

Acknowledgments

We thank Tancrède Lepoint for insightful discussions. We also thank the anonymous reviewers of USENIX Security for their helpful comments and suggestions.

References

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

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

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

[4] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J. Kusner, and Adrià Gascón. Quotient: Two-party secure neural network training and prediction. In *CCS*, pages 1231–1247, 2019.

[5] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *SIGMOD*, pages 86–97, 2003.

[6] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[7] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS*, 2013.

[8] Abdelrahaman Aly, Aysajan Abidin, and Svetla Nikova. Practically efficient secure distributed exponentiation without bit-decomposition. In *FC*, pages 291–309, 2018.

[9] Abdelrahaman Aly, K Cong, D Cozzo, M Keller, E Orsini, D Rotaru, O Scherer, P Scholl, NP Smart, T Tanguy, and T Wood. SCALE-MAMBA v1.10: Documentation, 2020. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf.

[10] Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In *ACNS*, pages 509–529, 2019.

[11] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pages 420–432, 1992.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3), 2014.

[13] E. L. Frome. The analysis of rates using Poisson regression models. *Biometrics*, 39(3):665–674, 1983.

[14] Hossein Ghodosi, Josef Pieprzyk, and Ron Steinfeld. Multi-party computation with conversion of secret sharing. *Des. Codes Cryptogr.*, 62(3):259–272, 2012.

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

[16] Mahimna Kelkar, Phi Hung Le, Mariana Raykova, and Karn Seth. Secure Poisson regression. Cryptology ePrint Archive, Report 2021/208, 2021. https://ia.cr/2021/208.

[17] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, pages 1–23, 2010.

[18] Sami Mestiri and Abdeljelil Farhat. Using non-parametric count model for credit scoring. *SSRN Electronic Journal*, 10 2019.

[19] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE SP*, pages 19–38, 2017.

[20] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, pages 681–700, 2012.

[21] Chao Ning and Qiuliang Xu. Constant-rounds, linear multi-party computation for exponentiation and modulo reduction with perfect security. In *ASIACRYPT*, pages 572–589, 2011.

[22] Emmanuela Orsini, Nigel P. Smart, and Frederik Vercauteren. Overdrive2k: Efficient secure MPC over $\mathbb{Z}_{2^k}$ from somewhat homomorphic encryption. In *CT-RSA*, pages 254–283, 2020.

[23] Deevashwer Rathee, Thomas Schneider, and K. K. Shukla. Improved multiplication triple generation over rings via RLWE-based AHE. In *CANS*, pages 347–359, 2019.

[24] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*, pages 1501–1518, 2019.

[25] Sean Richey. Who votes alone? the impact of voting by mail on political discussion. *Australian Journal of Political Science*, 40(3):435–442, 2005.

[26] Dinah Shender, Ali Nasiri Amini, Xinlong Bao, Mert Dikmen, Amy Richardson, and Jing Wang. A time to event framework for multi-touch attribution. arXiv 2009.08432, 2020.

[27] G. J. B. Sousa, T. S. Garces, V. R. F. Cestari, R. S. Florêncio, T. M. M. Moreira, and M. L. D. Pereira. Mortality and survival of COVID-19. *Epidemiology and Infection*, 148, 2020.

[28] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.

[29] Ching-Hua Yu, Sherman S. M. Chow, Kai-Min Chung, and Feng-Hao Liu. Efficient secure two-party exponentiation. In *CT-RSA*, pages 17–32, 2011.

A Detailed secure functionalities

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

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 $\underset{\sim}{x}/c$ in $\mathcal{R}$, with an error at most 2^{-l_f}. Note that while our protocol does not require it, a public negative divisor can easily be handled by first dividing by the absolute value and then subtracting the shares from the modulus N.

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 $0 \le d_i < 1$ 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 $\widehat{(c)(x)}$. 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, $(cx)/2^{l_f} - 1 \le [\![z]\!]_0 + [\![z]\!]_1 \bmod N \le (cx)/2^{l_f} + 1$.

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, $(N - \frac{c(N-x)}{2^{l_f}}) - 1 \le [\![z]\!]_0 + [\![z]\!]_1 \bmod N \le (N - \frac{c(N-x)}{2^{l_f}}) + 1$.

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 $(x)(c)$, with an error of at most 2^{-l_f}. Note that a negative c can also be handled analogously to PubFPDiv.

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.

B Background on Ring-LWE

Ring-Learning-With-Errors [17] (RLWE) is a hardness assumption based on which efficient homomorphic encryption schemes have been constructed. We use the leveled encryption scheme proposed by Brakersky et al. [12], 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.

We now describe the operations for Ring-LWE-based encryption in the two-party setting.

Key Generation. 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 $b = as + t\varepsilon \in R_q$ 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 $c_0 = m + bv + p\varepsilon_0$ and $c_1 = av + p\varepsilon_1$ (where $v, \varepsilon_0, \varepsilon_1$ 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 $\text{Dec}_{sk}(c) = c_0 + c_1 s \mod p$.

Plaintext addition. Given a ciphertext $c \equiv (c_0, c_1) = \text{Enc}(m) \in R_q^2$ and a plaintext message $m' \in R_p$, one can produce the encryption of $(m + m')$ as $c' = (c_0 + m', c_1) = \text{Enc}(m + m')$.

Scaling. Given a ciphertext $c \equiv (c_0, c_1) = \text{Enc}(m) \in R_q^2$ and a scalar $a \in \mathbb{Z}_p$, one can produce the encryption of $am = (am_1, \cdots, am_N)$ as $c' = (a \cdot c_0, a \cdot c_1) = \text{Enc}(a \cdot m)$.

Shifting. Given a ciphertext $c \equiv (c_0, c_1) = \text{Enc}(m) \in R_q^2$ where $m = (m_1, \cdots, m_K, 0, \cdots, 0)$, we can produce the encryption of $m' = (0, \cdots, 0, m_1, \cdots, m_K, 0, \cdots, 0)$ where m_1 is shifted by a distance t and $t + K \le N$. Let $v = (0, \cdots, 0, 1, 0, \cdots, 0)$ where v is zero everywhere except for the t^{th} position. Then $c' = (v \circ c_0, v \circ c_1) = \text{Enc}(m \circ v) = \text{Enc}(m')$ where $\circ$ denotes polynomial multiplication operation in $R_q = \mathbb{Z}_q[X]/\Phi_N(X)$.

Choosing parameters for Ring-LWE. Following the parameters suggested by [6], we use a ciphertext prime $q = 160$ bits for our RLWE scheme when generating Beaver triples for ring of size $p = 63$ bits ($\mathbb{Z}_{2^{63}}$) and $q = 295$ bits when generating triples for ring of size $p = 127$ bits ($\mathbb{Z}_{2^{127}}$). In the first case, we use a polynomial modulus of degree $2^{12} = 4096$, while for the later case $2^{14} = 16384$. This is sufficient for security of at least 128 bits. To allow efficient encryption and decryption via the use of number theoretic transform, we choose a ciphertext modulus q such that $q \equiv 1 \bmod 2N$.