Practical, Private Assurance of the Value of Collaboration via Fully Homomorphic Encryption

Hassan Jameel Asghar
Macquarie University
Australia
hassan.asghar@mq.edu.au

Zhigang Lu
Western Sydney University
Australia
z.lu@westernsydney.edu.au

Zhongrui Zhao
James Cook University
Australia
zhongrui.zhao@my.jcu.edu.au

Dali Kaafar
Macquarie University
Australia
dali.kaafar@mq.edu.au

Abstract

Two parties wish to collaborate on their datasets. However, before they reveal their datasets to each other, the parties want to have the guarantee that the collaboration would be fruitful. We look at this problem from the point of view of machine learning, where one party is promised an improvement on its prediction model by incorporating data from the other party. The parties would only wish to collaborate further if the updated model shows an improvement in accuracy. Before this is ascertained, the two parties would not want to disclose their models and datasets. In this work, we construct an interactive protocol for this problem based on the fully homomorphic encryption scheme over the Torus (TFHE) and label differential privacy, where the underlying machine learning model is a neural network. Label differential privacy is used to ensure that computations are not done entirely in the encrypted domain, which is a significant bottleneck for neural network training according to the current state-of-the-art FHE implementations. We formally prove the security of our scheme assuming honest-but-curious parties, but where one party may not have any expertise in labelling its initial dataset. Experiments show that we can obtain the output, i.e., the accuracy of the updated model, with time many orders of magnitude faster than a protocol using entirely FHE operations.

Keywords

privacy-preserving machine learning, fully homomorphic encryption, label differential privacy

1 Introduction

Data collaboration, i.e., joining multiple datasets held by different parties, can be mutually beneficial to all parties involved as the joint dataset is likely to be more representative of the population than its constituents. In the real world, parties have little to no knowledge of each others' datasets before collaboration. Arguably, the parties would only collaborate if they had some level of trust in the quality of data held by other parties. When the parties involved are reputable organizations, one may assume their datasets to be of high quality. However, in many cases, little may be known about them. In such cases, each party would like some sort of *assurance* that their collaboration will indeed be beneficial.

Let us elaborate this scenario with an example. Assume companies P_1 and P_2 are in the business of developing antivirus products. Each company holds a dataset of malware programs labelled as a particular type of malware, e.g., ransomware, spyware and trojan. This labelling is done by a team of human experts employed by the company. Since manual labelling is expensive, P_1 uses a machine learning model to label new malware programs. The performance of this model can be tested against a smaller *holdout* dataset of the latest malware programs labelled by the same experts. However, due to a number of reasons such as the under-representation of some of the malware classes in the training dataset or *concept drift* [31] between the training dataset and the holdout dataset, the performance of this model on the holdout dataset begs improvement. Company P_2 offers a solution: by combining P_2's dataset with P_1's, the resulting dataset would be more representative and hence would improve the accuracy of P_1's model. Before going into the laborious process of a formal collaborative agreement with P_2, P_1 would like to know whether this claim will indeed be true. On the other hand, P_2 would not want to hand over its dataset, in particular, its labels to P_1 before the formal agreement.

Many similar examples to the one outlined above may occur in real-world data collaboration scenarios. A rather straightforward solution to this problem can be obtained using fully homomorphic encryption (FHE). Party P_2 encrypts the labels of its dataset using the encryption function of the FHE scheme,[1] and sends its dataset with encrypted labels to P_1. Party P_1, then combines this dataset with its own, trains the model and tests its accuracy against the holdout dataset all using FHE operations. The final output is then decrypted by party P_2.[2] However, this involves training the model entirely in the encrypted domain, which even with the state-of-the-art homomorphic encryption schemes, is computationally expensive [16, 21, 26, 32, 33].

In this paper, we propose an efficient solution to this problem. Our main contributions are as follows

---

[1]For reasons discussed in Section 2.2, we only consider the case when the labels are encrypted, and not the features themselves.
[2]We are assuming that decryption is faithful.

- We combine fully homomorphic encryption over the torus [9] with label differential privacy [8] to provide an interactive solution to the problem where parties P_1 and P_2 would like to train a model on their combined datasets to assess its accuracy without learning the resulting model and each other's data labels. Our main idea is that since the features are known in the clear, the first forward pass in the backpropagation algorithm of neural networks can be performed in the clear, up to the point where we utilize the (encrypted) labels from party P_2's dataset. If further computation is done homomorphically, then we would endure the same computational performance bottleneck as previous work. We, therefore, add (label) differentially private noise to the gradients and decrypt them before the backward pass. This ensures that most steps in the neural network training are done in cleartext, albeit with differentially private noise, giving us computational performance improvements over an end-to-end FHE solution.

- We use label differential privacy in a novel way. More specifically, P_2 can use a specific value of the differentially privacy parameter ϵ such that the accuracy of the model on the joint datasets lies between the accuracy of P_1's model and the accuracy achievable on the joint model if no differential privacy were to be applied. This provides assurance to P_1 that the combined dataset will improve accuracy promising further improvement if the parties combine their datasets in the clear via the formal collaborative agreement. Thus, in our protocol, we need not worry about making the differentially private model as accurate as the final model, which is in general a difficult task in differential privacy literature.

- We initiate the study to determine whether it is possible to improve model accuracy by joining two datasets where the labels from the second dataset are not labelled with any domain knowledge. As we show, there are cases in which the combined dataset may show improvement over the standalone dataset even without domain knowledge. This problem has broader applications than our work. Our treatment is limited to binary classification and the 0-1 loss function. We leave full exploration of this topic as future work.

- We provide an implementation of our protocol using Zama's `Concrete` library and evaluate our protocol over multiple datasets. Our results show that our protocol can be implemented faster than an end-to-end FHE solution.

2 Preliminaries and Threat Model

2.1 Notation

We follow the notations introduced in [27]. The datasets come from the joint domain: $\mathbb{D} = \mathcal{X} \times \mathcal{Y}$, where $\mathcal{X}$ denotes the domain of features, and $\mathcal{Y}$ the domain of labels. A dataset D is a multiset of elements drawn i.i.d. under the joint distribution $\mathcal{D}$ over domain points and labels. We denote by $\mathcal{D}_x$, the marginal distribution of unlabelled domain points. In some cases, a dataset may be constructed by drawing unlabelled domain points under $\mathcal{D}_x$, and then labelled according to some labelling function, which may not follow the marginal distribution of labels under $\mathcal{D}$, denoted $\mathcal{D}_{y|x}$. In this case, we shall say that the dataset is labelled by the

labelling function to distinguish it from typical datasets. Let $\mathcal{A}$ denote the learning algorithm, e.g., a neural network training algorithm. We denote by $M \leftarrow \mathcal{A}(D)$ the model M returned by the learning algorithm on dataset D. The notation $(x, y) \sim \mathcal{D}$ means that the sample (x, y) is sampled from the distribution $\mathcal{D}$, and the notation $(x, y) \sim D$, where D is a dataset means that the sample (x, y) is sampled uniformly at random from D. Given the model M, and a sample $(x, y) \sim \mathcal{D}$, we define a generic loss function $\ell(M, x, y)$, which outputs a non-negative real number. For instance, $\ell(M, x, y)$ can be the 0-1 loss function, defined as: $\ell(M, x, y) = 1$ if $M(x) \neq y$, and 0 otherwise. We define the true error of M as $L_{\mathcal{D}}(M) = \mathbb{E}_{(x,y) \sim \mathcal{D}}[\ell(M, x, y)]$. Notice that for the 0-1 loss function, this means that $L_{\mathcal{D}}(M) = \Pr_{(x,y) \sim \mathcal{D}}[M(x) \neq y]$. The empirical error of the model M over the dataset D having m elements $(x_1, y_1), \ldots, (x_m, y_m)$ is defined as:

$$L_D(M) = \frac{1}{m} \sum_{i \in [m]} \ell(M, x_i, y_i) \tag{1}$$

Note that for the $0-1$ loss function $L_D(M) = \Pr_{(x,y) \sim D}[M(x) \neq y]$.

2.2 The Setting

The Scenario. We consider two parties P_1 and P_2. For $i \in \{1, 2\}$, party P_i's dataset is denoted D_i. Each D_i contains points of the form $(\mathbf{x}, y)$. The parties wish to collaborate on their datasets D_i. The features $\mathbf{x}$ are shared in the open; whereas the labels y for each $\mathbf{x}$ in D_i are to be kept secret from the other party. This scenario holds in applications where *gathering data (*$\mathbf{x}$*) may be easy, but labelling is expensive.* For example, malware datasets (binaries of malware programs) are generally available to antivirus vendors, and often times features are extracted from these binaries using publicly known feature extraction techniques, such as the LIEF project [30]. However, labelling them with appropriate labels requires considerable work from (human) experts. Other examples include sentiment analysis on public social media posts, and situations where demographic information is public (e.g., census data) but income predictability is private [8]. Lastly, we want to acknowledge that our focus is on supervised learning. There are works that show unsupervised learning techniques may improve the model [35] even if the labels are important in many applications such as malware detection. Our focus is on the added value of the labels.

The Model. Before the two parties reveal their datasets to each other, the parties want to have the guarantee that the collaboration would be *valuable*. We shall assume that party P_1 already has a model M_1 trained on data D_1. P_1 also has a labelled holdout data D_{hold} against which P_1 tests the accuracy of M_1. The goal of the interaction is to obtain a new model M_2 trained on $D_1 \cup D_2$. M_2's accuracy again is tested against D_{hold}. The collaboration is defined to be *valuable* for P_1 if the accuracy of M_2 is higher than the accuracy of M_1 against D_{hold}. In this paper, we only study *value* from P_1's perspective. We will consider a neural network trained via stochastic gradient descent as our canonical model.

The Holdout Dataset. As mentioned above, P_1 has a holdout

dataset D_{hold} against which the accuracy of the models is evaluated. This is kept separate from the usual training-testing split of the dataset D_1. It makes sense to keep the same holdout dataset to check how the model trained on the augmented/collaborated data performs. For instance, in many machine learning competitions teams compete by training their machine learning models on a publicly available training dataset, but the final ranking of the models, known as *private leaderboard*, is done on a hidden test dataset [4]. This ensures that the models are not overfitted by using the test dataset as feedback for re-training. We assume that D_{hold} is continually updated by adding new samples (e.g., malware never seen by P_1) labelled by human experts, and is more representative of the population than D_1. For instance, the holdout dataset reflects the *concept drift* [31] better than D_1. If D_2 happens to have the same *concept drift* as D_{hold}, then M_2 (trained on $D_1 \cup D_2$) would have better test accuracy than M_1 against D_{hold}. Alternatively, the holdout dataset could be more balanced than D_1, e.g., if D_1 has labels heavily skewed towards one class. Again, in this case, if D_2 is more balanced than D_1, then M_2 will show better accuracy. We argue that it is easy for P_1 to update D_{hold} than D_1 as the latter requires more resources due to the difference in size.

2.3 Privacy

Privacy Expectations. We target the following privacy properties:

- Datasets D_1 and D_{hold}, and model M_1 should be hidden from P_2.
- The labels of dataset D_2 should be hidden from P_1.
- Neither P_1 nor P_2 should learn M_2, i.e., the model trained on $D_1 \cup D_2$.
- Both parties should learn whether $L_{hold}(M_2) < L_{hold}(M_1)$, where L_{hold} is the loss evaluated on D_{hold} (see Eq. (1)).

Threat Model. We assume that the parties involved, P_1 and P_2, are honest-but-curious. This is a reasonable assumption since once collaboration is agreed upon, the model trained on clear data should be able to reproduce any tests to assess the quality of data pre-agreement. Why then would P_1 not trust the labelling from P_2? This could be due to the low quality of P_2's labels, for many reasons. For example, P_2's expertise could in reality be below par. In this case, even though the labelling is done honestly, it may not be of sufficient quality. Furthermore, P_2 can in fact lie about faithfully doing its labelling without the fear of being caught. This is due to the fact that technically there is no means available to P_1 to assess how P_2's labels were produced. All P_2 needs to do is to provide the same labels before and after the collaborative agreement. As long as labelling is consistent, there is no fear of being caught.

2.4 Background

Feedforward Neural Networks. A fully connected feedforward neural network is modelled as a graph with a set of vertices (neurons) organised into layers and weighted edges connecting vertices in adjacent layers. The sets of vertices from each layer form a disjoint set. There are at least three layers in a neural network, one input layer, one or more hidden layers and one output layer. The

number of neurons in the input layer equals the number of features (dimensions). The number of neurons in the output layer is equal to the number of classes K. The vector of weights $\mathbf{w}$ of all the weights of the edges constitutes the parameters of the network, to be learnt during training. We let R denote the number of weights in the last layer, i.e., the number of edges connecting to the neurons in the output layer. For a more detailed description of neural networks, see [27].

Backward Propagation and Loss. Training a deep neural network involves multiple iterations/epochs of forward and backward propagation. The input to each neuron is the weighted sum of the outputs of all neurons connected to it (from the previous layer), where the weights are associated with the edge connecting to the current neuron. The output of the neuron is the application of the activation function on this input. In this paper, we assume the activation function to be the sigmoid function. Let $\mathbf{z} = (z_1, z_2, \ldots, z_K)$ denote the output of the last layer of the neural network. We assume there to be a softmax layer, immediately succeeding this which outputs the probability vector $\mathbf{p}$, whose individual components are given as $p_i = e^{z_i}/(\sum_j e^{z_j})$. Clearly, the sum of these probabilities is 1. Given the one-hot encoded label $\mathbf{y}$, one can compute the loss as $L(\mathbf{w}) = \sum_{i=1}^{K} \ell(p_i, y_i)$, where $\ell(\cdot, \cdot)$ is the loss function. In this paper, we shall consider it to be the cross-entropy loss given by: $\ell(p_i, y_i) = -\sum_{i=1}^{K} y_i \ln p_i$. Given $L(\mathbf{w})$, we can calculate its gradient as:

$$\nabla L(\mathbf{w}) = \frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \mathbf{p}} \frac{\partial \mathbf{p}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{w}}$$

This chain rule can be used in the backpropagation algorithm to calculate the gradients to minimise the loss function via stochastic gradient descent (SGD). The calculated gradients will be then used to update the weights associated with each edge for the forward propagation in the next epoch. For more details of the backpropagation-based SGD algorithm, see [27].

Label Differential Privacy. Since only the labels of the dataset D_2 are needed to be private, we shall use the notion of label differential privacy to protect them. Ordinary differential privacy [13] defines neighbouring datasets as a pair of datasets which differ in one row. Label differential privacy considers two datasets of the same length as being neighbours if they differ only in the *label* of exactly one row [8]. This is a suitable definition of privacy for many machine learning applications such as malware labels (as already discussed) and datasets where demographic information is already public but some sensitive feature (e.g., income) needs to be protected [8]. Furthermore, for tighter privacy budget analysis we use the notion of f-differential privacy [12], modified for label privacy since it allows straightforward composition of the Gaussian mechanism as opposed to the normal definition of differential privacy which needs to invoke its approximate variant to handle this mechanism. More formally, two datasets D_1 and D_2 are said to be neighbouring datasets, if they are of the same size and differ only in at most one label. Due to lack of space, we present the formal definitions in Appendix B.

Fully Homomorphic Encryption over the Torus (TFHE). Let $\mathbb{T}$ denote the torus, the set of real numbers modulo 1, i.e., the set $[0, 1)$.

The torus defines an Abelian group where two elements can be added modulo 1. The internal product is not defined. However, one can multiply an integer with a torus element by simply multiplying them in a usual way and reducing modulo 1. For a positive integer q, the discretised torus $\mathbb{T}_q$ is defined as the set $\{0, \frac{1}{q}, \ldots, \frac{q-1}{q}\}$. Clearly $\mathbb{T}_q \subset \mathbb{T}$. Given elements $\frac{a}{q}, \frac{b}{q} \in \mathbb{T}_q$, their sum is $\frac{c}{q} \in \mathbb{T}_q$, where $c \equiv a + b \pmod{q}$. Given an integer $z \in \mathbb{Z}$ and a torus element $\frac{a}{q} \in \mathbb{T}_q$, we define their product $z \cdot \frac{a}{q}$ as $\frac{b}{q}$ where $b \equiv za \pmod{q}$. The plaintext space is a subgroup of $\mathbb{T}_q$, defined as $\mathcal{P} = \{0, \frac{1}{p}, \ldots, \frac{p-1}{p}\}$, for some $p \geq 2$ such that p divides q.

Let χ be the normal distribution $\mathcal{N}(0, \sigma^2)$ over $\mathbb{R}$. Let $e_0 \leftarrow \chi$. The noise error $e \in \hat{\chi}$ is defined as $e = \frac{\lfloor qe_0 \rceil}{q}$. Let N be a positive integer. Let $\mathbf{s} = (s_1, \ldots, s_N)$ be a binary vector chosen uniformly at random (the private key). Given a message $x \in \mathcal{P}$, the TLWE encryption of x under $\mathbf{s}$ is defined as $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{T}_q^{N+1}$, where $\mathbf{a}$ is a vector of N-elements drawn uniformly at random from $\mathbb{T}_q$, and

$$b = \langle \mathbf{s}, \mathbf{a} \rangle + x + e.$$

To decrypt $\mathbf{c}$ one computes: $x^* = b - \langle \mathbf{s}, \mathbf{a} \rangle$, and returns the nearest element in $\mathcal{P}$ to x^*. The scheme is secure under the learning with errors (LWE) problem over the discretized torus [9, 17]:

Definition 1 (TLWE Assumption). *Let q and $N \in \mathbb{N}$. Let $\mathbf{s} = (s_1, \ldots, s_N)$ be a binary vector chosen uniformly at random. Let $\hat{\chi}$ be an error distribution defined above. The learning with errors over the discretized torus (TLWE) problem is to distinguish samples chosen according to the following distributions:*

$$\mathcal{D}_0 = \{(\mathbf{a}, b) \mid \mathbf{a} \leftarrow \mathbb{T}_q^N, b \leftarrow \mathbb{T}_q\},$$

and

$$\mathcal{D}_1 = \{(\mathbf{a}, b) \mid \mathbf{a} \leftarrow \mathbb{T}_q^N, b = \langle \mathbf{s}, \mathbf{a} \rangle + e, e \leftarrow \hat{\chi}\},$$

where except for e which is sample according to the distribution $\hat{\chi}$, the rest are sampled uniformly at random from the respective sets.

Integer Encoding. Before encrypting, we will encode the input as an integer. This is a requirement in the current version of Zama's `Concrete` TFHE library [36], which we use for our implementation. Let $\mathbf{x} \in \mathbb{R}^m$. For a precision level r, where r is a positive integer, we will encode $\mathbf{x}$ as $\lfloor r\mathbf{x} \rceil = (\lfloor rx_1 \rceil, \ldots, \lfloor rx_m \rceil)$, before encrypting. Decoding is done, after decryption, by dividing the encoded vector by r. For instance if $r = 10$, then the real number 3.456 is encoded as $\lfloor 34.56 \rceil = 34$. Decoding it yields 3.4.

Parameters. We use the default parameters for TLWE encryption available through the `Concrete` library. Under the default setting, we have $N = 630$, giving us a security level of 128 bits. The parameter q is set to 64 bits, meaning a torus element can be represented by 64 bits [17]. Since the plaintext parameter p divides q, we can assume p to be less than 64 bits. The noise parameter σ is 2^{-15} [17].

3 Lack of Domain Knowledge

Before we give a privacy-preserving solution to our problem, we want to investigate whether the problem has a solution in the clear domain. More precisely, we seek to find conditions when party P_2, lacking domain knowledge, cannot come up with a dataset

D_2 such that $L_{\text{hold}}(M_2) < L_{\text{hold}}(M_1)$. Here $M_2 \leftarrow \mathcal{A}(D_1 \cup D_2)$ and $M_1 \leftarrow \mathcal{A}(D_1)$, and L_{hold} is as defined in Eq. (1). Although at first glance the problem appears to be simple, in reality it is quite involved as multiple scenarios need to be considered. We therefore only assume binary classification, with the results for the multiclass setting left for future work. We assume the 0-1 loss function. How do we define lack of domain knowledge? Since the feature vectors are public, P_2 can easily obtain a set of raw inputs to obtain the feature vectors in D_2. Thus, domain knowledge should be captured in the labels to the feature vectors in D_2. We define lack of domain knowledge as using a labeling function g, potentially probabilistic, which labels any domain point x independent of the distribution of its true label. That is given any $x \in \mathcal{X}$:

$$\Pr[g(x) = y' \mid \mathcal{D}_{y|x}] = \Pr[g(x) = y'], \tag{2}$$

for all $y' \in \mathcal{Y}$. Note that this does not mean that the labeling is necessarily incorrect. We call g as defined in Eq. 2 as an *oblivious labeling function*. We shall first show that if D is labelled by an oblivious labeling function g, then M as returned by a learning algorithm $\mathcal{A}$ (taking D as the input) will have:

$$\mathbb{E}_g[L_{\text{hold}}(M)] = 1/2,$$

provided that D_{hold} is a *balanced dataset*, where the subscript means that expectation is taken over the random choices of g. Having established the necessity of D_{hold} to be balanced, we shall then show that

$$\mathbb{E}_g[L_{\text{hold}}(M_2)] \geq L_{\text{hold}}(M_1),$$

where $M_2 \leftarrow \mathcal{A}(D_1 \cup D_2)$, and $M_1 \leftarrow \mathcal{A}(D_1)$, provided a certain condition is met. Details follow.

Theorem 1. *Let D_{hold} be a balanced dataset. That is $\Pr[y = 1 \mid (x, y) \sim D_{hold}] = \frac{1}{2}$, where $\sim$ denotes uniform random sampling. Let D be a dataset labelled by an oblivious labeling function g. Let $\mathcal{A}$ be a learning algorithm. Let $M \leftarrow \mathcal{A}(D)$. Then,*

$$\mathbb{E}_g[L_{hold}(M)] = 1/2,$$

where the expectation is over the random choices of g.

Proof. See Appendix A □

What happens if D_{hold} is not balanced? Then, we may get a loss less than 1/2. To see this, assume that $\Pr[y = 1 \mid (x, y) \sim D_{\text{hold}}] = q > \frac{1}{2}$. Consider the oblivious labelling function g which outputs 1 with probability $q = 1$, i.e., the constant function $g(x) = 1$. Then if M is the Bayes optimal classifier [27, §3.2.1] for g, then for $(x, y) \sim D_{\text{hold}}$, we have:

$$\Pr[M(x) \neq y] = \Pr[M(x) = 0 \mid y = 1] \Pr[y = 1]$$
$$+ \Pr[M(x) = 1 \mid y = 0] \Pr[y = 0]$$
$$= \Pr[M(x) = 1 \mid y = 0] \Pr[y = 0]$$
$$= \Pr[y = 0] = 1 - q < \frac{1}{2}.$$

Thus, it is crucial to test the model over a balanced dataset.

Theorem 2. *Let D_1 be a dataset. Let D_2 be a dataset labelled according to an oblivious labeling function g. Let D_{hold} be a balanced*
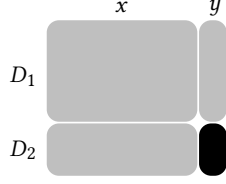
Figure 1: A batch will contain some points from D_1 and some from D_2. Only the labels from the points in D_2 needs to be kept private (shaded black) from P_1.

dataset. Let $\mathcal{A}$ be a learning algorithm. Let $M_1 \leftarrow \mathcal{A}(D_1)$ and $M_2 \leftarrow \mathcal{A}(D_1 \cup D_2)$. If :

$$\Pr_{g,(x,y)\sim D_{hold}}[M_2(x) = 0 \mid y = 1] + \Pr_{g,(x,y)\sim D_{hold}}[M_2(x) = 1 \mid y = 0]$$
$$\geq \Pr_{(x,y)\sim D_{hold}}[M_1(x) = 0 \mid y = 1] + \Pr_{(x,y)\sim D_{hold}}[M_1(x) = 1 \mid y = 0],$$
$$(3)$$

then

$$\mathbb{E}_g[L_{hold}(M_2)] \geq L_{hold}(M_1),$$

where the expectation is taken only over random choices of g, and $(x,y) \sim D_{hold}$ means the sample is chosen uniformly at random from D_{hold}.

PROOF. See Appendix A. □

Concluding, we have:

THEOREM 3. *Let D_1 be a dataset. Let D_2 be a dataset labelled according to an oblivious labeling function g. Let D_{hold} be a balanced dataset of size m. Let $\mathcal{A}$ be a learning algorithm. Let $M_1 \leftarrow \mathcal{A}(D_1)$ and $M_2 \leftarrow \mathcal{A}(D_1 \cup D_2)$. Under condition (3) of Theorem 2, for any $\delta > 0$*

$$\Pr_g[L_{hold}(M_1) - L_{hold}(M_2) \geq \delta] \leq \exp(-2m\delta^2),$$

where the probability is taken over the random choices of g.

PROOF. See Appendix A. □

Exactly when does condition (3) in Theorems 2 and 3 hold? We postulate and give some experimental evidence that the condition holds only when D_1 follows the distribution $\mathcal{D}$. On the other hand, if the labels of D_1 are out of distribution, i.e., one class is more under-represented compared to the true distribution, the above condition does not hold. Due to lack of space, details appear in Appendix A. We acknowledge that we have only touched the tip of the iceberg when it comes to the full exploration of this problem for both binary and multi-class setting and leave it as an open problem.

4 Our Solution

4.1 Intuition

Consider the training of the neural network on $D_1 \cup D_2$, with weights $\mathbf{w}$. Using the stochastic gradient descent (SGD) algorithm, one samples a batch B, from which we calculate per sample loss

$L_s(\mathbf{w})$, where $s = (\mathbf{x}, \mathbf{y}) \in B$. Given this, we can compute the average loss over the batch via:

$$L_B(\mathbf{w}) = \frac{1}{|B|} \sum_{s \in B} L_s(\mathbf{w}) = \frac{1}{|B|} \sum_{\substack{s \in B \\ s \in D_1}} L_s(\mathbf{w}) + \frac{1}{|B|} \sum_{\substack{s \in B \\ s \in D_2}} L_s(\mathbf{w}) \quad (4)$$

As shown in Figure 1, everything in this computation is known to P_1, except for the labels in D_2. Thus, P_1 can compute the gradients for samples in D_1, but to update the weights, P_1 needs the gradients for samples from D_2. From Eq. (4), we are interested in computing the loss through the samples in a batch B that belong to the dataset D_2. Overloading notation, we still use B to denote the samples belonging to D_2. The algorithm to minimize the loss is the stochastic gradient descent algorithm using backpropagation. This inolves calculating the gradient $\nabla L_B(\mathbf{w})$. As noted in [34], if we are using the backpropagation algorithm, we only need to be concerned about the gradients corresponding to the last layer. Again, to simplify notation, we denote the vector of weights in the last layer by $\mathbf{w}$.

In Appendix D, we show that the gradient of the loss can be computed as:

$$\nabla L_B(\mathbf{w}) = \frac{1}{|B|} \sum_{s \in B} \sum_{i=1}^{K} p_i(s) \frac{\partial z_i(s)}{\partial \mathbf{w}} - \frac{1}{|B|} \sum_{s \in B} \sum_{i=1}^{K} y_i(s) \frac{\partial z_i(s)}{\partial \mathbf{w}}, \quad (5)$$

where $y_i(s)$ is the ith label of the sample s, $p_i(s)$ is the probability of the ith label of sample s, $z_i(s)$ is the ith input to the softmax function for the sample s, and K denotes the number of classes. The LHS term of this equation can be computed by P_1 as this is in the clear. However, the RHS term requires access to the labels.

If we encrypt the labels, the gradients calculated in Eq. (5) will be encrypted, using the homomorphic property of the encryption scheme. This means that the gradient of the batch, as well as the weight updates will be encrypted. Thus, the entire training process after the first forward pass of the first epoch will be in the encrypted domain. While this presents one solution to our problem, i.e., obtaining an encrypted trained model, which could then be decrypted once the two parties wish to collaborate, existing line of works [16, 21, 26, 32, 33] show that neural network training entirely in the encrypted domain is highly inefficient. For instance, a single mini-batch of 60 samples can take anywhere from more than 30 seconds to several days with dedicated memory ranging from 16GB to 250GB using functional encryption or homomorphic encryption [32]. In some of our neural network implementations, we use a batch size of 128 with 100 epochs, which means that the time consumed for an end-to-end training entirely in the encrypted domain would be prohibitive. Our idea is to take advantage of the fact that the feature vectors are in the clear, and hence it may be possible to decrypt the labels in each batch so that backpropagation can be carried out in cleartext, giving us computational advantage over an all encrypted solution. This is where we employ label differential privacy. A straightforward way to accomplish this is to let P_2 add differentially private noise to all its labels and simply handover its noisy dataset to P_1, playing no further part (except for receiving the accuracy result). However, this is less desirable from the utility point of view as we argue in detail in Section 6.4. Instead, we add noise to the gradients in each batch with P_2 interactively adding noise to the average gradient computed in each epoch, similar to what is done in [34].

4.2   Proposed Protocol

Our solution is as follows. Figure 2 shows the higher level overview of our protocol.

(1) To start, P_2 sends only the encrypted form of its dataset D_2 where each sample is $(\mathbf{x}, [\![\mathbf{y}]\!]_k)$, where k is P_2's encryption key of a homomorphic encryption scheme. In particular, $[\![\mathbf{y}]\!]_k$ is a vector of K elements, each element of which is encrypted under k.

(2) For each sample $s \in B$ and $1 \le i \le K$, P_1 computes $\frac{\partial z_i(s)}{\partial \mathbf{w}}$. This results in $K \times |B|$ vectors of R-elements each, where R is the number of weights in the last layer.

(3) For each sample $s \in B$ and $1 \le i \le K$, P_1 does element-wise homomorphic scalar multiplication: $\frac{\partial z_i(s)}{\partial \mathbf{w}} [\![y_i(s)]\!]_k = [\![y_i(s) \lfloor r \frac{\partial z_i(s)}{\partial \mathbf{w}} \rfloor]\!]_k$, where r is the precision parameter. This amounts to a total of $K \times |B| \times R$ homomorphic scalar multiplications.

(4) P_1 homomorphically adds:

$$\sum_{s \in B} \sum_{i=1}^{K} \left[\!\!\left[y_i(s) \left\lfloor r \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\rfloor \right]\!\!\right]_k$$
$$= \left[\!\!\left[\sum_{s \in B} \sum_{i=1}^{K} y_i(s) \left\lfloor r \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\rfloor \right]\!\!\right]_k = [\![N_B(\mathbf{w})]\!]_k , \qquad (6)$$

which amounts to a total of $K \times |B| \times R$ homomorphic additions. This results in an R-element vector encrypted under k.

(5) P_1 computes the sensitivity of the gradients of the loss function for the current batch. As shown in Appendix E this is $r \Delta S_B(\mathbf{w})$, where:

$$\Delta S_B(\mathbf{w}) = \frac{2}{|B|} \max_{i,s} \left\| \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\|_2 , \qquad (7)$$

is the sensitivity of the gradients without encoding.

(6) P_2 computes a Gaussian noise vector of dimension R, i.e., $\mathcal{N}(0, \sigma^2 \mathbf{1}_R)$, where $\sigma = 1/\epsilon$ and $\mathbf{1}_R$ is a vector of R 1's. For each value s in the list of allowable values of sensitivity of size t (see below), P_2 updates the noise as $rs\mathcal{N}(0, \sigma^2 \mathbf{1}_R) = \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R)$, and sends $[\![\lfloor \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R) \rfloor]\!]_k$ to P_1.

(7) P_1 looks up the index of the smallest allowable sensitivity s such that $\Delta S_B(\mathbf{w}) \le s$. P_1 then chooses the noise vector $[\![\lfloor \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R) \rfloor]\!]_k$ corresponding to this index as sent by P_2.

(8) P_1 "blinds" the encrypted quantity $[\![N_B(\mathbf{w})]\!]_k$ resulting in $[\![N_B(\mathbf{w}) + \boldsymbol{\mu}]\!]_k$ (see below). P_1 homomorphically adds the scaled noise vector and sends the following to P_2:

$$[\![N_B(\mathbf{w}) + \boldsymbol{\mu} + \lfloor \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R) \rfloor]\!]_k \qquad (8)$$

(9) P_2 decrypts the ciphertext and sends the following to P_1

$$N_B(\mathbf{w}) + \boldsymbol{\mu} + \lfloor \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R) \rfloor \qquad (9)$$

(10) P_1 substracts $\boldsymbol{\mu}$ and obtains $N_B(\mathbf{w}) + \lfloor \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R) \rfloor$. P_1 decodes this by dividing by r, plugs this into Equation (5) and proceeds with backgpropagation in the unencrypted domain.
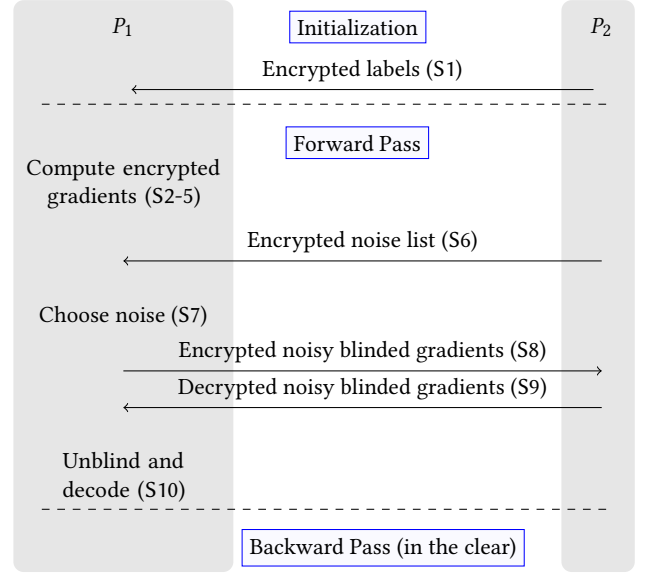

Figure 2: Higher level overview of our protocol for one epoch for training the neural network. The quantity S# in brackets indicates the step number in the protocol.

REMARK 1.   *The parameter R, i.e., the number of weights in the last layer is being leaked here. We assume that this quantity, along with the batch size and the number of epochs are known by party P_2.*

Adding a Random Blind. The labels are encrypted under party P_2's key. Therefore, the computation of the quantity

$$[\![N_B(\mathbf{w}) + \lfloor \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R) \rfloor]\!]_k$$

(without the blind) can be done homomorphically in the encrypted domain. At some point, this needs to be decrypted. However, decrypting this quantity will leak information about the model parameters to P_2. That's why P_1 uses the following construct to "blind" the plaintext model parameters. Namely, assume that encryption of a message x is done under TLWE. Then before decryption, the ciphertext will be of the form:

$$b = \langle \mathbf{s}, \mathbf{a} \rangle + x + e.$$

P_1 samples an element μ uniformly at random from $\mathcal{P}$ and adds it to b. This then serves as a one-time pad, as the original message m can be any of the p possible messages in $\mathcal{P}$. Once the ciphertext has been decrypted, player P_1 receives $x + \mu$, from which μ can be subtracted to obtain m.

Allowable Values of Sensitivity. The multiplication in Step 6 works for Gaussian noise because we can multiply a constant times a Gaussian distribution and still obtain a Gaussian with the scaled variance. If we encrypt the unit variance Gaussian noise first, then multiplication by a constant no longer implies a scaled Gaussian random variable, as the noise needs to be integer encoded before encryption. See Appendix C. As a result, we only use a predefined list of t values of sensitivity. This incurs a slight utility cost, as P_1 would choose a value of sensitivity which is the smallest value

greater than or equal to the true sensitivity in Eq. 7. Since P_2 does not know which of the t values are used by P_1, we maintain secrecy of the actual sensitivity, albeit P_2 now knows that the sensitivity in each batch can only be one of the t allowable values of sensitivity. For the concrete values used in our protocol, see Section 6.4.

4.3 Alternative Solutions

Several other ways of constructing a protocol for the problem addressed in this paper are conceivable. However we discarded them as they had major shortcomings compared to our proposal. One solution is to have P_2 add differentially private noise to the labels of D_2, for instance via randomized response, and send them to P_1 playing no further part in the protocol. While computationally preferable, we address its limitations in terms of accuracy in detail in Section 6.4. Another solution is for P_2 to train its own model on its dataset D_2, and then let P_1 query the trained model to check its consistency with D_{hold}. The major issue with this approach is that our scenario is expecting improvement over the combined dataset $D_1 \cup D_2$, and individually the datasets D_1 and D_2 may drift from D_{hold}. Furthermore, it relies on P_2's expertise in model training. Another possibility is to simply assess the quality of the combined dataset through statistical tests. It is not clear what kind of and how many number of statistical tests would suffice to demonstrate that the machine learning model trained over the combined dataset would show improvement, let alone whether these tests can be efficiently performed in the encrypted domain. Thus, while other alternatives may exist, we believe that they are unlikely to provide substantial improvement over our proposal.

5 Privacy and Security Analysis

5.1 Proving Privacy

Fix a batch B. The quantity $N_B(\mathbf{w})$ from Eq. 6 is a vector of R elements, R being the number of weights in the last layer. This quantity is first encoded into the integer domain (for encryption). In Appendix E, we show that the sensitivity of the loss function when no encoding is employed is given by:

$$\|\nabla L_{B'}(\mathbf{w}) - \nabla L_{B''}(\mathbf{w})\|_2 \leq \frac{2}{|B|} \max_{i,s} \left\| \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\|_2 = \Delta S_B(\mathbf{w}),$$

where B' and B'' are batches in neighbouring datasets D' and D''. We then show that the sensitivity of the loss function when the partial derivative vector is encoded as $\left\| \left\| r \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\| \right\|_2$, as in the quantity $N_B(\mathbf{w})$, is given as: $r\Delta S_B(\mathbf{w})$. Let $rs \geq r\Delta S_B(\mathbf{w})$, where s is the smallest value in the allowable list of sensitivity values that is greater than or equal to $\Delta S_B(\mathbf{w})$. Then the mechanism

$$N_B(\mathbf{w}) + rs\mathcal{N}(0, \sigma^2 \mathbf{1}_R) = N_B(\mathbf{w}) + \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R)$$

is ϵ-GLDP, where $\sigma = 1/\epsilon$, and $\mathbf{1}_R$ is an R-element vector each element of which is 1. To cast this result as an integer, we simply floor the result:

$$\lfloor N_B(\mathbf{w}) + \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R) \rfloor = N_B(\mathbf{w}) + \lfloor \mathcal{N}(0, (rs\sigma)^2 \mathbf{1}_R) \rfloor,$$

as the noise is already of scale $\approx r$. The mechanism on the left remains differentially private due to the post-processing property of Gaussian differential privacy. The expression on the right remains differentially private due to the fact that $N_B(\mathbf{w})$ is an integer. In

other words, we can generate noise and truncate it before adding it to the true value without losing the differential privacy guarantee, as is done in Steps 7 and 8 of the protocol. Since the batches are disjoint in each epoch (if not using a random sample), we retain ϵ-GLDP by invoking parallel composition (Proposition 3). Over n epochs the mechanism is $\sqrt{n}\epsilon$-GLDP invoking sequential composition (Proposition 3).

5.2 Proving Security

Due to lack of space, we present the security proof in Appendix F in the real-ideal paradigm [5, §23.5],[7]. Intuitively, P_1 only learns the noisy gradients of the batches, whereas P_2 does not learn parameters of P_1's model except for batch size $|B|$, number of weights R in the last layer, the number of epochs n, and the fact that the sensitivity can only be approximated to one of the t allowable values of sensitivity.

6 Experimental Evaluation

In this section, we evaluate our protocol over several datasets. We first show that joining two datasets (in cleartext) does indeed improve the accuracy of the model. We then implement our protocol on Zama's `concrete` TFHE library [36], show the training time and accuracy of the trained model, and the values of the privacy parameter ϵ where the model shows accuracy improvement over P_1's dataset but less than the accuracy if the model were to train without differential privacy. This setting is ideal for P_2 as this would persuade P_1 to go ahead with the collaboration without revealing the fully improved model. Our code is available at https://github.com/Ryndalf/Label-Encrypted.

Configurations and Datasets. For all the experiments, we used a 64-bit Ubuntu 22.04.2 LTS with 32G RAM and the 12th Gen Intel(R) Core(TM) i7-12700 CPU. We did not use GPUs for our experiments following existing works [16, 21, 26, 32, 33] to keep the comparison fair. Table 1 shows the common hyperparameters used in our experiments. When using different values (e.g., number of neurons in the hidden layer), we attach them with the experimental results. Table 2 illustrates the brief statistics (number of records, number of features, number of classes and number of samples in each class) of the datasets. All datasets contain (almost) balanced classes, except for Drebin and Purchase-10, where some labels have about twice the number of samples than others.

Table 1: Common hyperparameters for all models.

Hyperparameters	Configuration
activation function	sigmoid
output function	softmax
loss function	cross entropy
optimiser	SGD
$L2$ regulariser	0.01

6.1 Model Improvement by Joining Datasets

In Section 3, we explored if P_2 does not have domain knowledge, whether training the model M_2 on the dataset $D_1 \cup D_2$ will increase

Table 2: Datasets statistics.

Dataset	#Rec.	#Feat.	#Classes	Class distribution
Iris [19]	150	4	3	Uniform Distribution
Seeds [19]	210	7	3	Uniform Distribution
Wine [19]	178	13	3	(59, 71, 58)
Abrupto [23]	10,000	4	2	Uniform Distribution
Drebin [2]	16,680	3,506	2	(5560, 11120)
CIFAR-10 [20]	60,000	3072	10	Uniform Distribution
CIFAR-100 [20]	60,000	3072	100	Uniform Distribution
Purchase-10 [10]	200,000	594	10	(15107, 17341, 22684, 11722, 12727 22257, 29432, 19919, 33842,14969)

the accuracy of M_2 on D_{hold}, with D_{hold} being a balanced dataset. Here, we are interested in knowing the other side of the coin: if P_2's dataset D_2 is indeed of better quality, does this result in an improved performance on M_2? To demonstrate this, we will use the scenario where D_1 is small and imbalanced, i.e., for one of the labels, it has under-representative samples. On the other hand, D_2 is larger and more balanced. Intuitively, joining the two should show substantial improvement in accuracy.

To illustrate the effect, we use Abrupto dataset as a case study. In particular, we use the first $10,000$ samples from the Abrupto dataset [23], i.e., the dataset mixed_1010_abrupto, which is a balance binary dataset. From this dataset, we set aside 200 samples labelled 0 and 200 samples labelled 1 for D_{hold}, 96 samples labelled 0, and 864 samples labelled 1 to dataset D_1, and the remaining samples to dataset D_2. Note that D_2 is considerably larger and more balanced than dataset D_1. Given these datasets, model M_1 is trained on D_1, and model M_2 on $D_1 \cup D_2$, and their accuracies evaluated over D_{hold}.

The configuration is the same for the two models except for the learning rate and batch sizes, which are tailored to account for the relative size difference between datasets D_1 and $D_1 \cup D_2$. With these settings, we report the average test accuracy (over D_{hold}) of training each model 10 times. These are shown in Table 3. There is very little difference in accuracy results in each of these runs. Model M_1 achieves an average accuracy of 0.703, whereas model M_2 achieves an accuracy of 0.912. We therefore conclude that augmenting a small and unbalanced dataset with a large and balanced dataset will increase the accuracy of the resulting model (as long as the dataset labels are of good quality as ascertained by the holdout dataset).

However, as argued in Section 3, an imbalanced D_1 is risky in the sense that $D_1 \cup D_2$ might show improvement even when P_2 does not have any domain expertise. Thus, another, more appropriate scenario is when D_1 is smaller in size than D_2, but otherwise follows a similar label distribution as the true distribution. To see this, we also studied how the ratio of the number of samples of D_1 and of D_2 could impact the accuracy of the trained models M_1 and M_2. In general, a larger size difference between the two models, D_1 being smaller, shows an improvement in model accuracy. This is consistent with the results in Section 3. The experimental results are shown in Appendix G.

6.2 Neural Network Implementation

There are plenty of libraries available for training deep neural networks, e.g., PyTorch. Looking at our protocol, we want to be able

Table 3: Accuracy of Models M_1 (hidden neurons: 4, batch size: 128, learning rate: 0.2, epochs: 100) and M_2 (hidden neurons: 4, batch size: 512, learning rate: 0.1, epochs: 100) against The Holdout Dataset.

Run	Model M_1	Model M_2
1	0.700	0.912
2	0.700	0.915
3	0.712	0.912
4	0.700	0.908
5	0.700	0.915
6	0.700	0.912
7	0.700	0.915
8	0.700	0.908
9	0.718	0.912
10	0.708	0.910
Average	0.704	0.912

to access derivatives of the last layer in order to perform homomorphic encryption operations on them (Eq. (5)). However, existing ML frameworks (e.g., Keras and PyTorch) return all calculated gradients after backpropagation completes. This means that for every epoch we need to wait until PyTorch completes backpropagation, pause it, retrieve all calculated gradients, modify the gradients in the last layer, and restart the completed backpropagation from the modified gradients. To save the run-time from these unnecessary steps, we therefore implement the loss function, activation functions and their derivatives from scratch in Python 3.10 using Numpy and sklearn based on Eq. (5). To ensure our implementation (from scratch) ends up with the same learning outcome as the model implemented by torch.nn (the PyTorch neural network library), we compare the model parameters and the accuracy of the two models over the same training (70% of the raw Iris dataset [19]) and test (30% of the raw Iris dataset [19]) datasets.

Training Outcomes. For the two neural networks (excluding operations such as dropout), if the order in which the datasets are read, the initialization of the weights, and their parameter configurations (learning rate, epochs, etc.) are the same, then their final trained weights are also the same as can be seen from Table 6 in Appendix H, where the weights assigned to each layer are exactly the same in the two implementations up to four decimal places of accuracy. Additionally, Figure 6 in Appendix H further depicts the training performance measured by ROC of our model and the PyTorch implementation. As can be seen the two are almost identical. We are therefore convinced that our implementation from scratch is an accurate representation of the model from PyTorch.

Implementation Time. Table 4 compares the average (of 10 runs) training time (in seconds) of the model M_2 implemented from scratch (Model M_2, Plaintext, no DP), and the PyTorch baseline model (Model M_2 Baseline) on all datasets from Table 2. It is observed that, training time in the plaintext of our implementation is about 100 to 300 times slower than that of the PyTorch baseline.

We analysed the source code of our implementation and the Py-Torch documentation and then concluded three points caused this observation.

- Due to the need to incorporate Gaussian-distributed noise, smaller negative numbers may occur, which can lead to data overflow when the activation function sigmoid is entered. Therefore our program is designed in such a way that for all model training, the sigmoid will be adjusted to Eq. (10).

$$\text{sigmoid}(x) = \begin{cases} \frac{1}{1+e^{-x}}, & \text{if } x \geq 0, \\ \frac{e^x}{1+e^x}, & \text{otherwise.} \end{cases} \tag{10}$$

- PyTorch optimizes low-level implementation for training, i.e., dynamic computational graphs, which means the network behaviour can be changed programmatically at training time to accelerate the training process.
- We implemented more functions in order to accommodate the Concrete library [36] even in the plaintext domain. A main deviation from the general neural network training is the way we treat the gradients. As discussed in Section 4.1, we calculate the gradients separately using two pathways depending on unencrypted (from D_1) vs encrypted labels (from D_2) as shown in Eq. (5). Hence, in our code, we defined a separate function for preparing a batch of sample forward propagation results, which are forced to be divided into two terms for computation.

Encrypted Domain. Zama's current Concrete framework for TFHE [36] mandates decimals to be converted to integers. For our experiments, we chose a precision level of six decimal places, i.e., we use $r = 10^6$. Additionally, in order to verify that the encryption operations do not affect the accuracy of our implementation, we use the Iris dataset for a simple experiment. After setting the same initialization of weights and the order of reading the datasets, we train the model from scratch in plaintext and train another model from scratch on the ciphertext, and compare the trained model parameters. The weights of the two models were almost identical up to four decimal places. The exact weights and biases are shown in Table 7 in Appendix I.

6.3 Protocol Implementation

We use the Concrete framework from Zama [36] to implement the TFHE components of our protocol. Note that not all the operations in our protocol require homomorphic operations. The circuit to compute the protocol operations for one sample is given in Figure 3. Given a sample s of the batch B, from Eq. (6) in Step 4 of the protocol, we first need to multiply the derivatives of the inputs to the softmax function with the encrypted labels. Dropping subscripts and abbreviating notation, these are shown as $z_{\mathbf{w}}$ and y respectively in the figure. For brevity, we skip the integer encoding step, but it is understood that $z_{\mathbf{w}}$ is multiplied by r (the precision parameter), truncated, and then multiplied with y as given by Eq. (6). Next, Step 6 of the protocol samples noise of scale $\mathcal{N}(0, \sigma^2 \mathbf{1}_R)$, which for a single entry with Gaussian differential privacy is $\eta = \mathcal{N}(0, 1/\epsilon^2)$. The noise is then multiplied by the t allowable values of sensitivity and then the truncated noise value is encrypted as in Step 6 of the protocol. Again to avoid excessive notation, we simply show this

as multiplying η by one of the allowable values of sensitivity rs (noting r being the precision parameter). These quantiities are then added homomorphically according to Step 8 of the protocol. On the other side of the circuit, once decrypted, we get the blinded and differentially private gradients (Eq (9)), which result in the differentially private gradients after subtracting the blind.

Implementation Notes. As Concrete only accepts integer inputs, we need to convert the respective inputs to integer equivalents. To do so, we compute $z_{\mathbf{w}}$ to six decimal places, i.e., we multiply it by the precision factor $r = 10^6$, and floor the result. Similarly, we floor $rs \cdot \eta$, since the noise is already of scale r. Thus, the two quantities are in effect multiples of $r = 10^6$. Note that the label y is unchanged, as it is already an integer.

6.4 Plaintext vs Ciphertext Versions of the Protocol

To ensure that the protocol replicates the scenario of Section 6.1, we evaluate our protocol against the unencrypted setting. Namely, we evaluate model M_1 on dataset D_1, model M_2 on dataset $D_1 \cup D_2$, both unencrypted and without differential privacy, and model $\widetilde{M}_2$ on dataset $D_1 \cup D_2$ computed through our protocol (with encryption and differential privacy). We use all the datasets from Table 2 to evaluate this. Furthermore, we retain 3, 506 features from the Drebin dataset which is a subset of all available features. The feature classes retained include 'api_call', 'call', 'feature', 'intent', 'permission', 'provider', and 'real_permission'.

For each dataset and each value of ϵ, we report the average (of 10 runs) results. Each time, the dataset is re-partitioned and 30% of the data is randomly used as D_{hold}. Due to the different sizes of the datasets, we divide D_1 and D_2 differently for different datasets:

- For Iris, Seeds and Wine, D_1 is 10% of the total data and D_2 is 60% of the total data.
- For Abrupto, Drebin, CIFAR-10, CIFAR-100 and Purchase-10, D_1 is 1% of the total data and D_2 is 69% of the total data.

Note that we do not artificially induce a skewed distribution of samples in D_1 for these experiments, as this was done in Section 6.1.

Table 4 shows the results of our experiments. The column ϵ contains the overall privacy budget. We use $t = 100$ allowable values of sensitivity. In all our experiments, we found $\Delta S_B(\mathbf{w})$, i.e., the sensitivity of the gradients without encoding from Eq. (7), ranged from 0.019 to 0.035. We extended this range to $[0, 0.1]$ and picked 100 evenly spaced samples as the $t = 100$ allowed values of sensitivity. For datasets Iris, Seeds and Wine, training our protocol ($\widetilde{M}_2$) is about 10,000 times slower than training the same model in the clear (M_2). However, this time is mainly due to the encryption of the noise vector with all $t = 100$ allowable values of sensitivity. The time per epoch and total time consumed in encrypting this list is shown in the column labelled "t-list time" in the table. The total time is obtained by multiplying per epoch time by 50, the total number of epochs. Subtracting this time means that these three datasets can be trained between 100 to 250 seconds, which is very reasonable. Note that this list of noisy sensitivities does not depend on the datasets. Therefore P_2 can precompute these lists, significantly reducing interactive time. With this speed-up our implementation is only ≈100 times slower than training in the clear,
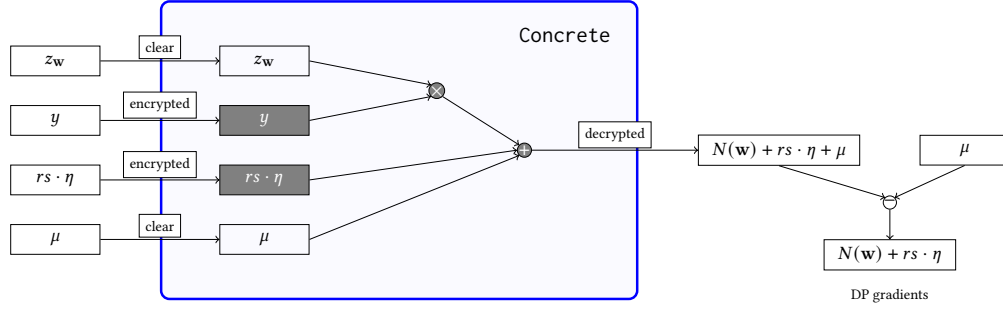
Figure 3: Our homomorphic encryption circuit in Zama's `Concrete`.

i.e., M_2. The $\epsilon = 100$ regime gives little to no privacy. Looking at the numbers corresponding to those rows, we see that the accuracy of $\widetilde{M}_2$ is almost identical to M_2. This shows that our encrypted portion of the protocol does not incur much accuracy loss. For each dataset, we also show values of ϵ between 0.25 and 0.5, where the accuracy of $\widetilde{M}_2$ lies nicely in between M_1 and M_2. Thus, this value of ϵ can be used by P_2 such that $\widetilde{M}_2$ shows an improvement over M_1, yet at the same time, training over the noiseless data promises even more improvement.

The appropriate choice of ϵ depends on the size of the dataset, as it is different for the two larger datasets. One is a synthetic dataset Abrupto and the other one is a real-world Android malware dataset Drebin. Table 4 also shows the results of these two larger datasets. In terms of training time, unlike small datasets, there is a significant difference in their training time. This is due to the large number of features in the Drebin dataset, which is around 876 times more than the features in the Abrupto dataset. However, once again, the majority of the time is consumed in encrypting the noisy t-list, which as we mentioned before, can be precomputed. Since these datasets have multiple batches per epoch, the per epoch time for the t-list is also higher. However, subtracting the total time of the t-lists, the training of these two datasets takes between 2,000 to 11,000 seconds, which is reasonably fast considering the sizes of these datasets. When $\epsilon = 100$, the accuracy of $\widetilde{M}_2$ is almost the same or even better than that of M_2. For Abrupto datasets, at $\epsilon \in [0.15, 0.5]$ we find the spot between the accuracy of model M_1 and M_2. On the other hand, with $\epsilon \in [0.3, 0.5]$ we find the spot between the accuracy of model M_1 and M_2 based on the Drebin dataset.

Randomized Response-based Protocol. In Section 4.1 we briefly mentioned that one way to obtain an updated model M_2 is for P_2 to apply differentially private noise to the labels of D_2, and then handover the noisy labelled dataset to P_1. This will then enable P_1 to run the entire neural network training in the clear, giving significant boost in training time. This can be done by adding noise to the labels via the randomized response (RR) protocol, given in [3], which is used in many previous works as well. However, our experiments show that for a reasonable guarantee of privacy of the labels, the resulting accuracy through RR is less than the accuracy of even M_1 in most cases. To be more precise, let us detail the RR protocol for binary labels. For each label, P_2 samples a bit from a Bernoulli distribution with parameter γ. If the bit is 0, P_2 keeps the

true label, otherwise, P_2 replaces the label with a random binary label [3]. Note that the probability that the resulting label is the true label is $1 - \gamma/2$. The mechanism is ϵ-label differentially private as long as:

$$\frac{1 - \gamma/2}{\gamma/2} \leq e^\epsilon \Rightarrow \epsilon \geq \ln(\gamma/2 - 1) \qquad (11)$$

for all $\gamma < 1$. Setting the value of ϵ exactly equal to the quantity on the right, we see that the probability that the resulting label is the true label is given by $e^\epsilon/(e^\epsilon + 1)$. Thus, for instance, if $\epsilon = 1$, the probability of the label being the true label is $\approx 73\%$, for $\epsilon = \ln 3$ it is 75%, for $\epsilon = 3$ it is $\approx 95\%$, for $\epsilon = 5$ it is $\approx 99.33\%$, and for $\epsilon = 10$ it is $\approx 99.99\%$. Thus, for reasonable privacy we would like to choose ϵ less than 1, and definitely not close to 10, otherwise, P_2 is effectively handing over the true labels to P_1.

We trained the model by using the above mentioned RR protocol. The results are shown in Table 4 under the column labelled M_2^{RR}. As we can see, the accuracy is less than the accuracy of even M_1 for the smaller Iris, Seeds and Wine datasets, only approaching the accuracy between M_1 and M_2 when ϵ is close to 10, but which as we have argued above is not private enough. For the two larger datasets, Abrupto and Drebin, $\epsilon = 1$ gives better accuracy than M_1 while being less than M_2. However, we achieve the same using our protocol for a significantly smaller value of ϵ. This is advantageous because it means that P_1 and P_2 can do multiple collaborations without exhausting the budget.

Effect of the Number of Classes and Dataset Size on Runtime. The analogous results on the three larger datasets, CIFAR-10, CIFAR-100 and Purchase-10 to the ones shown in Table 4 are shown in the Appendix J due to lack of space, where we evaluate the effect of the size of the dataset and the number of classes on the runtime.

Communication Overhead. The TLWE ciphertext is defined as $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{T}_q^{N+1}$. As mentioned in Section 2.4, we use $N = 630$ and q is 64 bits long. This means that the ciphertext is of size: $631 \times 64 = 40,384$ bits. Luckily, as shown in [17], a more compact way to represent $\mathbf{c}$ is to first uniformly sample a random seed θ of 128 bits (the same level of security as the key k), and then use a cryptographically secure PRNG to evaluate the vectors $\mathbf{a} \leftarrow \text{PRNG}(\theta)$. This ciphertext representation is only $128 + 64 = 192$ bits long (adding the bit-lengths of θ and b). This reduction applies only when P_2 generates fresh ciphertexts, as any homomorphic

Table 4: Training time and accuracy of M_1 (dataset D_1), M_2 (dataset $D_1 \cup D_2$) in the clear, $\widetilde{M}_2$ (dataset $D_1 \cup D_2$) through our protocol, $M_2^{\mathbf{Py}}$ (dataset $D_1 \cup D_2$) implemented in PyTorch in the clear, and $M_2^{\mathbf{RR}}$ (dataset $D_1 \cup D_2$) with randomized response (hidden neurons: 20, batch size: 256, learning rate: 0.1, epochs: 50, weight decay: 0.01).

Dataset	ϵ	Model M_1 Plaintext, no DP		Model M_2 Plaintext, no DP		Model $\widetilde{M}_2$ Our protocol			Model M_2^{Py} PyTorch baseline		Model M_2^{RR} Randomized Response
		Time (s)	Test Acc.	Time (s)	Test Acc.	t-list time (s) per epoch & total	Gap Time (s)	Test Acc.	Time (s)	Test Acc.	Test Acc.
Iris	0.1						10m55s	0.7733			0.3711
	1						11m13s	0.8511			0.6244
	10					1m11s	10m52s	0.8521			0.8189
	100	0.0619	0.7289	0.4244	0.8467	≈58m57s	10m49s	0.8554	0.0078	0.8467	0.8311
	0.2						11m14s	0.7821			0.3416
	0.5						11m11s	0.8422			0.6266
Seeds	0.1						13m19s	0.6132			0.3682
	1						13m7s	0.8762			0.6873
	10					1m38s	13m24s	0.8719			0.8550
	100	0.0896	0.8079	0.6216	0.8762	≈1h21m55s	12m54s	0.8730	0.0084	0.8762	0.8762
	0.2						12m35s	0.8111			0.3794
	0.5						13m36s	0.8714			0.6174
Wine	0.1						12m59s	0.7792			0.4361
	1						13m17s	0.9340			0.7981
	10					01m24s	13m5s	0.9396			0.9205
	100	0.1161	0.7981	0.5265	0.9302	≈1h10m12s	12m56s	0.9396	0.0105	0.9302	0.9283
	0.2						13m2s	0.8905			0.4999
	0.5						12m49s	0.9320			0.5168
Abrupto	0.1						4h27m20s	0.7794			0.6593
	1						4h27m5s	0.9028			0.8869
	10					53m15s	4h27m14s	0.9059			0.9004
	100	0.4212	0.8399	28.3430	0.9077	≈1d20h22m38s	4h27m6s	0.9061	0.2687	0.9077	0.9067
	0.2						4h27m3s	0.8850			0.7413
	0.5						4h27m13s	0.9045			0.8916
Drebin	0.1						23h09m8s	0.7010			0.6053
	1						23h09m4s	0.9235			0.9191
	10					1h11m55s	23h09m20s	0.9462			0.9490
	100	8.5758	0.8571	10m28s	0.9470	≈2d11h56m5s	23h09m12s	0.9476	2.8677	0.9470	0.9489
	0.5						23h09m6s	0.8860			0.8741
	0.7						23h09m13s	0.9172			0.9023

computations will, in general, not be able to keep track of the changes to the vectors **a**. With these calculations, we estimate the communication cost of our protocol as follows:

- In Step 1 of the protocol, P_2 sends m_2 encrypted labels under key k of its dataset D_2 of size m_2. Using the compact representation just mentioned, this results in communication cost of $192m_2$ bits. We ignore the cost of plaintext features of D_2 being transmitted, as this cost is the same for any protocol, secure or not.
- In Step 6, P_2 sends the encryption of the R-element noise vector for all t allowable values of sensitivity. Once again, since these are fresh encryptions under k, we have communication cost of $192tR$ bits.
- In Step 8, P_1 sends the R-element encrypted, blinded and noise added vector to P_2. Since this is obtained through homomorphic operations, we use the non-compact representation, yielding $40384R$ bits of communication cost.

- Lastly, in Step 9 P_2 sends the decrypted version of the quantity above. The communication cost is less than $64R$ bits, as this is plaintext space.

Note that Step 6, 8 and 9 need to be repeated for every batch in an epoch. Let $m_{1,2}$ be the size of the dataset $D_1 \cup D_2$. Then the number of times these steps are repeated in each epoch is given by $\lceil m_{1,2}/|B| \rceil$, where $|B|$ is the batch size. Thus the total communication cost for one epoch is $192m_2 + (192tR + 40384R + 64R)(\lceil m_{1,2}/|B| \rceil) = 192m_2 + (192tR + 40448R)(\lceil m_{1,2}/|B| \rceil)$. Over n epochs, the communication cost in bits is thus:

$$C = \left(192m_2 + (192t + 40448)R \left\lceil \frac{m_{1,2}}{|B|} \right\rceil \right) n \tag{12}$$

For the datasets in Table 4, we have $R = 40$, since the number of hidden neurons is 20 (and binary classification), number of epochs $n = 50$, and $t = 100$. The size of D_2, i.e., m_2 is $0.6m$ for Iris, Seeds and Wine datasets, and equals $0.69m$ for the remaining two datasets where m is the total size of the datasets as mentioned in Section 6.4. For all datasets we have $m_{1,2} = 0.7m$. The batch size $|B| = 256$. The communication cost in units of megabytes MB is shown in Table 5.

By far, the major cost is the t encrypted noise values which need to be sent for each batch in an epoch in case of larger datasets. As mentioned earlier, these encrypted noise vectors can be generated well in advance to reduce interactive communication cost, as they do not use of any data specific parameter. For instance, with pre-computed noise lists, the communication cost on the Drebin dataset reduces to 9.54MB per epoch. These times without the encrypted noise vectors are shown in the last two columns of the table labelled "no t-list." Regardless, we see that communication in each round of the protocol (each epoch) can be done in a few seconds with typical broadband speeds.

Table 5: Communication cost in megabytes (MBs) of our protocol as calculated through Eq. 12 for the five datasets used in our implementation.

Dataset	Size m	Cost per epoch	Total cost	Cost per epoch no t-list	Total cost no t-list
Iris	150	1.16MB	58.22MB	0.20MB	10.22MB
Seeds	210	1.17MB	58.26MB	0.21MB	10.26MB
Wine	178	1.16MB	58.24MB	0.20MB	10.24MB
Abrupto	10,000	32.69MB	1,634.34MB	5.81MB	290.33MB
Drebin	16,680	53.70MB	2,685.16MB	9.54MB	477.16MB

6.5 Discussion – Privacy Budget, Training Time and Limitations

Setting ϵ for Data Collaboration. The value of ϵ used depends on the size of the dataset, with a smaller ϵ required for larger datasets. However, for all the datasets used in the experiment, with ϵ in the range $(0.3, 0.5)$ the accuracy of the model $\widetilde{M_2}$ is higher than that of M_1, and lower than that of M_2. Thus, party P_2 can set an ϵ within this range, which provides visible benefits for P_1 to proceed with data collaboration in the clear.

Fast Training of Our Protocol. The training time of our protocol (Table 4) is many orders of magnitude faster than protocols using entirely FHE operation reported in [16, 21, 26]. This is a direct result of keeping the forward and backward propagation in the clear even though some labels from the training data are encrypted. To compare the training time of our protocol against an end-to-end encrypted solution for neural network training, we look at the work from [16]. They use polynomial approximations for the activation functions (e.g., sigmoid) to allow homomorphic operations. In one set of experiments, they train a neural network with one hidden layer over the Crab dataset, which has 200 rows and two classes. This dataset is comparable in size and number of classes to the Iris dataset used by us. They implement the homomorphically encrypted training of the neural network using HELib [15]. The time required for training one batch per epoch is 217 seconds (Table 3a in [16]). For 50 epochs, and processing all batches in parallel, this amounts to 10,850 seconds. We note that after each round the ciphertext is sent to the client to re-encrypt and send fresh ciphertexts back to the server in order to reduce noise due to homomorphic operations. Thus, this time is a crude lower bound on the total time. In comparison, our protocol takes a total time of around 4,200

seconds for the entire 50 epochs over the comparable Iris dataset with one hidden layer, and just 150 seconds if the noise lists are pre-processed. (Table 4). With a more optimised implementation (say via PyTorch), this time can be reduced even further.

Implementation Limitations and Potential Speedups. The minimum value of the total budget ϵ tested by us is 0.1. This is because of the limitation of Concrete in handling high precision real numbers (as they need to be converted into integers). When we inject differentially private noise using a (very) small ϵ, it is highly likely to be inaccurate as it generate numbers with large absolute value, leading to the float overflow problem for sigmoid. In addition to that, our implementation of a neural network is many orders of magnitude slower than the PyTorch benchmark even though our accuracy matches that of the PyTorch baseline. If we are able to access gradients in a batch from the PyTorch implementation, then we would significantly accelerate the proposed protocol. For instance, the model M_2 through our implementation takes about 220 magnitudes more time than the baseline M_2^{Py} through PyTorch on the Drebin dataset, i.e., 10 mins 28 seconds versus only 2.8677 seconds (see Table 4). Thus our protocol on this dataset can potentially run in just over 6 mins if run over PyTorch's implementation which is significantly lower than the 23 hours taken by current implementation. Furthermore, computing mini-batches in parallel can further improve computational time especially if run over GPUs. Lastly, as mentioned above the encrypted values of noise in the t-list can be pre-computed.

Malicious Parties. Our protocol is only secure in the honest-but-curious model. In the malicious setitng, many new challenges arise. A key challenge is to maintain efficiency as several constructs used by us cannot be used in the malicious setting, e.g., the use of the random blind. Another issue in the malicious model is that P_1 may lie about how many samples in the current batch are from P_2's dataset and hence have less noise added to the gradients. We therefore leave protocol for the malicious setting as future work.

7 Related Work

The closest work to ours is that of Yuan et al [34]. They assume a scenario where one of the two parties holds the feature vectors and the labels are secret shared between the two parties. The goal is to jointly train a neural network on this dataset. At the end of the protocol, the first party learns the trained model whereas the second party does not learn anything. Like us, they use label differential privacy to obtain a more computationally efficient solution. Our scenario is different in that it enables party P_1 to assess the quality of the resulting dataset *without knowing* the model, where P_1 does not initially trust the labelling from P_2. Another major difference is that we use FHE instead of secure multiparty computation (SMC) to train the model [34]. Most importantly, it is unclear whether their protocol remains differentially private as they also encode the noise in the group of integers modulo a prime before multiplying with encoded sensitivity. As mentioned in Section 4.2, this multiplication no longer means that the resulting Gaussian is appropriately scaled, resulting in violation of differential privacy. We show this in detail in Appendix C. It is unclear how their protocol can be modified to

include pre-computed shares of a list of allowable sensitivity values as we do to resolve this issue. For instance, one party multiplies all sensitivities with unit Gaussian noise in the allowable list of sensitivites. In order to hide the actual noise, the resulting list needs to be secret shared with the second party. However, the second party is the one which knows the actual sensitivity. Hence telling the first party which share to use will reveal the sensitivity of the gradients.

Apart from Yuan et al [34], several works have investigated training neural network models entirely in the encrypted domain, encompassing both features and labels [16, 21, 26, 32, 33]. Hesami-fard et al [16] propose the CryptoDL framework, which employs Somewhat Homomorphic Encryption (SWHE) and Leveled Homomorphic Encryption (LHE) on approximated activation functions to facilitate interactive deep neural network training over encrypted training sets. However, the training time is prolonged even on small datasets (e.g., on the Crab dataset with dimensions of 200×6 cells, it takes over 200 seconds per epoch/iteration during the training phase). Furthermore, the omission of details regarding the CPU clock speed (frequency) and the number of hidden layer neurons in their experiments renders the reported training time less equitable. Nandakumar et al [26] introduce the first fully homomorphic encryption (FHE)-based stochastic gradient descent technique (FHE-SGD). As pioneers in this field, FHE-SGD investigates the feasibility of training a DNN in the fixed-point domain. Nevertheless, it encounters substantial training time challenges due to the utilisation of BGV-lookup-table-based sigmoid activation functions. Lou et al [21] present the Glyph framework, which expedites training for deep neural networks by alternation between TFHE (Fast Fully Homomorphic Encryption over the Torus) and BGV [6] cryptosystems. However, Glyph heavily relies on transfer learning to curtail the required training epochs/iterations, significantly reducing the overall training time for neural networks. It should be noted that Glyph's applicability is limited in scenarios where a pre-trained teacher model is unavailable. Xu et al propose CryptoNN [33], which employs functional encryption for inner-product [1] to achieve secure matrix computation. However, the realisation of secure computation in CryptoNN necessitates the presence of a trusted authority for the generation and distribution of both public and private keys, a dependency that potentially compromises the security of the approach. NN-emd [32] extends CryptoNN's capabilities to support training a secure DNN over vertically partitioned data distributed across multiple users.

By far, the major bottleneck of end-to-end homomorphic encryption or functional encryption approaches to neural network training is the runtime. For instance, a single mini-batch of 60 samples can take anywhere from 30 seconds to several days with dedicated memory [32]. Another option is to use a secure multiparty computation (SMC) approach. In this case, the (joint) dataset can be secret shared between two parties, and they can jointly train the model learning only the trained model [24]. However, once again end-to-end SMC solutions remain costly. For instance, the work from [24] achieves SMC-based neural network training in 21,000 seconds with a network of 3 layers and 266 neurons. Undoubtedly, these benchmarks are being surpassed, e.g., [18], however, it is unlikely that they will achieve the speed achieved by our solution, or

the one from Yuan et al. [34]. One way to improve our work would be to explore a combination of FHE and SMC.

A related line of work looks at techniques to check the validity of inputs without revealing them. For instance, one can use zero-knowledge range proofs [25] to check if an input is within an allowable range without revealing the input, e.g., age. This has, for instance, been used in privacy-preserving joint data analysis schemes such as Drynx [14] and Prio [11] to ensure that attribute values of datasets are within the allowable range. However, in our case, we do not assume that the party P_2 submits any label that is out of range. Instead, the party may not have the domain expertise to label feature vectors correctly. This cannot be determined through input validity checking.

8 Conclusion

We have shown how two parties can assess the value of their potential machine learning collaboration without revealing their models and respective datasets. With the use of label differential privacy and fully homomorphic encryption over the torus, we are able to construct a protocol for this use case which is many orders of magnitude more efficient than an end-to-end homomorphic encryption solution. Our work can be improved in a number of ways. Due to several limitations in accessing components in PyTorch's neural network implementation and the integer input requirement in Zama's Concrete TFHE framework, our implementation is short of speedups that can potentially be achieved. As a result, we are also not able to test our protocol on larger datasets (say, 100k or more rows). Future versions of this framework may remove these drawbacks. Finally, there could be other ways in which two parties can check the quality of their datasets. We have opted for the improvement in the model as a proxy for determining the quality of the combined dataset.

References

[1] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. 2015. Simple functional encryption schemes for inner products. In *IACR International Workshop on Public Key Cryptography*. Springer, 733–751.

[2] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26.

[3] Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. 2019. The privacy blanket of the shuffle model. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part II 39*. Springer, 638–667.

[4] Avrim Blum and Moritz Hardt. 2015. The ladder: A reliable leaderboard for machine learning competitions. In *International Conference on Machine Learning*. PMLR, 1006–1014.

[5] Dan Boneh and Victor Shoup. 2023. A graduate course in applied cryptography. *Draft 0.6* (2023).

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[7] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.

[8] Kamalika Chaudhuri and Daniel Hsu. 2011. Sample complexity bounds for differentially private learning. In *Proceedings of the 24th Annual Conference on Learning Theory*. JMLR Workshop and Conference Proceedings, 155–186.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[10] Kaggle competition. 2014. Acquire Valued Shoppers Challenge. https://www.kaggle.com/c/acquire-valued-shoppers-challenge/overview

[11] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th {USENIX} Symposium on Networked*

Systems Design and Implementation ({NSDI} 17). 259–282.

[12] Jinshuo Dong, Aaron Roth, and Weijie J Su. 2022. Gaussian differential privacy. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 84, 1 (2022), 3–37.

[13] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*. Springer, 265–284.

[14] D. Froelicher, J. R. Troncoso-Pastoriza, J. S. Sousa, and J. Hubaux. 2020. Drynx: Decentralized, Secure, Verifiable System for Statistical Queries and Machine Learning on Distributed Datasets. *IEEE Transactions on Information Forensics and Security* 15 (2020), 3035–3050.

[15] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481.

[16] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. 2018. Privacy-preserving machine learning as a service. *Proc. Priv. Enhancing Technol.* 2018, 3 (2018), 123–142.

[17] Marc Joye. 2022. SoK: Fully homomorphic encryption over the [discretized] torus. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022), 661–692.

[18] Marcel Keller and Ke Sun. 2022. Secure quantized training for deep learning. In *International Conference on Machine Learning*. PMLR, 10912–10938.

[19] Markelle Kelly, Rachel Longjohn, and Kolby Nottingham. 2022. The UCI Machine Learning Repository. https://archive.ics.uci.edu

[20] Alex Krizhevsky et al. 2009. Learning multiple layers of features from tiny images. (2009).

[21] Qian Lou, Bo Feng, Geoffrey Charles Fox, and Lei Jiang. 2020. Glyph: Fast and accurately training deep neural networks on encrypted data. *Advances in Neural Information Processing Systems* 33 (2020), 9193–9202.

[22] Zhigang Lu, Hassan Jameel Asghar, Mohamed Ali Kaafar, Darren Webb, and Peter Dickinson. 2022. A differentially private framework for deep learning with convexified loss functions. *IEEE Transactions on Information Forensics and Security* 17 (2022), 2151–2165.

[23] Jesús López Lobo. 2020. Synthetic datasets for concept drift detection purposes. https://doi.org/10.7910/DVN/5OWRGB

[24] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[25] Eduardo Morais, Tommy Koens, Cees Van Wijk, and Aleksei Koren. 2019. A survey on zero knowledge range proofs and applications. *SN Applied Sciences* 1 (2019), 1–17.

[26] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. 2019. Towards deep neural network training on encrypted data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. 0–0.

[27] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding machine learning: From theory to algorithms*. Cambridge university press.

[28] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 3–18.

[29] Josh Smith, Hassan Jameel Asghar, Gianpaolo Gioiosa, Sirine Mrabet, Serge Gaspers, and Paul Tyler. 2022. Making the Most of Parallel Composition in Differential Privacy. *Proceedings on Privacy Enhancing Technologies* 1 (2022), 253–273.

[30] Romain Thomas. 2017. LIEF - Library to Instrument Executable Formats. https://lief.quarkslab.com/.

[31] Gerhard Widmer and Miroslav Kubat. 1996. Learning in the presence of concept drift and hidden contexts. *Machine learning* 23 (1996), 69–101.

[32] Runhua Xu, James Joshi, and Chao Li. 2022. NN-EMD: Efficiently Training Neural Networks Using Encrypted Multi-Sourced Datasets. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2022), 2807–2820. https://doi.org/10.1109/TDSC.2021.3074439

[33] Runhua Xu, James BD Joshi, and Chao Li. 2019. Cryptonn: Training neural networks over encrypted data. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1199–1209.

[34] Sen Yuan, Milan Shen, Ilya Mironov, and Anderson Nascimento. 2021. Label Private Deep Learning Training based on Secure Multiparty Computation and Differential Privacy. In *NeurIPS 2021 Workshop Privacy in Machine Learning*.

[35] Umme Zahoora, Asifullah Khan, Muttukrishnan Rajarajan, Saddam Hussain Khan, Muhammad Asam, and Tauseef Jamal. 2022. Ransomware detection using deep learning based unsupervised feature extraction and a cost sensitive Pareto Ensemble classifier. *Scientific reports* 12, 1 (2022), 15647.

[36] Zama. 2022. Concrete: TFHE Compiler that converts python programs into FHE equivalent. https://github.com/zama-ai/concrete.

A Proofs of Theorems from Section 3

Proof of Theorem 1

PROOF. Let the samples in D_{hold} be (x_i, y_i) for $i \in [m]$. Let $\mathbb{1}_{[M(x_i) \neq y_i]}$ denote the indicator random variable which is 1 if $M(x_i) \neq y_i$ for each $i \in [m]$. Then through the linearity of expectation:

$$\mathbb{E}_g[L_{\text{hold}}(M)] = \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}_g[\mathbb{1}_{[M(x_i) \neq y_i]}]$$
$$= \frac{1}{m} \sum_{i=1}^{m} \Pr_g[M(x_i) \neq y_i]$$
$$= \Pr_{g,(x,y) \sim D_{\text{hold}}}[M(x) \neq y],$$

where the subscript g indicates that probability is taken over the randomness in g. The last result follows from the law of total probability. Recall that the notation $(x, y) \sim D_{\text{hold}}$ means the sample is chosen uniformly at random from D_{hold}. Dropping subscripts consider the probability $\Pr[M(x) \neq y]$. We have

$$\Pr[M(x) \neq y] = \Pr[M(x) \neq y \mid y = 1] \Pr[y = 1]$$
$$+ \Pr[M(x) \neq y \mid y = 0] \Pr[y = 0]$$
$$= \Pr[M(x) = 0 \mid y = 1] \Pr[y = 1]$$
$$+ \Pr[M(x) = 1 \mid y = 0] \Pr[y = 0]$$

Now, the learning algorithm $\mathcal{A}$'s input, i.e., D, remains unchanged whether y, i.e., the label of x in D_{hold}, is equal to 0 and 1. This is because D is labelled by g which is independent of the true label y (Eq. 2). Therefore,

$$\Pr[M(x) = 0 \mid y = 1] = \Pr[M(x) = 0],$$

and

$$\Pr[M(x) = 1 \mid y = 0] = \Pr[M(x) = 1].$$

We get:

$$\Pr[M(x) \neq y] = \Pr[M(x) = 0] \Pr[y = 1]$$
$$+ \Pr[M(x) = 1] \Pr[y = 0]$$
$$= \Pr[M(x) = 0] \frac{1}{2} + \Pr[M(x) = 1] \frac{1}{2}$$
$$= \frac{1}{2} (\Pr[M(x) = 0] + \Pr[M(x) = 1]) = \frac{1}{2}$$

Therefore $\mathbb{E}[L_{\text{hold}}(M)] = \Pr_g[M(x) \neq y] = \frac{1}{2}$, where $(x, y) \sim D_{\text{hold}}$. □

Proof of Theorem 2

PROOF. Since D_{hold} is balanced, for $(x, y) \sim D_{\text{hold}}$, and following the proof of Theorem 1, we get

$$
\begin{aligned}
\mathbb{E}_g[L_{\text{hold}}(M_2)] &= \Pr_{g,(x,y)\sim D_{\text{hold}}}[M_2(x) \neq y] \\
&= \Pr_g[M_2(x) = 0 \mid y = 1]\Pr[y = 1] \\
&\quad + \Pr_g[M_2(x) = 1 \mid y = 0]\Pr[y = 0] \\
&= \frac{1}{2}(\Pr_g[M_2(x) = 0 \mid y = 1] \\
&\quad + \Pr_g[M_2(x) = 1 \mid y = 0]) \\
&\geq \frac{1}{2}(\Pr[M_1(x) = 0 \mid y = 1] \\
&\quad + \Pr[M_1(x) = 1 \mid y = 0]) \\
&= \Pr[M_1(x) \neq y] = L_{\text{hold}}(M_1),
\end{aligned}
$$

where the inequality follows from condition (3) in the theorem statement, and we have dropped the subscript $(x, y) \sim D_{\text{hold}}$ from the second equality onwards for better readability. □

Proof of Theorem 3

PROOF. Let

$$
\delta \leq L_{\text{hold}}(M_1) - L_{\text{hold}}(M_2) = (1 - L_{\text{hold}}(M_2)) - (1 - L_{\text{hold}}(M_1)). \tag{13}
$$

From Theorem 2, we know that

$$
\begin{aligned}
\mathbb{E}_g[L_{\text{hold}}(M_2)] &\geq L_{\text{hold}}(M_1) \\
1 - \mathbb{E}_g[L_{\text{hold}}(M_2)] &\leq 1 - L_{\text{hold}}(M_1),
\end{aligned}
$$

Giving us:

$$
\begin{aligned}
&(1 - L_{\text{hold}}(M_2)) - (1 - \mathbb{E}_g[L_{\text{hold}}(M_2)]) \\
&\geq (1 - L_{\text{hold}}(M_2)) - (1 - L_{\text{hold}}(M_1)) \geq \delta \tag{14}
\end{aligned}
$$

Let us call the first term A_1 and the second A_2 in the above three-term inequality. Then

$$
\Pr[A_1 \geq \delta] \leq \Pr[A_2 \geq \delta], \tag{15}
$$

since whenever the event $A_2 \geq \delta$ occurs, the event $A_1 \geq \delta$ is guaranteed to occur due to Eq. (14). Now, consider

$$
\begin{aligned}
1 - L_{\text{hold}}(M_2) &= 1 - \frac{\sum_{i=1}^m \mathbb{1}_{[M_2(x_i) \neq y_i]}}{m} \\
&= \frac{\sum_{i=1}^m \left(1 - \mathbb{1}_{[M_2(x_i) \neq y_i]}\right)}{m}, \tag{16}
\end{aligned}
$$

where $\mathbb{1}_{[M_2(x_i) \neq y_i]}$ is the indicator function which is 1 if the ith sample in D_{hold}, i.e., (x_i, y_i) is mislabelled by M_2. Consider also:

$$
\begin{aligned}
1 - \mathbb{E}_g[L_{\text{hold}}(M_2)] &= 1 - \frac{\sum_{i=1}^m \mathbb{E}_g[\mathbb{1}_{[M_2(x_i) \neq y_i]}]}{m} \\
&= \frac{\sum_{i=1}^m \mathbb{E}_g\left[1 - \mathbb{1}_{[M_2(x_i) \neq y_i]}\right]}{m}, \tag{17}
\end{aligned}
$$

Thus, A_1 in Eq. (15) includes the sum of m random variables minus the sum of expected values of these m random variables. Each of these variables are between 0 and 1, being indicator random variables. Therefore, we can invoke Hoeffding's inequality to conclude, combining Eqs 13, 14, 15, 16 and 17 that:

$$
\begin{aligned}
&\Pr_g[L_{\text{hold}}(M_1) - L_{\text{hold}}(M_2) \geq \delta] \\
&= \Pr_g[(1 - L_{\text{hold}}(M_2)) - (1 - L_{\text{hold}}(M_1)) \geq \delta] \\
&\leq \Pr_g[(1 - L_{\text{hold}}(M_2)) - (1 - \mathbb{E}_g[L_{\text{hold}}(M_2)])] \\
&\leq \exp(-2m\delta^2).
\end{aligned}
$$

□

When Does Condition (3) Hold?

The proofs of Theorems 2 and 3 rely on condition (3) to hold. The condition states that with the model and datasets as defined before, if $(x, y) \sim D_{\text{hold}}$ then

$$
\begin{aligned}
&\Pr_g[M_2(x) = 0 \mid y = 1] + \Pr_g[M_2(x) = 1 \mid y = 0] \\
&\geq \Pr[M_1(x) = 0 \mid y = 1] + \Pr[M_1(x) = 1 \mid y = 0],
\end{aligned}
$$

where the probabilities in the left hand side of the inequality are also over the random choices of g. Notice that this does not necessarily mean that one-sided errors or any particular g may be worse off, i.e., we may have for a particular g that $\Pr[M_2(x) = 0 \mid y = 1] \leq \Pr[M_1(x) = 0 \mid y = 1]$, since M_2 may assign the label 1 to every sample.

The condition is not always satisfied. For example, the condition does not hold if the labels of D_1 are out of distribution. For instance, assume that almost all samples in D_1 have label 0. Further, assume that all the errors made by M_1 are on label 1. This is likely to be the case, as the model is trained overwhelmingly on samples with label 0. In this case, if D_2 has all samples labeled 1 (i.e., the oblivious labeling function $g(x) = 1$, for all x), then M_2 may have decreased error on samples with label 1, and therefore the overall error will be reduced.

On the other hand, if D_1 follows the original distribution $\mathcal{D}$, then since D_2 is labelled according to g, it does not increase $\mathcal{A}$'s information about the original conditional distribution $\mathcal{D}_{y|x}$. Therefore, the overall error of M_2 on D_{hold}, which follows the original conditional distribution $\mathcal{D}_{y|x}$, can at best be greater than or equal to that of M_1 on D_{hold} over all possible random choices of the oblivious functions g, as both are outputs of the same learning algorithm $\mathcal{A}$. We postulate that in this case, the above condition holds.

To test this, we used the purchase dataset [3] clustered into two classes. We used the first 200,000 samples of the dataset. The dataset is relatively balanced with slightly more than 65% of the samples having label 1. We reserved the first 100,000 rows for D_1. The remaining 100,000 rows were reserved for D_2 and D_{hold}. For D_{hold}, we took a 20% sample from these 100,000 samples, and discarded samples so that we have 2,000 samples in D_{hold} with exactly equal distribution of the two labels. Thus, 80,000 samples were reserved for D_2. We trained the Random Forest (RF) classifier from `scikit-learn` with default settings on D_1, and tested the accuracy on D_{hold}. The average accuracy over 100 runs was ≈ 0.90. We then restricted D_1

---

[3] See https://www.kaggle.com/c/acquire-valued-shoppers-challenge/data.

to the first 2,000 samples to simulate lack of samples. We trained the RF classifier again on the updated D_1, and obtained average accuracy of ≈ 0.87. Thus, there is a gap in the two accuracies, and hence room for improvement. We also trained the RF classifier on D_2. For a realistic scenario, we took D_2 to be the first 2,000, 3,000 and 5,000 samples (from the total of 80,000 available samples). This is a realistic setting as increasing the size of the combined dataset is likely to increase accuracy. The average accuracy of the RF classifier on D_2 alone was ≈ 0.89, and on $D_1 \cup D_2$ was ≈ 0.89, both over 100 runs. Thus, the collaboration results in a better model if the two datasets are joined.

Now to check if Theorem 3 holds, we chose $\delta = 0.01$ and $\delta = 0.02$, and selected different (balanced) sizes of D_{hold} upto a maximum size of 2,000. The oblivious labeling function g_q was chosen such that it assigns the label 1 to a given sample with probability $1 - q$, and 0 otherwise, where $q \in \{0, 0.1, 0.2, \ldots, 1\}$. This constitutes a subset of all possible oblivious labeling functions. With these parameters, for each size of D_{hold} in the above set, and for each value of q, we trained the RF classifier on $D_1 \cup D'_2$ where D'_2 had the same samples (features) as D_2, except that the labels were labeled by g_q. We checked if the resulting accuracy was greater than that obtained from the model trained on D_1 plus δ. This process was repeated 100 times for each configuration, and the average number of times the resulting accuracy was greater than this quantity was noted as the estimated probability of Theorem 3.

Figure 4 shows the results of this experiment. For each size of D_{hold}, we pick the highest probability against all possible values of q in g_q. As we can see, the empirical probabilities are lower than the estimated bound from Theorem 3. Furthermore, with increasing size of D'_2, the estimated probabilities are significantly less than the theoretical bound. Thus, in this case the theorem holds. A comprehensive analysis of when the condition holds and whether it is true for other classifiers and datasets requires significant effort. We leave it as an interesting open question.

B Differential Privacy Background

The f-differential privacy framework is based on the hypothesis testing interpretation of differential privacy. Given the output of the mechanism $\mathcal{A}$, the goal is to distinguish between two competing hypotheses: the underlying data set being D_1 or D_2. Let Q_1 and Q_2 denote the probability distributions of $\mathcal{A}(D_1)$ and $\mathcal{A}(D_2)$, respectively. Given any rejection rule $0 \leq \phi \leq 1$, the type-I and type-II errors are defined as follows [12]: $\alpha_\phi = \mathbb{E}_{Q_1}[\phi]$ and $\beta_\phi = 1 - \mathbb{E}_{Q_2}[\phi]$.

Definition 2 (Trade-off function [12]). *For any two probability distributions Q_1 and Q_2 on the same space, the trade-off function $T(Q_1, Q_2) : [0, 1] \to [0, 1]$ is defined by*

$$T(Q_1, Q_2)(\alpha) = \inf\{\beta_\phi : \alpha_\phi \leq \alpha\}$$

for all $\alpha \in [0, 1]$, where the infimum is taken over all (measurable) rejection rules.

A trade-off function gives the minimum achievable type-II error at any given level of type-I error. For a function to be a trade-off function, it must satisfy the following conditions.

Proposition 1 ([12]). *A function $f : [0, 1] \to [0, 1]$ is a trade-off function if and only if f is convex, continuous and non-increasing, and $f(x) \leq 1 - x$ for all $x \in [0, 1]$.*

Abusing notation, let $\mathcal{A}(D)$ denote the distribution of a mechanism $\mathcal{A}$ when given a data set D as input. We now give a definition of f-label differential privacy based on the definition of ordinary f-differential privacy:

Definition 3 (f-Label Differential Privacy [12]). *Let f be a trade-off function. A mechanism $\mathcal{A}$ is said to be f-label differentially private if $T(\mathcal{A}(D_1), \mathcal{A}(D_2)) \geq f$ for all neighbouring data sets D_1 and D_2.*

Note that the only change here from f-differential privacy as defined in [12] is how we define neighbouring datasets. In our case neighbouring datasets differ only in the label of at most one sample. We now give a concrete f-label differentially private mechanism.

Proposition 2 (ϵ-Gaussian Label Differential Privacy [12]). *The mechanism $q(D) + \mathcal{N}(0, \Delta q^2 / \epsilon^2)$ is ϵ-GLDP where Δq is the sensitivity of the function q over any pairs of neighbouring (in label) datasets D_1 and D_2.*

The notion of ϵ-GLDP satisfies both sequential and parallel composition.

Proposition 3 (Sequential and Parallel Composition). *The composition of n-fold sequential ϵ_i-GLDP mechanisms is*

$$\sqrt{\epsilon_1^2 + \cdots + \epsilon_n^2}\text{-GLDP}$$

[12]. Let a sequence of n mechanisms $\mathcal{A}_i$ each be ϵ_i-GLDP. Let $\mathbb{D}_i$ be disjoint subsets of the data domain $\mathbb{D}$. The joint mechanism defined as the sequence of $\mathcal{A}_i(D \cap \mathbb{D}_i)$ (given also the output of the previous $i - 1$ mechanisms) is $\max\{\epsilon_1, \ldots, \epsilon_n\}$-GLDP [29].

Lastly, f-differential privacy is also immune to post processing [12]. That is, applying a randomised map with an arbitrary range to the output of an f-label differentially private algorithm maintains f-label differential privacy.

C Differential Privacy Violation with Integer Encoded Gaussian Noise

In this section, we show that multiplying encoded sensitivity with encoded noise violates differential privacy. This is the main reason why we need the list of allowable sensitivities. This vulnerability also exists in the work from Yuan et al [34]. For ease of exposition, consider the one dimensional case, which can be straighforwardly extended to higher dimensions. Suppose we have a function $\ell(x)$ whose sensitivity is s over neighbouring datasets. We therefore add Gaussian noise $s\mathcal{N}(0, \sigma^2) = \mathcal{N}(0, (s\sigma)^2)$ to $\ell(x)$ to make it differentially private, where for completeness assume that $\sigma = 1/\epsilon$. However, in our case, and also in [34], the sensitivity and the noise are computed by two different parties. We therefore first need to encrypt them and then homomorphically multiply. Encryption implies that these quantities need to be integer encoded. But in general the encoded versions do not satisfy the above relation, i.e.,

$$\lfloor rs \rfloor \cdot \lfloor \mathcal{N}(0, \sigma^2) \rfloor \neq \lfloor rs\mathcal{N}(0, \sigma)^2 \rfloor = \lfloor \mathcal{N}(0, (rs\sigma)^2) \rfloor,$$

where r is the precision used in encoding. For instance, consider $r = 10^6$, $s = 1.2$, and $\mathcal{N}(0, \sigma^2) = 1.1$. Then

$$\lfloor \mathcal{N}(0, (rs\sigma)^2) \rfloor = \lfloor rs\mathcal{N}(0, \sigma^2) \rfloor = \lfloor 10^6 \times 1.2 \times 1.1 \rfloor = 1.32 \times 10^6,$$
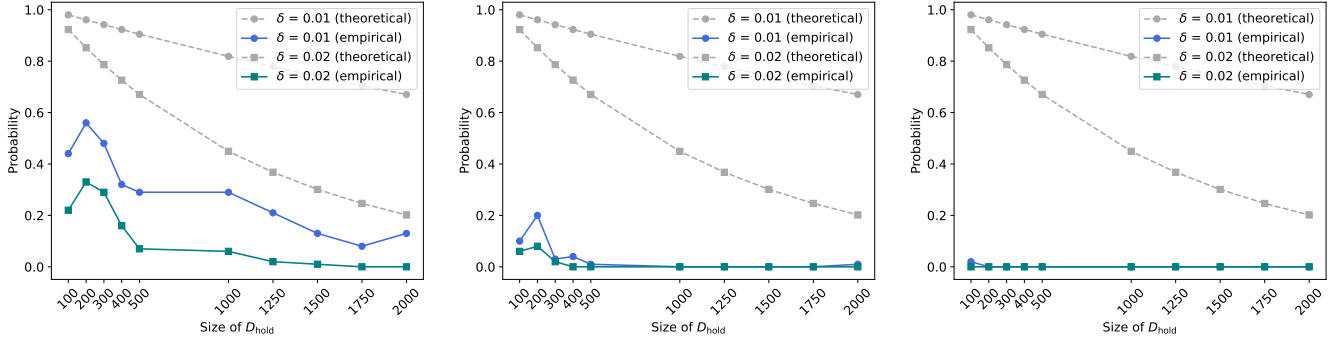
Figure 4: The number of times $L_{\mathbf{hold}}(M_2) \leq L_{\mathbf{hold}}(M_1) + \delta$, where M_1 is trained on D_1 and M_2 on $D_1 \cup D'_2$, and D'_2 labelled through an oblivious function. The size of D'_2 in the leftmost image is 2,000, for the middle image it is 3,000 and for the rightmost image it is 5,000. The empirical probabilities are far below the theoretical bound from Theorem 3 as the size of D'_2 grows.

whereas

$$\lfloor rs \rfloor \cdot \lfloor \mathcal{N}(0, \sigma^2) \rfloor = \lfloor 10^6 \times 1.2 \rfloor \cdot \lfloor 1.1 \rfloor = 1.2 \times 10^6.$$

Note that the authors in [34] mention that to integer encode their scheme one can use the encoding scheme used in [33], which is the same as ours. The fact that the encoded noise does not follow the said property means that the resulting mechanism is no longer differentially private. To see this, again for simplicity consider one dimension and further assume that we have one sample in the batch and the sensitivity is $s = 2$, and therefore $rs = 2r$. Then the encoded noise is a multiple of $2r$, regardless of the actual noise value and the derivative of the loss function. Now, if the initial label is $y = 0$, the quantity with noise remains a multiple of $2r$. However, if the label is $y = 1$, then the quantity with noise is r plus a multiple of $2r$. Thus after unblinding (step 10), P_1 will be able to determine the label as it knows the sensitivity.[4] This reasoning can be extended to the entire mini-batch by assuming the neighbouring datasets to only differ in the label y.

D Computing Gradients via the Chain Rule

From Equation (4), we are interested in computing the loss through the samples in a batch B that belong to the dataset D_2. Overloading notation, we still use B to denote the samples belonging to D_2. The algorithm used to minimizing the loss is the stochastic gradient descent algorithm using backpropagation. This inolves calculating the gradient $\nabla L_B(\mathbf{w})$. As noted in [34], if we are using the backpropagation algorithm, we only need to be concerned about the gradients corresponding to the last layer. Again, to simplify notation, we denote the vector of weights in the *last layer* by $\mathbf{w}$.

Let $s = (\mathbf{x}, \mathbf{y})$ be a sample in the batch B. Let K denote the number of classes. We assume that $\mathbf{y}$ is one-hot encoded, meaning that only one of its element is 1, and the rest are zero. We denote the index of this element by c (which of course is different for different samples). The output from the neural network is the softmax output $\mathbf{p}$ of size

K, where:

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}},$$

where $\mathbf{z} = (z_1, \ldots, z_K)$ is the output before the softmax layer. Consider the cross-entropy loss. Under this loss, we have:

$$L_s(\mathbf{w}) = - \sum_{i=1}^{K} y_i \ln p_i = - \ln p_c, \tag{18}$$

since $\mathbf{y}$ is one-hot encoded. Now, we have:

$$\nabla L_B(\mathbf{w}) = \frac{1}{|B|} \sum_{s \in B} \nabla L_s(\mathbf{w}) \tag{19}$$

Let us, therefore, focus on the gradient of per-sample loss. By the chain rule, we have:

$$\nabla L_s(\mathbf{w}) = \frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{w}} \tag{20}$$

Consider the ith element of $\frac{\partial L}{\partial \mathbf{z}}$:

$$\frac{\partial L}{\partial z_i} = \frac{\partial}{\partial z_i} (- \ln p_c) = - \frac{1}{p_c} \frac{\partial p_c}{\partial z_i}. \tag{21}$$

Solving for the case when $i = c$, the above becomes $p_c - 1$, whereas for the case $i \neq c$, we get p_i. In both cases, we get:

$$\frac{\partial L}{\partial z_i} = p_i - \mathbf{1}_c(i) = p_i - y_i$$

Plugging this into Equation (20), we get:

$$\nabla L_s(\mathbf{w}) = \sum_{i=1}^{K} (p_i - \mathbf{1}_c(i)) \frac{\partial z_i}{\partial \mathbf{w}} \tag{22}$$

---

[4]In Protocol 2 from Yuan et al. [34], Alice will be able to determine the label as she computed the sensitivity. Note that in their protocol it is the quantity $p - y$ which is encoded, where p is the predicted confidence (probability) associated with label y. The reasoning holds by assume $p = 1$ for the input sample.

Plugging this into Equation (19), we get:

$$\nabla L_B(\mathbf{w}) = \frac{1}{|B|} \sum_{s \in B} \sum_{i=1}^{K} (p_i(s) - \mathbf{1}_{c(s)}(i)) \frac{\partial z_i(s)}{\partial \mathbf{w}}$$

$$= \frac{1}{|B|} \sum_{s \in B} \sum_{i=1}^{K} p_i(s) \frac{\partial z_i(s)}{\partial \mathbf{w}}$$

$$- \frac{1}{|B|} \sum_{s \in B} \sum_{i=1}^{K} \mathbf{1}_{c(s)}(i) \frac{\partial z_i(s)}{\partial \mathbf{w}}, \tag{23}$$

where $c(s), p_i(s)$ and $z_i(s)$ denote the fact that these quantities depend on the sample s.

E   Sensitivity of the Gradient of the Loss Function

We now compute the ℓ_2-sensitivity of the loss function. Let D' and D'' be neighbouring datasets, where all but one of the samples have the labels changed to another label. Let s' and s'' represent the differing samples. Let B' and B'' be the batches drawn in the two datasets. Note that $|B'| = |B''|$ and we denote this by $|B|$. Then, the ℓ_2-sensitivity of the loss function, when no encoding is employed, is:

$$\|\nabla L_{B'}(\mathbf{w}) - \nabla L_{B''}(\mathbf{w})\|_2$$

$$= \frac{1}{|B|} \left\| \sum_{s \in B'} \nabla L_s(\mathbf{w}) - \sum_{s \in B''} \nabla L_s(\mathbf{w}) \right\|_2$$

$$= \frac{1}{|B|} \|\nabla L_{s'}(\mathbf{w}) - \nabla L_{s''}(\mathbf{w})\|_2$$

$$= \frac{1}{|B|} \left\| \sum_{i=1}^{K} (p_i - \mathbf{1}_{c(s')}(i)) \frac{\partial z_i}{\partial \mathbf{w}} \right.$$

$$\left. - \sum_{i=1}^{K} (p_i - \mathbf{1}_{c(s'')}(i)) \frac{\partial z_i}{\partial \mathbf{w}} \right\|_2$$

$$= \frac{1}{|B|} \left\| \sum_{i=1}^{K} (\mathbf{1}_{c(s'')}(i) - \mathbf{1}_{c(s')}(i)) \frac{\partial z_i}{\partial \mathbf{w}} \right\|_2$$

$$\leq \frac{1}{|B|} \sum_{i=1}^{K} \left\| (\mathbf{1}_{c(s'')}(i) - \mathbf{1}_{c(s')}(i)) \frac{\partial z_i}{\partial \mathbf{w}} \right\|_2$$

$$= \frac{1}{|B|} \sum_{i=1}^{K} |\mathbf{1}_{c(s'')}(i) - \mathbf{1}_{c(s')}(i)| \left\| \frac{\partial z_i}{\partial \mathbf{w}} \right\|_2$$

$$\leq \frac{1}{|B|} \left(\sum_{i=1}^{K} |\mathbf{1}_{c(s'')}(i) - \mathbf{1}_{c(s')}(i)| \right) \max_i \left\| \frac{\partial z_i}{\partial \mathbf{w}} \right\|_2$$

$$= \frac{2}{|B|} \max_i \left\| \frac{\partial z_i}{\partial \mathbf{w}} \right\|_2$$

And over all possible choice of samples s in the batch, we get that

$$\|\nabla L_{B'}(\mathbf{w}) - \nabla L_{B''}(\mathbf{w})\|_2 \leq \frac{2}{|B|} \max_{i,s} \left\| \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\|_2$$

$$= \Delta S_B(\mathbf{w}), \tag{24}$$

where $z_i(s)$ is the ith coordinate of the vector $\mathbf{z}$ in the sample s.

Now, in Step 4 of our protocol the partial derivative vector $\frac{\partial z_i(s)}{\partial \mathbf{w}}$ from Eq. 23 is encoded as:

$$\left\lfloor r \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\rfloor.$$

Plugging this instead of the partial derivative vector in Eq. 23, we see that the expression remains the same except that now the partial derivative vector has been changed. Therefore, from Eq. 24, we get:

$$\|\nabla \widetilde{L}_{B'}(\mathbf{w}) - \nabla \widetilde{L}_{B''}(\mathbf{w})\|_2 \leq \frac{2}{|B|} \max_{i,s} \left\| \left\lfloor r \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\rfloor \right\|_2,$$

where $\widetilde{L}$ represents the loss function using the encoded partial derivative vector. Next we use the following proposition:

PROPOSITION 4. *Let $\mathbf{x} \in \mathbb{R}^m$ and r be a positive integer. Then*

$$\|\lfloor r\mathbf{x} \rfloor\|_2 \leq r\|\mathbf{x}\|_2.$$

PROOF. We have

$$\|\lfloor r\mathbf{x} \rfloor\|_2 = \sqrt{\sum_{i=1}^{m} (\lfloor rx_i \rfloor)^2}$$

$$\leq \sqrt{\sum_{i=1}^{m} (rx_i)^2} = r\sqrt{\sum_{i=1}^{m} (x_i)^2} = r\|\mathbf{x}\|_2.$$

$$\square$$

Let us denote the vector $\frac{\partial z_i(s)}{\partial \mathbf{w}}$ which maximizes $\left\| \left\lfloor r \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\rfloor \right\|_2$ as $\mathbf{z}$. Then, in light of the above proposition and Eq. 24:

$$\|\nabla \widetilde{L}_{B'}(\mathbf{w}) - \nabla \widetilde{L}_{B''}(\mathbf{w})\|_2 \leq \frac{2}{|B|} \max_{i,s} \left\| \left\lfloor r \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\rfloor \right\|_2$$

$$= \frac{2}{|B|} \|\lfloor r\mathbf{z} \rfloor\|_2$$

$$\leq \frac{2}{|B|} r \|\mathbf{z}\|_2$$

$$\leq \frac{2}{|B|} r \max_{i,s} \left\| \frac{\partial z_i(s)}{\partial \mathbf{w}} \right\|_2$$

$$= r\Delta S_B(\mathbf{w}) \tag{25}$$

F   Proving Security

We will prove security in the real-ideal paradigm [5, §23.5],[7]. Under this framework, we need to consider how we can define ideal functionality for P_2. More specifically, P_2 applies differentially private (DP) noise at places in the protocol. One way around this is to assume that the ideal functionality applies DP noise to the labels of P_2's dataset at the start, and uses these noisy labels to train the dataset. However, this may cause issues with the amount of DP noise added in the ideal world vs the real world. To get around this, we assume that the ideal functionality does the same as what happens in the real-world, i.e., in each batch, the ideal functionality adds noise according to the sensitivity of the batch. We can then argue that the random variables representing the output in both settings will be similarly distributed. We assume R, the number of weights in the last layer, $|B|$, the batch size, and the number of epochs to be publicly known.

The ideal world. In the ideal setting, the simulator S replaces the real-world adversary $\mathcal{B}$. The ideal functionality $\mathcal{F}$ for our problem

is defined as follows. The environment $\mathcal{Z}$ hands the following inputs to the ideal functionality $\mathcal{F}$:

- P_1's inputs: dataset D_1, hold-out set D_{hold}, learning algorithm $\mathcal{A}$, $L_{\text{hold}}(M_1)$, where $M_1 \leftarrow \mathcal{A}(D_1)$, and the features of the dataset D_2, i.e., without the labels, which is the input to party P_1.
- P_2's input: dataset D_2.
- Leaked parameters of M_1: the parameters R, i.e., the number of weights in the last layer, batch size $|B|$, and the number of epochs n.
- System parameters: the privacy parameter ϵ, and the list of t allowable sensitivities to the ideal functionality $\mathcal{F}$.

The ideal functionality $\mathcal{F}$ then proceeds as follows:

- $\mathcal{F}$ first sets $D \leftarrow D_1 \cup D_2$.
- For each epoch, it samples a random batch B of size $|B|$ from D. If B does not contain any sample from D_2, it proceeds with backpropagation, after which it moves to the next epoch. Else it proceeds as follows.
- It computes the sensitivity of the gradients of the loss function for the current batch.
- It then looks up the smallest value s in the list of allowable sensitivity values which is greater than or equal to the computed sensitivity, and generates DP-noise $rs\mathcal{N}(0, \sigma^2 \mathbf{1}_R)$, and adds it to the quantity $N_B(\mathbf{w})$ as $N_B(\mathbf{w}) + \lfloor rs\mathcal{N}(0, \sigma^2 \mathbf{1}_R) \rfloor = \widetilde{N}_B(\mathbf{w})$.
- $\mathcal{F}$ then sends the current batch B and the quantity $\widetilde{N}_B(\mathbf{w})$ to P_1. Note that this information is leaked in our protocol to P_1.
- $\mathcal{F}$ then decodes the quantity as $\widetilde{N}_B(\mathbf{w})/r$ and continues with backpropogation. This is exactly what is done by P_1.
- If this is the last epoch, the functionality outputs 1 if

$$L_{\text{hold}}(M_2) < L_{\text{hold}}(M_1),$$

where M_2 is the resulting training model on dataset D using algorithm $\mathcal{A}$.

Note that each party's input is forwarded directly to the ideal functionality $\mathcal{F}$ (as described above). Thus the simulator $\mathcal{S}$ may ask the ideal functionality for the inputs to and outputs from a corrupt party. Since we assume the honest-but-curious model, $\mathcal{S}$ cannot modify these values. Further note that the ideal functionality is only leaking the noisy gradients $\widetilde{N}_B(\mathbf{w})$ per batch B to P_1, and nothing to P_2 (except for the output and the leaked model parameters as input). In our proof, we shall show that the real world protocol's output is statistically indistinguishable from this.

The real world. In the real world, the environment $\mathcal{Z}$ supplies the inputs (both private and public) to and receives the outputs from both parties. Any corrupt party immediately reports any message it receives or any random coins it generates to the environment $\mathcal{Z}$.

Simulation when P_1 is corrupt. $\mathcal{S}$ first obtains the inputs to P_1 directly from the ideal functionality.

- At some point $\mathcal{Z}$ will generate a control message which results in the real-world P_1 receiving the (label) encrypted dataset D_2 from real-world P_2 (Step 1). To simulate this, $\mathcal{S}$ generates $|D_2|$ fresh samples from the distribution $\mathcal{D}_0$

from Definition 1, and adds one to each row of D_2 as the purported encrypted label. $\mathcal{S}$ reports this encrypted dataset to $\mathcal{Z}$, just like real-world P_1 would do. After this step the two encrypted databases (real and simulated) are statistically indistinguishable due to the TLWE assumption over the torus (Definition 1).

- At a later point $\mathcal{Z}$ generates a control message resulting in P_1 receiving t encrypted R-element noise vectors (one for each allowable value of sensitivity) from P_2 (Step 6). To simulate this, $\mathcal{S}$ again generates tR fresh samples from $\mathcal{D}_0$, and reports the t encrypted R-element vectors as the supposed noise vectors to $\mathcal{Z}$. Notice that, the length of each vector does not depend on the number of elements in the batch belonging to D_2, as long as there is at least one. If none are from D_2, then P_1 will not ask P_2 to send a noise vector. Once again, at this step the real and simulated outputs are statistically indistinguishable under the TLWE assumption over the torus.

- Lastly, $\mathcal{Z}$ generates a control message resulting in P_1 obtaining the blinded and noise-added R-element vector $N_B(\mathbf{w})$ (Step 9). $\mathcal{S}$ queries $\mathcal{F}$ to obtain the batch B and the quantity $\widetilde{N}_B(\mathbf{w})$. Note that these are sent to P_1 and hence the simulator can ask for these inputs. If the batch B does not contain any sample from D_2, $\mathcal{S}$ does nothing. Otherwise, it generates a blind vector $\boldsymbol{\mu}$ (Step 8 of the protocol), by generating fresh coins to generate R elements uniformly at random from $\mathcal{P}$ (the plaintext space). $\mathcal{S}$ reports these coins to $\mathcal{Z}$. $\mathcal{S}$ reports $\widetilde{N}_B(\mathbf{w}) + \boldsymbol{\mu}$ to $\mathcal{Z}$. At this step the real and simulated outputs are *perfectly* indistinguishable due to the fact that the blinds are generated uniformly at random.

- If this is the last epoch, $\mathcal{S}$ sends the output returned by $\mathcal{F}$ to P_1, dutifully to $\mathcal{Z}$.

Thus, overall the output of the environment is statistically indistinguishable from the ideal case under the TLWE assumption over the torus.

Simulation when P_2 is corrupt. The simulation from $\mathcal{S}$ is as follows.

- When $\mathcal{Z}$ sends the initial input to P_2, $\mathcal{S}$ queries $\mathcal{F}$ to obtain this input. At the same time, $\mathcal{S}$ generates fresh coins and uses them to generate the key k for the TLWE scheme. $\mathcal{S}$ reports these coins to $\mathcal{Z}$.

- For generation of Gaussian noise, $\mathcal{S}$ generates an R-element noise vector using fresh coins as $\mathcal{N}(0, \sigma^2 \mathbf{1}_R)$. Then for each s in the list of allowable sensitivities, $\mathcal{S}$ computes

$$rs\mathcal{N}(0, \sigma^2 \mathbf{1}_R).$$

$\mathcal{S}$ then encrypts each of these t vectors under k as

$$\left[\!\left[\lfloor rs\mathcal{N}(0, \sigma^2 \mathbf{1}_R) \rfloor \right]\!\right]_{\mathsf{k}}.$$

$\mathcal{S}$ reports these coins to the environment $\mathcal{Z}$ (as the real-world P_2 would do).

- At some point $\mathcal{Z}$ generates a control message resulting in P_2 receiving the encrypted, blinded and noise-added R-element vector $N_B(\mathbf{w})$ (Step 8). $\mathcal{S}$ generates R elements uniformly at random from $\mathcal{P}$, and then encrypts the resulting R-element

vector under k. $\mathcal{S}$ hands this to $\mathcal{Z}$ as the purported received vector.

- If this is the last epoch, $\mathcal{S}$ sends the output returned by $\mathcal{F}$ to P_2 dutifully to $\mathcal{Z}$.

In this case, in each step, the simulation is perfect. We are using the fact that the blinds used completely hide the underlying plaintext.

G Ratio of the Number of Samples vs Ratio of Model Accuracy

Figure 5 shows the effect of the size difference between D_1 and D_2 on the accuracy of the model M_2 over all datasets used in our work. Recall that M_1 is trained on D_1, and M_2 is trained on $D_1 \cup D_2$. The x-axis shows the ratio $\frac{|D_1|}{|D_2|}$ and the y-axis shows the ratio $\frac{M_1\text{'s Accuracy}}{M_2\text{'s Accuracy}}$. For each dataset, we take D_2 to be one-third of the entire dataset. The size of D_1 is then varied to be $10\%, 20\%, \ldots, 100\%$ of D_2 by sampling points from the remaining dataset. D_{hold} is constructed to be a balanced dataset and its size remains the same across the different ratios. Each data point is averaged over 100 runs.

We have two key observations. First, the ratio of model accuracy being less than one indicates that M_1 has worse accuracy than M_2. Second, the ratio of model accuracy gradually increases and stabilizes as the size of the two becomes similar. This indicates that with the size difference considered in our experiments M_2 shows noticeable improvement over M_1 performance. Hence we conclude that more training data samples result in better training performance. Note that since the deep neural networks are not good on CIFAR datasets [22, 28], the test accuracy of models on CIFAR-10 and CIFAR-100 could not be enhanced by tuning the dataset size, which is reflected by the two flat curves on the two CIFAR datasets. There are drops in accuracy in some of the dataset as we increase the ratio, more prominently in smaller datasets. This is most probably due to the small number of experimental runs (100 per ratio).
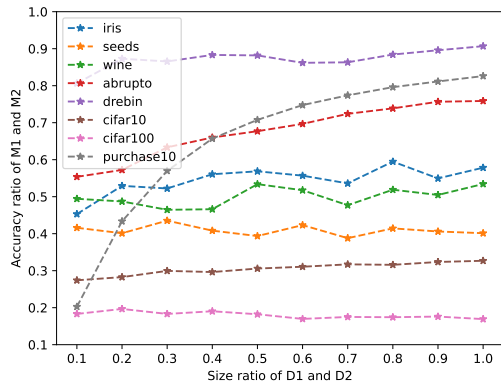

Figure 5: Ratio of Dataset Size vs Ratio of Model Accuracy.

H Neural Network Weights of PyTorch vs Our Implementation

Learned Parameters. Table 6 below shows the weights and biases of the neural network trained via our implementation from scratch versus the PyTorch implementation.

Table 6: Weights and biases of of the neural network training from scratch versus neural netowrk from PyTorch (hidden neurons: [4,4], batch size: 16, learning rate: 0.1, epochs: 100).

Input layer to hidden layer 1					
Type	Weights				Bias
Ours	-0.4916	-0.6516	1.0534	0.6013	-0.2006
	0.298	0.4451	-0.7782	-0.2451	0.119
	-0.4741	-0.7003	1.1115	0.4583	-0.2123
	0.3005	0.6029	-0.754	-0.6411	0.1407
PyTorch	-0.4916	-0.6516	1.0534	0.6013	-0.2006
	0.2980	0.4451	-0.7782	-0.2451	0.119
	-0.4741	-0.7003	1.1115	0.4583	-0.2123
	0.3005	0.6029	-0.754	-0.6411	0.1407

Hidden layer 1 to hidden layer 2					
Type	Weights				Bias
Ours	-1.4098	0.94	-1.3221	1.0272	0.4485
	1.1843	-0.5146	1.0718	-0.9094	-0.2474
	-0.8463	0.5649	-1.0814	0.7063	0.2926
	1.0468	-0.4571	1.0761	-0.6371	-0.1646
PyTorch	-1.4098	0.94	-1.3221	1.0272	0.4485
	1.1843	-0.5146	1.0718	-0.9094	-0.2474
	-0.8463	0.5649	-1.0814	0.7063	0.2926
	1.0468	-0.4571	1.0761	-0.6371	-0.1646

Hidden layer 2 to output layer					
Type	Weights				Bias
Ours	2.0928	-1.7484	1.3045	-1.6554	-
	-0.1631	0.3837	0.1516	0.4605	-
	-2.1381	1.5229	-1.2348	1.0557	-
PyTorch	2.0928	-1.7484	1.3045	-1.6554	-
	-0.1631	0.3837	0.1516	0.4605	-
	-2.1381	1.5229	-1.2348	1.0557	-

Accuracy. Figure 6 depicts the ROC curve from our model versus the PyTorch implementation on the Iris dataset. As can be seen, our model faithfully reproduces the results from PyTorch. We are therefore convinced that our implementation from scratch is an accurate representation of the model from PyTorch.

I Neural Network Weights of Our Model in Plaintext vs Ciphertext

Table 7 shows the weights and biases of our neural network model when no encryption is involved (plaintext) versus those of the model implemented through Zama's `Concrete`, i.e., with homomorphic opertions.
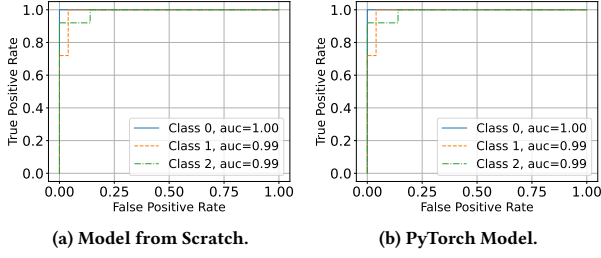
(a) Model from Scratch.　　　　　**(b) PyTorch Model.**

Figure 6: ROC Curves of Model from Scratch and PyTorch Model on The Iris Dataset.

Table 7: Weights and biases of models trained on the Iris dataset in plaintext and ciphertext (hidden neurons: 4, batch size: 16, learning rate: 0.1, epochs: 50).

Input layer to hidden layer 1					
Type	Weight				Bias
Model Trained on Plaintext	0.6511	0.0704	0.2909	1.0547	−0.3171
	0.4029	−0.4721	1.2437	0.6597	0.472
	−0.374	0.5	−0.9141	−0.512	0.0603
	−0.3569	0.3907	−0.9173	−0.8551	−0.0134
Model Trained on Ciphertext	0.6511	0.0704	0.2909	1.0546	−0.3171
	0.4029	−0.4721	1.2437	0.6597	0.472
	−0.374	0.4999	−0.9141	−0.5119	0.0603
	−0.3569	0.3907	−0.9173	−0.8551	−0.0135

Hidden layer 1 to output layer					
Type	Weight				Bias
Model Trained on Plaintext	−1.0483	−1.8553	1.194	1.3057	-
	−0.4617	0.8366	0.2772	−0.0569	-
	0.9567	0.822	−0.9533	−1.4343	-
Model Trained on Ciphertext	−1.0483	−1.8553	1.194	1.3057	-
	−0.4617	0.8366	0.2772	−0.0569	-
	0.9566	0.822	−0.9533	−1.4343	-

J　Effect of the Number of Classes and Dataset Size on Runtime

Table 8 gives the results on the same features as Table 4 but this time on the three larger datasets CIFAR-10, CIFAR-100 and Purchase-10. Unfortunately, due to the large total time required to train the entire datasets, these times are extrapolated from the time taken in one epoch. The neural network for these datasets is trained over 70 epochs as the overall accuracy, especially in case of CIFAR-10 and CIFAR-100 is always low (regardless of encryption and differential privacy). The accuracy is determined by evaluating the accuracy when only differentially private noise is added, i.e., with no encryption, since in our earlier experiments in Table 4 the accuracy with and without encryption match. We acknowledge that the times mentioned in the table are prohibitive. However, once again, this is largely due to the slower time taken by our implementation over the PyTorch implementation. On CIFAR-10, CIFAR-100 and Purchase-10, our implementation is approximately 154, 185, and 71 times slower than that of PyTorch. This means that if we were able to implement our protocol on PyTorch, even without

the other speedups mentioned in Section 6.5, the protocols would take just over 4 hours for CIFAR-10, one day and 14 hours for CIFAR-100, and one day and 2 hours for Purchase-10. These numbers are substantially better than an end-to-end FHE solution. We also note that the pre-processing time is very large for these datasets. Thus, the time and communication complexity can further be improved by a different noise adding mechanism which allows multiplication of sensitivity with unit noise without breaking the differential privacy guarantee. Lastly, we note that the runtime increases with both the size of the dataset and the number of classes. This is because the homomorphic operations are linear in the number of classes K, as mentioned in steps (3) and (4) of the protocol.

Table 8: Extrapolated training time and accuracy of M_1 (dataset D_1), M_2 (dataset $D_1 \cup D_2$) in the clear, $\widetilde{M}_2$ (dataset $D_1 \cup D_2$) through our protocol, M_2^{Py} (dataset $D_1 \cup D_2$) implemented in PyTorch in the clear, and M_2^{RR} (dataset $D_1 \cup D_2$) with randomized response (hidden neurons: 128, batch size: 32, learning rate: 0.01, epochs: 70, weight decay: 0.001).

Dataset	ϵ	Model M_1 Plaintext, no DP		Model M_2 Plaintext, no DP		Model $\widetilde{M}_2$ Our protocol			Model M_2^{Py} PyTorch baseline		Model M_2^{RR} Randomized Response
		Time (s)	Test Acc.	Time (s)	Test Acc.	t-list time (s) per epoch & total	Gap Time (s)	Test Acc.	Time (s)	Test Acc.	Test Acc.
CIFAR-10	1						26d16h	0.1709			0.1967
	10	1m48s	0.3032	1h59m27s	0.4752	6d12h37m43s $\approx$ 456d20h	26d16h	0.2281	46.5314	0.4658	0.4610
	<u>100</u>						26d16h	0.3168			0.4712
	<u>1000</u>						26d16h	0.3544			0.4721
CIFAR-100	1						294d16h	0.0146			0.0139
	10	1m46s	0.0572	2h02m18s	0.2082	69d02h $\approx$ 4836d01h	294d16h	0.0334	39.5251	0.2084	0.2061
	<u>100</u>						294d16h	0.0950			0.2084
	<u>1000</u>						294d16h	0.1088			0.2084
Purchase-10	1						79d20h	0.5970			0.8654
	10						79d20h	0.8025			0.9684
	100	1m46s	0.911	2h03m32s	0.9742	21d13h $\approx$ 1510d3h	79d20h	0.8912	01m44s	0.9742	0.9714
	<u>125</u>						79d20h	0.9312			0.9710
	<u>1000</u>						79d20h	0.9652			0.9721