# Falcon: A Privacy-Preserving and Interpretable Vertical Federated Learning System

Yuncheng Wu
National University of Singapore
wuyc@comp.nus.edu.sg

Naili Xing
National University of Singapore
xingnl@comp.nus.edu.sg

Gang Chen
Zhejiang University
cg@zju.edu.cn

Tien Tuan Anh Dinh
Deakin University
anh.dinh@deakin.edu.au

Zhaojing Luo
National University of Singapore
zhaojing@comp.nus.edu.sg

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

Xiaokui Xiao
National University of Singapore
xkxiao@nus.edu.sg

Meihui Zhang
Beijing Institute of Technology
meihui_zhang@bit.edu.cn

## ABSTRACT

Federated learning (FL) enables multiple data owners to collaboratively train machine learning (ML) models without disclosing their raw data. In the vertical federated learning (VFL) setting, the collaborating parties have data from the same set of users but with disjoint attributes. After constructing the VFL models, the parties deploy the models in production systems to infer prediction requests. In practice, the prediction output itself may not be convincing for party users to make the decisions, especially in high-stakes applications. Model interpretability is therefore essential to provide meaningful insights and better comprehension on the prediction output.

In this paper, we propose Falcon, a novel privacy-preserving and interpretable VFL system. First, Falcon supports VFL training and prediction with strong and efficient privacy protection for a wide range of ML models, including linear regression, logistic regression, and multi-layer perceptron. The protection is achieved by a hybrid strategy of threshold partially homomorphic encryption (PHE) and additive secret sharing scheme (SSS), ensuring no intermediate information disclosure. Second, Falcon facilitates understanding of VFL model predictions by a flexible and privacy-preserving interpretability framework, which enables the implementation of state-of-the-art interpretable methods in a decentralized setting. Third, Falcon supports efficient data parallelism of VFL tasks and optimizes the parallelism factors to reduce the overall execution time. Falcon is fully implemented, and on which, we conduct extensive experiments using six real-world and multiple synthetic datasets. The results demonstrate that Falcon achieves comparable accuracy to non-private algorithms and outperforms three secure baselines in terms of efficiency.

**PVLDB Reference Format:**
Yuncheng Wu, Naili Xing, Gang Chen, Tien Tuan Anh Dinh, Zhaojing Luo, Beng Chin Ooi, Xiaokui Xiao, and Meihui Zhang. Falcon: A Privacy-Preserving and Interpretable Vertical Federated Learning System. PVLDB, 16(10): 2471 - 2484, 2023.
doi:10.14778/3603581.3603588

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:10.14778/3603581.3603588

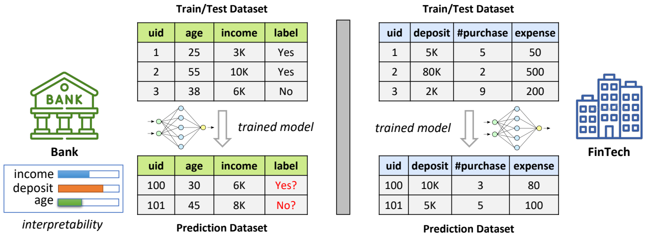
**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/nusdbsystem/falcon.

## 1 INTRODUCTION

Federated learning (FL) [8, 23, 29, 39, 40, 42, 44, 50, 68, 75] enables collaborations among multiple parties, as it provides a framework with which the parties can jointly build machine learning [47, 49, 58] (ML) models from their combined datasets. An essential feature of FL is that it supports such collaborative computation without requiring the parties to disclose their sensitive data, which is important under recent privacy regulations such as GDPR [1] and CCPA [2]. FL can be classified into three settings depending on how the data is partitioned, namely horizontal FL [8, 50], vertical FL [27, 34, 46, 73], and hybrid [75]. In this paper, we focus on the vertical federated learning (VFL) setting, in which the parties hold the data of the same set of users but with different attributes.

Figure 1 shows an example of VFL, in which a bank wishes to build an ML model to predict whether it should approve a credit card application. To improve the model accuracy, it collaborates with a FinTech company to obtain more attributes (features) for the model. For example, the bank has the 'age' and 'income' features while the FinTech company has the 'deposit', 'monthly shopping times', and 'shopping expense' features. Only the bank has the labels, i.e., the decisions on past applications. We refer to the party which has the labels as the *active party* (e.g., the bank) and the other parties providing only the features as the *passive parties* (e.g., the FinTech company). The bank and the FinTech company can collaboratively train a VFL model using their joint datasets, and then deploy the model in their production systems to evaluate new applicants in the *prediction dataset*. Typically, the prediction output consists of a vector of *confidence scores*, where each score indicates the probability of the new applicant belonging to each class label. For example, a prediction output (0.32, 0.68) represents that the bank shall approve the application with a 32% probability. The bank benefits from a more accurate model, while the FinTech company benefits from providing the data via a pay-per-use model [72, 73].

However, the prediction output itself is usually insufficient for the active party to make final decisions, especially in high-stakes applications such as financial risk management [9] and healthcare analytics [17, 48, 76]. For example, the Polish government enforces



Train/Test Dataset

| uid | age | income | label |
|---|---|---|---|
| 1 | 25 | 3K | Yes |
| 2 | 55 | 10K | Yes |
| 3 | 38 | 6K | No |

| uid | age | income | label |
|---|---|---|---|
| 100 | 30 | 6K | Yes? |
| 101 | 45 | 8K | No? |

Prediction Dataset

Train/Test Dataset

| uid | deposit | #purchase | expense |
|---|---|---|---|
| 1 | 5K | 5 | 50 |
| 2 | 80K | 2 | 500 |
| 3 | 2K | 9 | 200 |

| uid | deposit | #purchase | expense |
|---|---|---|---|
| 100 | 10K | 3 | 80 |
| 101 | 5K | 5 | 100 |

Prediction Dataset

**Figure 1: A vertical federated learning example.**

that a customer has the right to receive an explanation in case of a negative credit decision [52]. Thus, simply returning a negative decision with a number 32% to the customer is unacceptable, and it is essential to provide proper explanations (aka. interpretability) so that customers can obtain meaningful insights about the decisions.

Recently, various solutions [14, 25, 26, 30, 34, 43, 57, 66, 69, 73], and several systems, such as SecureML [54], FATE [71], PaddleFL [5], and PySyft [59], have been proposed for privacy-preserving model training and prediction in VFL. However, they suffer from several limitations. First, most existing works [14, 25, 30, 43, 59, 71] protect each party's data through partially homomorphic encryption (PHE) [18, 60] or split learning [70]. However, they may leak sensitive information during training and prediction when some participating parties are semi-honest [24, 38] or malicious [61] because the revealed intermediate results are vulnerable to various inference attacks. Second, while generic secure multiparty computation (MPC) solutions (e.g., MP-SPDZ [36]) can ensure strong privacy protection, i.e., no leakage during the computation, they incur high performance overheads. Several works [5, 26, 32, 54] improve the performance of generic MPC by allowing the parties to outsource their private data to a small number of non-colluding servers, which is not always acceptable. Third, approaches based on trusted hardware [57, 66] can achieve good trade-offs between performance and privacy. However, they make a strong assumption that both the hardware and the software inside the protected environments are trusted, which is often violated in practice [74, 77]. Moreover, none of these works considers the interpretability calculation, which is particularly important in VFL because the active party cannot see the passive parties' data for making a decision.

To bridge the gap, we present Falcon, a privacy-preserving and interpretable vertical federated learning system. Falcon has the following three novelties. *First*, it supports private and efficient VFL training for a variety of ML models, including linear regression, logistic regression, and multi-layer perceptron (MLP). It eliminates the need for outsourcing parties' private data to several non-colluding servers while ensuring there is no intermediate information leakage during the computation. We achieve this strong privacy protection through a hybrid of threshold partially homomorphic encryption (PHE) and additive secret sharing scheme (SSS). In particular, PHE is communication efficient but can only support limited computations, while SSS can execute arbitrary computations but incurs high communication costs. Falcon employs the hybrid strategy with novel designs to improve the VFL model training efficiency.

*Second*, Falcon provides a flexible and privacy-preserving interpretability framework Falcon-INP that supports state-of-the-art interpretable methods in a decentralized setting, such as local interpretable model-agnostic explanations (LIME) [64]. Integrating interpretability in VFL is challenging as the calculation involves features across different parties in multiple stages. If the outputs of intermediate stages are revealed, a party's private information may be inferred by other parties. Falcon-INP addresses this challenge by ensuring each stage's output is securely protected and efficiently adapting the following stages to accommodate the input variations.

*Third*, our system enables system-level optimizations to improve efficiency. It achieves high performance by organizing the executors in each party into a parameter server architecture for parallel execution of both PHE and SSS operations. And we design an auto-parallelism mechanism that finds an effective scheduling strategy to speed up the interpretability calculation.

Falcon consists of three main components: coordinator, agent, and executor, as illustrated in Figure 2. The coordinator accepts a VFL job and schedules it to the agent on each party. The job specifies the parties involved, the private data location of each party, and job configurations such as the hyper-parameters. The coordinator is only responsible for scheduling, and it cannot see the private data and the trained model parameters. Upon receiving the job, the agent on each party launches a set of executor(s) and periodically reports the execution status to the coordinator for further scheduling. The executors across different parties jointly execute the job in a privacy-preserving and peer-to-peer manner.

In summary, we make the following contributions in this paper.

- We propose Falcon, a system that supports VFL training and prediction for a wide range of ML models. It employs PHE and SSS techniques with novel designs to provide strong and efficient privacy protection during the entire execution.
- We design a flexible and privacy-preserving interpretability framework Falcon-INP that supports interpretable solutions such as LIME. To the best of our knowledge, Falcon is the first system that offers prediction interpretability in VFL.
- We present system-level data parallelism optimizations that improve the efficiency of both PHE and SSS operations in Falcon. Moreover, we propose an auto-parallelism mechanism to further speed up the interpretability calculation.
- We implement Falcon system and evaluate its performance using six real-world datasets and multiple synthetic datasets on a distributed cloud cluster. The experimental results demonstrate that Falcon achieves comparable accuracy to non-private algorithms and is more efficient compared to three secure baselines.

The rest of this paper is structured as follows. We introduce preliminaries in Section 2 and give an overview in Section 3. We elaborate the training protocols and the interpretable framework in Sections 4 and 5, respectively. Section 6 presents the optimizations. The experimental evaluation is given in Section 7. We review the related work in Section 8, followed by the conclusions in Section 9.

## 2 PRELIMINARIES

### 2.1 Secure Building Blocks

**Threshold partially homomorphic encryption.** A partially homomorphic encryption (PHE) scheme typically consists of three algorithms (Gen, Enc, Dec). The key generation algorithm $(sk, pk) =$

$\mathsf{Gen}(keysize)$ returns secret key $sk$ and public key $pk$, given a security parameter $keysize$. The encryption algorithm $c = \mathsf{Enc}(x, pk)$ maps a plaintext $x$ to a ciphertext $c$ using $pk$. The decryption algorithm $x = \mathsf{Dec}(c, sk)$ reverses the encryption by $sk$ and outputs the plaintext $x$. For simplicity, we write $\mathsf{Enc}(x, pk)$ as $[x]$.

We adopt the Paillier cryptosystem [18, 60], which offers two homomorphic properties. The first is addition, which allows obtaining the ciphertext of the sum given two ciphertexts, i.e., $[x_1] \oplus [x_2] = [x_1 + x_2]$. The second is plaintext multiplication, which allows computing the ciphertext of the product of given a plaintext and a ciphertext, i.e., $x_1 \otimes [x_2] = [x_1 x_2]$. Since Paillier works on integers, we use fixed-point integer representation [73] to encode floating-point values. Moreover, we employ a threshold variant of Paillier, where the public key $pk$ is known to everyone while each party only holds a partial secret key. Decrypting a ciphertext requires inputs from a minimum (threshold) number of parties. In Falcon, we require all parties to participate in the decryption.

**Additive secret sharing scheme.** Secure multiparty computation (MPC) allows multiple parties to compute a function over their inputs while keeping the inputs private. In this paper, we use the additive secret sharing scheme (SSS) in MP-SPDZ [19, 36]. Specifically, a value $x \in \mathbb{Z}_q$ is additively shared among parties by creating a secret share $\langle x \rangle_i$ for party $P_i (i \in [1, m])$, such that $x = (\sum_{i=1}^{m} \langle x \rangle_i)$ mod $q$ for a large value of $q$. For ease of exposition, we omit the modular operation in the following. Given additive secret shares, we can construct secure primitives (or building blocks), such as addition, multiplication [6], division [11], comparison [10, 11], and exponential [31]. The outputs are also secretly shared and can be reconstructed with the additive secret sharing scheme.

**PHE and SSS conversion.** PHE allows each party to locally compute necessary statistics and transfer the aggregate data to other parties for further computations. Thus, the communication cost is low, and it can easily scale to multiple parties as the parties can do the local computations in parallel. However, PHE only supports limited computations, i.e., addition and plaintext multiplication, therefore it cannot support complex protocols. In contrast, SSS is able to support arbitrary computations but is communication heavy, as most SSS computations require the parties to communicate with each other, resulting in low performance.

PHE and SSS can be converted from one to another [19, 73, 77]. *On the one hand*, a ciphertext $[x]$ can be converted into $\langle x \rangle$ by the $\mathsf{PHE2SSS([x])}$ primitive: (1) each party $P_i (i \in [1, m])$ randomly chooses a value $r_i$, encrypts the value, and sends it to a designated party, say $P_1$; (2) $P_1$ aggregates $[e] = [x] \oplus [r_1] \oplus \cdots \oplus [r_m]$; (3) all parties jointly decrypt $[e]$ to obtain $e$; (4) finally, $P_1$ sets its secret share $\langle x \rangle_1 = e - r_1$ while the other parties $P_i (i \in [2, m])$ sets $\langle x \rangle_i = -r_i$. *On the other hand*, the additive secret shares $\langle x \rangle = (\langle x \rangle_1, \cdots, \langle x \rangle_m)$ can be converted into a PHE ciphertext $[x]$ via $\mathsf{SSS2PHE}(\langle x \rangle)$ as follows. Each party $P_i (i \in [1, m])$ encrypts its secret share $\langle x \rangle_i$ and broadcasts the encrypted share $[x_i]$ to all parties. Each party aggregates the received encrypted shares via homomorphic addition, i.e., $[x] = [x_1] \oplus \cdots \oplus [x_m]$.

## 2.2 Supervised Machine Learning

Given a training dataset $(X, Y)$, where $X$ is the training features and $Y$ is the ground truth labels, supervised machine learning (ML) aims to learn an ML model $f$ with parameters $\boldsymbol{\theta}$ that minimizes the following objective function:

$$L(X, Y; \boldsymbol{\theta}) = -\frac{1}{|X|} \sum\nolimits_{(\boldsymbol{x}, y) \in (X,Y)} \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y) + \lambda \cdot \Omega(f), \quad (1)$$

where $\ell$ is the loss function, $|X|$ is the number of training instances, $(\boldsymbol{x}, y)$ is an instance, $\lambda$ is the regularization parameter, and $\Omega(f)$ is the regularization term to avoid overfitting. Typically, $\Omega(\cdot)$ is L2 regularization $\|\boldsymbol{\theta}\|_2^2$ or L1 regularization $\|\boldsymbol{\theta}\|_1$. To train an ML model, a popular solution is mini-batch stochastic gradient descent (SGD), which is an iterative method. In each iteration, we randomly sample a batch of $b$ instances $(X^b, Y^b)$ and compute the gradients of model parameters $\nabla \boldsymbol{\theta} = \partial L / \partial \boldsymbol{\theta}$; then, we can update the parameters by $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla \boldsymbol{\theta}$, where $\alpha$ is the learning rate. Unless noted otherwise, we omit the superscript $b$ that denotes the batch.

## 2.3 Model-Agnostic Interpretability

Local interpretable model-agnostic explanations (LIME) [64] is one of the most popular interpretable methods [55]. It trains a simple and explainable model (e.g., linear regression) to approximate the predictions of the underlying black box model. The loss function is:

$$\text{explanation}(\boldsymbol{x}) = \arg\min_{g \in G} L(f, g, \pi_{\boldsymbol{x}}) + \Omega(g), \quad (2)$$

where $\boldsymbol{x}$ is the instance being explained and $g$ is the explainable model. $L$ measures how close the explanation that the explainable model $g$ produces to the prediction of the original model $f$ (e.g., the trained VFL model), and the loss function is the weighted mean square error [64]. The proximity measure $\pi_{\boldsymbol{x}}$ defines how large the neighborhood around instance $\boldsymbol{x}$ is for the explanation. $\Omega(g)$ denotes the model complexity, which prevents $g$ to be too complex for explaining the prediction. To train the model $g$ for instance $\boldsymbol{x}$, we sample a set of instances $S = \{\boldsymbol{s}_1, \cdots, \boldsymbol{s}_n\}$ and compute the predictions given $f$, i.e., $Y = f(S)$. Then, we weight the instances according to $\pi_{\boldsymbol{x}}$, obtaining $\boldsymbol{w} = \{w_1, \cdots, w_n\}$. Finally, we train a weighted model based on $(S, Y, \boldsymbol{w})$ to explain the prediction.

# 3 FALCON OVERVIEW

## 3.1 System Model and Threat Model

There are $m$ parties (or data owners) $\{P_1, \cdots, P_m\}$ with the their private datasets $\{X_1, \cdots, X_m\}$ used for model training and prediction. Each row in the datasets corresponds to a data instance, and each column corresponds to a feature. Let $d_i (i \in [1, m])$ be the number of features held by $P_i$, respectively. Since we focus on the model training and prediction protocols, we follow the same assumption in existing VFL works [5, 25, 26, 46, 73] that the datasets $\{X_1, \cdots, X_m\}$ have a common identifier (e.g., national ID or phone number) and are already aligned beforehand, for example, using private set intersection techniques [12, 51, 62, 63]. When such common identifiers do not exist, the parties could employ entity resolution techniques [30, 56] to align the datasets. We consider it an orthogonal direction to this paper. Without loss of generality, we assume that $P_1$ is the active party that has the labels.

We consider the semi-honest model [15, 16, 28, 54, 73] where every party (i.e., the adversary) follows the protocol, but it tries to infer other parties' private data based on the messages received. Like any other party, no additional trust is assumed of the active party.

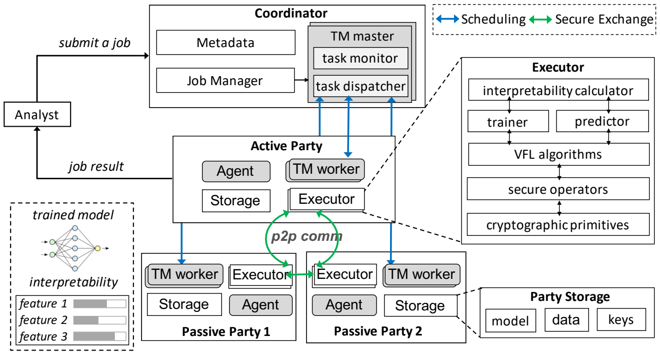


**Figure 2: Overview of Falcon system architecture.**

The adversary can corrupt up to $m-1$ parties. In other words, up to $m-1$ parties can collude to infer the remaining party's private data. Also, we make the standard cryptographic assumptions; for example, the adversary cannot break the PHE and SSS schemes. Finally, Falcon does not protect against attacks launched on the published models or prediction results. Our objective is to guarantee that the computation process does not disclose intermediate information. Investigating techniques such as differential privacy [3, 15, 33, 35] to protect the released outputs is a complementary direction to Falcon, and we expect these techniques could be plugged into our system for further privacy protection.

## 3.2 System Architecture

Figure 2 shows Falcon's system architecture, which consists of three main components: coordinator, agent, and executor.

**Coordinator.** The coordinator schedules jobs on the other components of the system. It can run on a party involved in the VFL task, or be outsourced to a third party. It does not have to be trusted, since it only collects execution status to make scheduling decisions. In other words, it does not see the party's data or model parameters, even in the encrypted format. This is different from other VFL systems, such as FATE [71] and PySyft [59], where the coordinator sees some of the party's intermediate data. The coordinator accepts jobs, such as model training, prediction, and interpretability computation, from the users (e.g., data analysts) and returns the results. The job specifies the parties involved, the private data location for each party, and the job's hyper-parameters.

The coordinator consists of a job manager and metadata. The job manager parses a submitted job into a set of tasks and stores them in the metadata. The task is the scheduling unit in Falcon. Then, the job manager fetches a job and its tasks from metadata and generates a directed acyclic graph (DAG) based on the task dependencies. For each task, the job manager spawns a task manager at the coordinator, referred to as *TM master*, and dispatches the task to the agent on each party, which also spawns a corresponding task manager, referred to as *TM worker*, to execute the task. Once it finishes, the job manager schedules the next task(s) based on DAG.

**Agent.** The agent is local to each party. It receives a task from the coordinator and creates one or multiple task managers (i.e., TM workers), depending on whether the task is running in the centralized mode (see Sections 4 and 5) or distributed mode (see Section 6). Each TM worker creates an executor, manages the life-cycle of that executor, and periodically reports the status to the TM master on the coordinator. If the task is running in the distributed mode, one TM worker creates an executor that runs as the parameter server, while the others create executors that run as workers.

**Executor.** Each executor consists of six main modules. The cryptographic primitive module includes the PHE and SSS building blocks introduced in Section 2.1. The secure operator module supports operations based on the primitives, for example, converting one primitive to another. The VFL algorithm module leverages the secure operators to implement privacy-preserving linear regression, logistic regression, and multi-layer perceptron models. The model trainer module performs model fitting, while the predictor module computes predictions using a trained VFL model. Finally, the interpretability module computes the explanation for a given prediction result. Each executor has a storage module that maintains the data, cryptographic keys, and trained VFL models.

# 4 MODEL TRAINING

In this section, we present model training in Falcon. We aim to provide the same level of privacy protection as secure multiparty computation (MPC) such that no intermediate information other than the agreed output (i.e., the trained VFL model) is disclosed. We note that MPC incurs high communication overhead, while PHE is efficient but can only support limited computations. Our approach is to employ a hybrid of PHE and SSS techniques. The intuition is to let each party compute encrypted statistics using PHE locally (e.g., the aggregated information of the batch data and encrypted model weights), and convert the ciphertexts to SSS when PHE cannot support a given computation. Nevertheless, integrating this hybrid strategy in SGD-based model training still need to address two issues to improve efficiency. First, to prevent intermediate information leakage, we need to encode the floating-point model parameters by fixed-point representation, encrypt and update them with PHE securely. But the parameter update step may contain ciphertexts with different encoding precision due to the homomorphic computations. Thus, we need to handle the incompatible ciphertexts for parameter updates efficiently. Second, an MLP model often contains multiple non-linear layers that PHE does not support. If we simply use SSS to execute the forward-propagation and backward-propagation steps on these non-linear layers, the computation will be degraded to the generic SSS solution, incurring significant overheads. Thus, we need to improve the training efficiency in a scalable manner. We shall discuss them in more detail in subsequent subsections.

## 4.1 Linear Model Training

*Linear regression.* Figure 3 shows a running example of linear regression model training in Falcon with two parties. $P_1$ is the active party that holds two features $X_1$ of an instance and its label $Y$, while $P_2$ is the passive party that holds another two features $X_2$ of this instance. Suppose $P_1$ and $P_2$ have initialized the encrypted model parameter vectors $[\boldsymbol{\theta}_1]$ and $[\boldsymbol{\theta}_2]$ for their own features, respectively. In this iteration, each party $P_i (i \in \{1, 2\})$ first computes the encrypted local aggregation between $X_i$ and $[\boldsymbol{\theta}_i]$ by a homomorphic matrix multiplication operator $\mathsf{MatMul}$ (which will be introduced later),

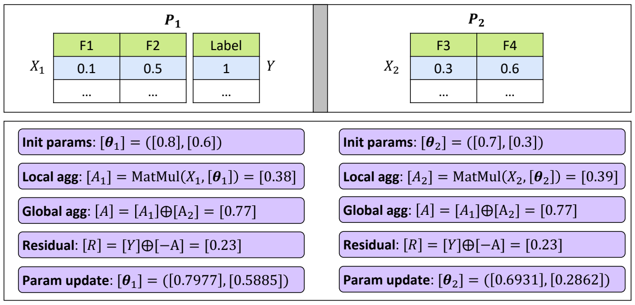

$P_1$
F1
F2
Label
$X_1$
0.1
0.5
1
$Y$
...
...
...
$P_2$
F3
F4
$X_2$
0.3
0.6
...
...
**Init params**: $[\boldsymbol{\theta}_1] = ([0.8], [0.6])$
**Init params**: $[\boldsymbol{\theta}_2] = ([0.7], [0.3])$
**Local agg**: $[A_1] = \mathrm{MatMul}(X_1, [\boldsymbol{\theta}_1]) = [0.38]$
**Local agg**: $[A_2] = \mathrm{MatMul}(X_2, [\boldsymbol{\theta}_2]) = [0.39]$
**Global agg**: $[A] = [A_1] \oplus [\mathrm{A}_2] = [0.77]$
**Global agg**: $[A] = [A_1] \oplus [\mathrm{A}_2] = [0.77]$
**Residual**: $[R] = [Y] \oplus [-\mathrm{A}] = [0.23]$
**Residual**: $[R] = [Y] \oplus [-\mathrm{A}] = [0.23]$
**Param update**: $[\boldsymbol{\theta}_1] = ([0.7977], [0.5885])$
**Param update**: $[\boldsymbol{\theta}_2] = ([0.6931], [0.2862])$

**Figure 3: A linear regression model training example.**

obtaining $[A_i]$. Then, they exchange the ciphertexts and calculate the encrypted global aggregation by $[A] = [A_1] \oplus [A_2]$, where $A$ is exactly the prediction of this instance as $A = A_1 + A_2 = X_1\boldsymbol{\theta}_1 + X_2\boldsymbol{\theta}_2$. Next, $P_1$ calculates the encrypted residual $[R]$ given the encrypted prediction $[A]$ and the ground-truth label $Y$, and sends $[R]$ to $P_2$. Finally, $P_i$ updates $[\boldsymbol{\theta}_i]$ based on the homomorphic properties. In general, linear regression model training has three stages: initialization, forward propagation, and backward propagation.

**Initialization.** Since the $m$ parties hold different features, we let each party $P_i (i \in [1, m])$ initialize the encrypted parameters $[\boldsymbol{\theta}_i]$ w.r.t. their own features. To prevent intermediate information leakage, the model parameters $[\boldsymbol{\theta}] = ([\boldsymbol{\theta}_1], \cdots, [\boldsymbol{\theta}_m])$ will be kept encrypted and updated securely during the entire training process.

**Forward propagation.** In each iteration of the training, the active party $P_1$ randomly samples a batch of $b$ instance IDs and broadcasts it to passive parties. Each party $P_i$ extracts the corresponding batch instances. We abuse the notation $X_i$ to denote the batch instances. The predictions of this batch, i.e., $X\boldsymbol{\theta}$, can be rewritten as follows:

$$X\boldsymbol{\theta} = (X_1 \mid \cdots \mid X_m) \times (\boldsymbol{\theta}_1 \mid \cdots \mid \boldsymbol{\theta}_m)^\top = \textstyle\sum_{i=1}^{m} X_i\boldsymbol{\theta}_i. \tag{3}$$

Therefore, each party first performs local aggregation through the MatMul operator below between $X_i^{b \times d_i}$ and $[\boldsymbol{\theta}_i]^{d_i \times 1}$.

*Definition 4.1.* Homomorphic matrix multiplication operator $\mathrm{MatMul}(U, [V])$ between a plaintext matrix $U^{r \times s}$ and a ciphertext matrix $[V]^{s \times t}$. Let $\boldsymbol{u}_j (j \in [1, r])$ be a row vector in $U$ and $[\boldsymbol{v}_k](k \in [1, t])$ be a column vector in $V$, which have the same dimension $s$. Let $[W]^{r \times t}$ be the multiplication result, then each element $[W_{j,k}]$ is the dot product of $\boldsymbol{u}_j$ and $[\boldsymbol{v}_k]$, which equals $(u_{j,1} \odot [v_{j,1}]) \oplus \cdots \oplus (u_{j,s} \odot [v_{j,s}]) = [u_{j,1}v_{j,1} + \cdots + u_{j,s}v_{j,s}] = [\boldsymbol{u}_j \cdot \boldsymbol{v}_k]$.

Let $[A_i] \in \mathbb{R}^{b \times 1}$ be the encrypted local aggregation result. Each party $P_i$ then broadcasts its $[A_i]$ to all parties so that each party can homomorphically aggregate them, obtaining $[A] = [A_1] \oplus \cdots \oplus [A_m]$, which is the encrypted predictions. $P_1$ also extracts the ground-truth labels $Y$ to compute the instance-wise residual $[R]$ of this batch by $[Y] \oplus ([A] \otimes (-1))$, and broadcasts $[R]$ to passive parties.

**Backward propagation.** Given the encrypted residuals, each party $P_i$ can update its encrypted parameters, i.e.,

$$[\boldsymbol{\theta}_i] := [\boldsymbol{\theta}_i] - \alpha\Big(\frac{1}{b}\textstyle\sum_{j=1}^{b}([R_j] \otimes \boldsymbol{x}_j) + \lambda \otimes [\dfrac{\partial \Omega}{\partial \boldsymbol{\theta}_i}]\Big), \tag{4}$$

where $\boldsymbol{x}_j$ is an instance in $X_i$ and $[R_j]$ is its encrypted residual. For the gradient of the regularization term $[\frac{\partial \Omega}{\partial \boldsymbol{\theta}_i}]$, there are two cases. First, for L2 regularization, each party can directly compute homomorphic multiplication $2\lambda \otimes [\boldsymbol{\theta}_i]$. Second, for L1 regularization, each party needs to compute the signs of $[\boldsymbol{\theta}_i]$, which cannot be directly accomplished as PHE does not support comparison. Thus, the parties need to jointly convert it into secret shares by $\mathrm{PHE2SSS}([\boldsymbol{\theta}_i])$, compute secure comparison, and convert the signs back to ciphertext using $\mathrm{SSS2PHE}(\langle \mathrm{signs} \rangle)$.

A noteworthy aspect is that the precision of the ciphertexts in Eqn 4 may not be aligned. Note that the precision of two ciphertexts needs to be the same for homomorphic addition, and the precision of a homomorphic multiplication is the sum of the precision of the input plaintext and ciphertext. Since we use fixed-point representation in PHE, we transform the plaintext values into fixed-point integers for homomorphic multiplications. But the precision of the resulted ciphertexts increases by the precision of plaintext values, making them not aligned with $[\boldsymbol{\theta}_i]$ in Eqn 4. One solution is to truncate the precision to the same level using a PrecTrunc operator (defined as follows) in each iteration.

*Definition 4.2.* PHE ciphertext precision truncation operator $\mathrm{PrecTrunc}(\mathrm{prec}_1, \mathrm{prec}_2, [u])$, where $\mathrm{prec}_1$ is the current precision and $\mathrm{prec}_2$ is the target precision. The parties first convert $[u]$ into secret shares $\langle u \rangle$ using $\mathrm{PHE2SSS}([u])$, and then invoke $\mathrm{SSS2PHE}(\langle u \rangle)$ to re-encrypt it into ciphertext $[u]$ with precision $\mathrm{prec}_2$.

However, this operator requires threshold decryption, which is computational expensive. Instead, we align the precision by increasing the precision of each ciphertext to the largest precision in Eqn 4. This can be achieved by multiplying a plaintext 1.0 with the designated precision, which is lightweight and can be executed locally. We truncate the precision only when it reaches the maximum precision allowed by the ciphertext to reduce the overheads.

*Logistic regression.* The training process for logistic regression models is similar to linear regression training, except that the parties need to further compute the non-linear Sigmoid function $h(\cdot)$ on the global aggregation $[A]$. To do so, the parties convert $[A]$ into secret shares $\langle A \rangle$ and compute $h(\langle A \rangle)$ with SSS. Afterward, the parties convert it back to PHE for the rest computations similarly.

## 4.2 Multi-layer Perceptron Training

For MLP model training, a straightforward solution is: the parties compute the encrypted inputs of the first hidden layer, convert them into secret shares, execute all the forward and backward operations with SSS, and update the encrypted model parameters. However, it degrades to a generic SSS protocol as most computations are executed by SSS. To mitigate this problem, we propose a layer-wise MLP training method, in which the parties use SSS to compute the layer-wise activation function and output function, while PHE does the encrypted gradient computation and parameter updates. This provides a flexible and efficient solution. The training process also consists of initialization, forward and backward propagation stages. Figure 4 shows an example with three layers on two parties.

**Initialization.** Let $L$ be the number of layers in the MLP model. The parties first initialize the encrypted model parameters $[\boldsymbol{\theta}] = ([\boldsymbol{\theta}^{(1)}], \cdots, [\boldsymbol{\theta}^{(L-1)}])$, where $[\boldsymbol{\theta}^{(l)}]$ is the parameters of layer $l + 1$,

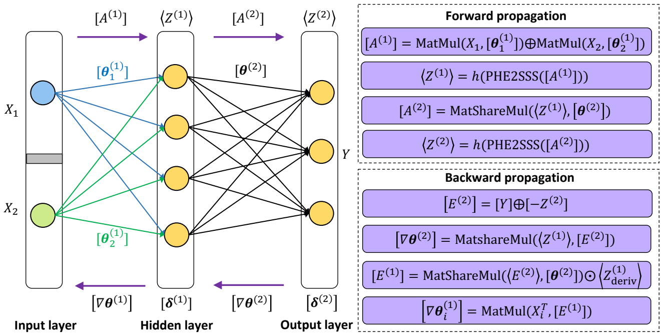

$[A^{(1)}]$
$\langle Z^{(1)}\rangle$
$[A^{(2)}]$
$\langle Z^{(2)}\rangle$
$[\boldsymbol{\theta}_1^{(1)}]$
$[\boldsymbol{\theta}^{(2)}]$
$X_1$
$X_2$
$Y$
$[\boldsymbol{\theta}_2^{(1)}]$
$[\nabla\boldsymbol{\theta}^{(1)}]$
$[\boldsymbol{\delta}^{(1)}]$
$[\nabla\boldsymbol{\theta}^{(2)}]$
$[\boldsymbol{\delta}^{(2)}]$
Input layer
Hidden layer
Output layer
Forward propagation
$[A^{(1)}] = \text{MatMul}(X_1, [\boldsymbol{\theta}_1^{(1)}]) \oplus \text{MatMul}(X_2, [\boldsymbol{\theta}_2^{(1)}])$
$\langle Z^{(1)}\rangle = h(\text{PHE2SSS}([A^{(1)}]))$
$[A^{(2)}] = \text{MatShareMul}(\langle Z^{(1)}\rangle, [\boldsymbol{\theta}^{(2)}])$
$\langle Z^{(2)}\rangle = h(\text{PHE2SSS}([A^{(2)}]))$
Backward propagation
$[E^{(2)}] = [Y] \oplus [-Z^{(2)}]$
$[\nabla\boldsymbol{\theta}^{(2)}] = \text{MatshareMul}(\langle Z^{(1)}\rangle, [E^{(2)}])$
$[E^{(1)}] = \text{MatShareMul}(\langle E^{(2)}\rangle, [\boldsymbol{\theta}^{(2)}]) \odot \langle Z_{\text{deriv}}^{(1)}\rangle$
$[\nabla\boldsymbol{\theta}_i^{(1)}] = \text{MatMul}(X_i^T, [E^{(1)}])$

**Figure 4: A two-party MLP model training example.**

and $l \in [1, L-1]$. Specifically, we let each party $P_i (i \in [1, m])$ initialize $[\boldsymbol{\theta}_i^{(1)}]$ w.r.t. its features for the first hidden layer. The active party initializes the parameters of the remaining layers, say $[\boldsymbol{\theta}^{(2)}]$ in the example, and broadcasts them to all passive parties.

**Forward propagation.** We differentiate two types of layers in forward propagation: the first hidden layer and the remaining layers. *First*, for the input to neurons in the first hidden layer, it is calculated based on $[\boldsymbol{\theta}^{(1)}]$ and the input layer $X$. Since $X$ is distributed on $m$ parties, each party $P_i$ computes $\mathsf{MatMul}(X_i, [\boldsymbol{\theta}_i^{(1)}])$, the partial aggregation w.r.t. its own features. Next, the parties jointly aggregate these partial results (similar to that in linear regression model training) to get the encrypted input to the first hidden layer, say $[A^{(1)}]$. For example, the encrypted input in Figure 4 is:

$$[A^{(1)}] = \mathsf{MatMul}(X_1, [\boldsymbol{\theta}_1^{(1)}]) \oplus \mathsf{MatMul}(X_2, [\boldsymbol{\theta}_2^{(1)}]). \quad (5)$$

Note that the hidden layer often contains non-linear activation functions (e.g., Sigmoid or ReLU), which PHE cannot compute. Hence, we convert the ciphertexts into secret shares $\langle A^{(1)}\rangle$ and calculate $\langle Z^{(1)}\rangle = h(\langle A^{(1)}\rangle)$, where $\langle Z^{(1)}\rangle$ is the secretly shared output of the first hidden layer and $h(\cdot)$ denotes the activation function. We keep this output in the SSS form instead of converting it back to PHE ciphertexts for easier calculation of the following forward and backward propagation steps.

*Second*, for the remaining layers, the parties can jointly compute the input given the encrypted parameters $[\boldsymbol{\theta}^{(l)}]$ and secretly shared output of the previous layer $\langle Z^{(l-1)}\rangle$ for each layer $l \in [2, L-1]$. Specifically, we define the following operator.

*Definition 4.3.* Homomorphic matrix share multiplication operator $\mathsf{MatShareMul}(\langle U\rangle, [V])$ between a ciphertext matrix $[V]^{s\times t}$ and a secretly shared matrix $\langle U\rangle^{r\times s}$, where each party $P_i (i \in [1, m])$ holds a matrix share $\langle U\rangle_i$ in the plaintext form. Thus, $P_i$ first computes $[\langle W\rangle_i] = \mathsf{MatMul}(\langle U\rangle_i, [V])$ to obtain a partial ciphertext matrix. Then, each party $P_i$ sends $[\langle W\rangle_i]$ to a designated party, which aggregates them, i.e., $[\langle W\rangle_1 \oplus \cdots \oplus \langle W\rangle_m] = [W]$.

Let $[A^{(l)}] = \mathsf{MatShareMul}(\langle Z^{(l-1)}\rangle, [\boldsymbol{\theta}^{(l)}])$ be the resulted encrypted input to layer $l+1$. Similarly, the parties can convert it into secret shares for activation function computations until reaching the output layer. Let $\langle Z^{(L-1)}\rangle$ be the secretly shared predictions. The parties convert it back into PHE ciphertexts $[Z^{(L-1)}]$.

**Backward propagation.** Now we compute the encrypted back-propagation error and gradients w.r.t. the output layer and the hidden layers. There are three cases. *First*, if the current layer is the output layer, the back-propagation error is the encrypted residual between the encrypted predictions and ground-truth labels, i.e., $[E^{(L-1)}] = [Y] \oplus ([Z^{(L-1)}] \otimes (-1))$. Given this error and the secretly shared input to this layer, the parties can calculate the encrypted gradient according to the plaintext update rules [7].

$$[\nabla\boldsymbol{\theta}^{(L-1)}] = \mathsf{MatShareMul}(\langle Z^{(L-2)}\rangle, [E^{(L-1)}]^\top). \quad (6)$$

*Second*, if the current layer is a hidden layer but not the first hidden layer, i.e., when $l \in [3, L-1]$, the parties compute the encrypted back-propagation error by:

$$[E^{(l-1)}] = \mathsf{MatShareMul}(\langle E^{(l)}\rangle, [\boldsymbol{\theta}^{(l)}]^\top) \odot \langle Z_{\text{deriv}}^{(l-1)}\rangle, \quad (7)$$

where $\langle E^{(l)}\rangle$ is the back-propagation error of the next layer computed by $\mathsf{PHE2SSS}([E^{(l)})]$, and $\langle Z_{\text{deriv}}^{(l-1)}\rangle$ is the secretly shared derivative of the current layer, which can be computed together with $\langle Z^{(l-1)}\rangle$ using SSS and cached in forward propagation. Moreover, the operation $\odot$ denotes the element-wise multiplication between a ciphertext matrix and a secretly shared matrix, which can be computed by an element-wise homomorphic multiplication and a global aggregation via element-wise homomorphic addition among parties. The encrypted gradient calculation is similar to Eqn 6.

*Third*, if the current layer is the first hidden layer, the parties also use Eqn 7 to compute the back-propagation error. But the encrypted gradient calculation is different because the input to the first hidden layer is the original data held by $m$ parties. Therefore, each party $P_i$ executes the encrypted gradient as follows:

$$[\nabla\boldsymbol{\theta}_i^{(1)}] = \mathsf{MatMul}(X_i^\top, [E^{(1)}]). \quad (8)$$

In Figure 4, the parties first calculate the error and the gradient of the output layer by $[E^{(2)}] = [Y] \oplus [-Z^{(2)}]$ and $[\nabla\boldsymbol{\theta}^{(2)}] = \mathsf{MatShareMul}(\langle Z^{(1)}\rangle, [E^{(2)}]^\top)$. As there is only one hidden layer, they compute $[E^{(1)}] = \mathsf{MatShareMul}(\langle E^{(2)}\rangle, [\boldsymbol{\theta}^{(2)}]) \odot \langle Z_{\text{deriv}}^{(1)}\rangle$, and then each party can obtain its encrypted gradient by Eqn 8. After back-propagation, the regularization can be calculated similarly to that in the linear model training, and the encrypted model parameters $[\boldsymbol{\theta}]$ can be updated accordingly (see Section 4.1).

## 4.3 Discussion

It can be seen that the messages exchanged between the parties during training are in either ciphertext or secret shares. Given the security of PHE and SSS, it follows that there is no leakage during training. Therefore, Falcon provides a stronger privacy protection level compared to FATE [71] and PySyft [59], because they require the parties to reveal plaintext information during training. Moreover, since Falcon employs a general hybrid framework based on PHE and SSS, it tolerates up to $m-1$ corrupted parties, i.e., each party's data privacy is broken only if all $m$ parties are corrupted. In contrast, SecureML [54] and PaddleFL [5] are built on two-party or three-party protocols [22, 53]. That is, each party's data is outsourced to two or three non-colluding external parties. Therefore, their assumption is stronger.

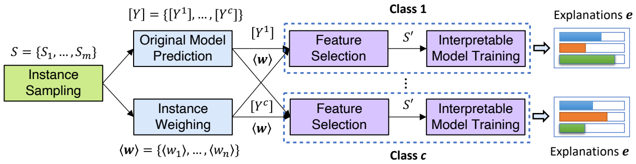

$[Y] = \{[Y^1], \ldots, [Y^c]\}$
Class 1
Explanations $\boldsymbol{e}$
$S = \{S_1, \ldots, S_m\}$
Instance Sampling
Original Model Prediction
Instance Weighing
$[Y^1]$
$\langle \boldsymbol{w} \rangle$
$[Y^c]$
$\langle \boldsymbol{w} \rangle$
Feature Selection
$S'$
Interpretable Model Training
Feature Selection
$S'$
Interpretable Model Training
$\langle \boldsymbol{w} \rangle = \{\langle w_1 \rangle, \ldots, \langle w_n \rangle\}$
Class $c$
Explanations $\boldsymbol{e}$

**Figure 5: The Falcon-INP framework.**

## 5 PREDICTION AND INTERPRETABILITY

Given a trained VFL model, the predictions can be computed following the same forward-propagation step as in model training and then decrypting the outputs. However, as discussed in Section 1, the prediction itself may not be sufficient for the active party to make final decisions, where interpretability is needed to provide more insights into how the prediction is generated.

We focus on model-agnostic interpretability in Falcon so that it can work with any VFL model. We note that existing interpretable methods, such as LIME [64], cannot be directly applied to the VFL setting. The reason is that the calculation involves features held by $m$ participating parties in multiple stages, which cannot be shared due to the privacy requirement. A feasible solution is to adopt the PHE and SSS techniques to protect each stage. However, two issues need to be solved. First, we shall carefully protect the output of each stage since it is calculated on the parties' joint sensitive data. To do so, we can ensure that the outputs of the intermediate stages are in a secure format. Nevertheless, it brings new difficulties to the following calculations due to the input variations. Thus, we shall design the VFL algorithms in the following stages to efficiently accommodate these secure-format inputs. Second, since the interpretability calculation entails expensive cryptographic computations in multiple stages, we shall make it flexible so that it is easier to optimize the execution time and extend with new algorithms.

### 5.1 Privacy-Preserving Interpretability in VFL

We first formulate the privacy-preserving interpretability problem under VFL. Suppose the parties have already built a VFL model $f$. In the model prediction stage, the active party $P_1$ broadcasts a predicting data instance ID, then all the parties jointly compute the model prediction $\boldsymbol{y} = f(\boldsymbol{x})$, where $\boldsymbol{x} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_m)$ is the predicting instance and $\boldsymbol{x}_i$ denotes the feature values held by $P_i$. When $f$ is a classification model, $\boldsymbol{y} = (y^1, \cdots, y^c)$ is the probability vector for being in each class, and $c$ is the number of classes. When $f$ is a regression model, $\boldsymbol{y}$ is a single value. Interpretability is defined as an explanation $\boldsymbol{e}$ for the prediction $\boldsymbol{y}$. The active party uses this explanation to understand how the prediction is derived. Similarly, we require no additional information leakage during the computation of $\boldsymbol{e}$. More formally, the interpretability computation takes $\{f, \boldsymbol{x}, \boldsymbol{y}\}$ as an input and returns $\boldsymbol{e}$, while the intermediate information calculated on the parties' data is protected.

### 5.2 Falcon-INP Framework

To solve the problem, we design a flexible and extensible framework Falcon-INP, which consists of five stages: instance sampling, original model prediction, instance weighing, feature selection, and interpretable model training, as illustrated in Figure 5. Each stage is defined as a task in Falcon (see Section 3.2) and can be easily extended with new algorithms. We detail each stage as follows.

**Instance sampling.** To explain a prediction $\boldsymbol{y} = (y^1, \cdots, y^c)$ of an instance $\boldsymbol{x} = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_m\}$ in VFL, the parties first sample instances for training the interpretable model. We let $P_i$ sample a dataset $S_i = \{\boldsymbol{s}_1^i, \cdots, \boldsymbol{s}_n^i\}$ with $n$ instances, where $\boldsymbol{s}_j^i = \{s_{j,1}^i, \cdots, s_{j,d_i}^i\}$ $(j \in [1, n])$, and $d_i$ is the number of local features held by $P_i$. For example, regarding LIME, $P_i$ samples $n$ instances with $d_i$ features around the predicting instance $\boldsymbol{x}_i$. Then, the parties organize the sampled instances $S = \{S_1, \cdots, S_m\}$ in a vertically partitioned manner. Note that each party's local instance set $S_i$ implicitly contains the statistic information of its private data of $\boldsymbol{x}_i$, we therefore need to protect $S_i$ during the calculation process to uphold each party's privacy.

**Original model prediction.** In this stage, the parties compute the predictions $Y = \{\boldsymbol{y}_1, \cdots, \boldsymbol{y}_n\}$ based on $S$ and the trained VFL model $f$. In Falcon-INP, we treat $f$ as a blackbox and invoke its secure prediction API for the calculation. For example, $f$ can be the linear and MLP models presented in Section 4, or other VFL models (e.g., in [54, 73]) as long as they provide a prediction API. This design enables our framework to easily support the VFL models trained by other systems and provide explanations. We assume that the VFL prediction process is secure such that it satisfies our privacy requirement in Section 5.1. However, privacy risks may exist if we reveal the plaintext predictions $Y$ to the parties. For instance, the active party can infer the passive parties' feature values with high accuracy if given the plaintext model predictions [46], and thus sensitive information may be disclosed. To mitigate possible leakages, we require the calculated predictions to be in the encrypted format, say $[Y] = \{[\boldsymbol{y}_1], \cdots, [\boldsymbol{y}_n]\}$, as the output of this stage.

**Instance weighing.** Next, the parties compute the proximity of the sampled instances $S$ from the predicting instance $\boldsymbol{x}$, and derive the instance weights, denoted by $\boldsymbol{w}$. Falcon-INP supports the Euclidean distance for proximity measurement $\pi$ and Exponential kernel function $\kappa$. For each instance $\boldsymbol{s} \in S$, where $\boldsymbol{s} = (\boldsymbol{s}_1, \cdots, \boldsymbol{s}_m)$ and $\boldsymbol{s}_i (i \in [1, m])$ is held by party $P_i$, each party first computes the square of Euclidean-norm $a_i = \|\boldsymbol{x}_i - \boldsymbol{z}_i\|_2^2$ locally. Afterward, the parties provide their partial results $a_1, \cdots, a_m$ as secret shares, and compute the instance weight $\langle w \rangle = \exp(-\frac{(\sum_{i=1}^m \langle a_i \rangle)^2}{2\ell^2})$ using SSS, where $\ell$ is the kernel width, and exp is the secure exponential function. We require that these instance weights are also protected and keep them in the secret-share format, say $\langle \boldsymbol{w} \rangle$.

**Feature selection.** Given the sampled instances $S$, encrypted predictions $[Y]$, and secretly shared weights $\langle \boldsymbol{w} \rangle$, the parties can explain the prediction for each label $k \in [1, c]$, where $c$ is the number of classes for classification and is one for regression. For a given $k$ to be explained, the parties first extract the corresponding encrypted predictions $[Y^k] = \{[y_1^k], \cdots, [y_n^k]\}$ from $[Y]$, i.e., the probability of predicting class $k$ in classification. We abuse $[Y]$ to denote $[Y^k]$ in the following for simplicity. Then, an optional stage is feature selection to explain several significant features, say $d_{\text{sel}}$ features.

We currently support a simple but effective method based on the weighted Pearson correlation coefficient (WPCC). The plaintext WPCC between a feature $F$ and the label $Y$ can be calculated by:

$$\rho(F, Y; \boldsymbol{w}) = \frac{\sum_{t=1}^n w_t (y_t - \mu_Y)(f_t - \mu_F)}{\sqrt{\sum_{t=1}^n w_t (f_t - \mu_F)^2} \sqrt{\sum_{t=1}^n w_t (y_t - \mu_Y)^2}}, \qquad (9)$$

where $\mu_a = \frac{\sum_t w_t a_t}{\sum_t w_t}$ and $\mathrm{cov}(A, B; \boldsymbol{w}) = \frac{\sum_t w_t (a_t - \mu_a)(b_t - \mu_b)}{\sum_t w_t}$ are the weighted mean of a vector and weighted covariance between two vectors, respectively. Note that the instance weight $w_t$ and the prediction $y_t$ are in secret formats. One straightforward method is to simply utilize the hybrid framework of PHE (e.g., for local statistics computation) and SSS (e.g., for ciphertext multiplication) in Section 4. However, it results in frequent conversion back and forth, which is undesirable as the conversion operations are costly. Specifically, it needs $O(dn)$ conversions to calculate Eqn 9, where $d$ and $n$ are the numbers of total features and instances, respectively.

We devise an optimized method for the WPCC-based feature selection. The basic idea is to identify the common terms used in Eqn 9 for different features, and reuse the results in the following executions. In particular, we observe that the element-wise $w_t(y_t - \mu_Y)$ in Eqn 9 can be reused for different features. We first calculate the weighted mean by $\langle \mu_Y \rangle = \frac{\mathsf{MatShareMul}(\langle \boldsymbol{w} \rangle, [Y])}{\sum_{i=t}^{n} \langle w_t \rangle}$ and convert it to $[\mu_Y]$, where the dimensions of the inputs in the $\mathsf{MatShareMul}$ operator are $1 \times n$ and $n \times 1$, respectively. Then, we compute the element-wise term above using $\langle r_t \rangle = \mathsf{PHE2SSS}(\mathsf{MatShareMul}(\langle w_t \rangle, [y_t] - [\mu_Y]))$ for $t \in [1, n]$. Let $\langle \boldsymbol{r} \rangle$ be the resulting vector, which only needs to be computed once, and we can calculate the second item of the denominator in Eqn 9 by $\langle q_2 \rangle = \mathsf{MatMulShare}(\langle \boldsymbol{r} \rangle, [Y] - [\mu_Y])$.

Next, we calculate feature-dependent items. For each feature $F$, we compute $[\mu_F] = \mathsf{SSS2PHE}(\frac{\mathsf{MatMul}(F, [\boldsymbol{w}])}{\sum_{t=1}^{n} \langle w_t \rangle})$, and $[\mu_F^2]$ as an auxiliary term to reduce the computations later. Afterward, we obtain the numerator by $\langle p \rangle = \mathsf{PHE2SSS}(\mathsf{MatShareMul}(\langle \boldsymbol{r} \rangle, [F] - [\mu_F]))$ in Eqn 9. Then, we compute element-wise values $[v_t] = (f_t - [\mu_F])^2 = ([f_t^2] \oplus (-2 f_t) \otimes [\mu_F] \oplus [\mu_F^2])$, where $f_t$ is the plaintext feature value held by each party and $[\mu_F^2]$ is cached in the previous step. Let $[\boldsymbol{v}]$ be the resulting vector. The parties can obtain $\langle q_1 \rangle = \mathsf{PHE2SSS}(\mathsf{MatShareMul}(\langle \boldsymbol{w} \rangle, [\boldsymbol{v}]))$ as well as the WPCC of this feature by $\langle p \rangle / (\sqrt{\langle q_1 \rangle} \sqrt{\langle q_2 \rangle})$.

Finally, the parties can sort and compare the WPCC values using SSS [73] and select the top-$d_{\mathrm{sel}}$ features with the largest ones. Each party $P_i$ then filters its dataset $S_i$ based on the selected features belonging to itself. Notice that the proposed method requires only $O(d + n)$ conversion operations, which is much more efficient.

**Interpretable model training.** Falcon-INP uses linear regression as the interpretable model. Since both the predictions and instance weights are in secure formats, the linear regression model training algorithm in Section 4.1 is not applicable. Thus, we modify it to accommodate the input changes, i.e., $[Y]$ and $\langle \boldsymbol{w} \rangle$. For $[Y]$, we can directly incorporate it when calculating the encrypted residual using the global aggregation $[A]$ based on homomorphic properties. For $\langle \boldsymbol{w} \rangle$, we use the $\mathsf{MatShareMul}$ operator to calculate the multiplication between $\langle w_i \rangle$ and $[R_j] \otimes \boldsymbol{x}_j$ in Eqn 4. The rest computations are the same, and the trained model parameters are the explanation.

## 5.3 Discussion

Similar to model training in Section 4, Falcon-INP ensures that only the final explanation is disclosed as the intermediate information computed on the parties' data is in secure formats (i.e., PHE or SSS). Note that the feature selection stage reveals the selected feature IDs, which are part of the final explanation. Therefore, the privacy guarantee in Section 5.1 is satisfied.

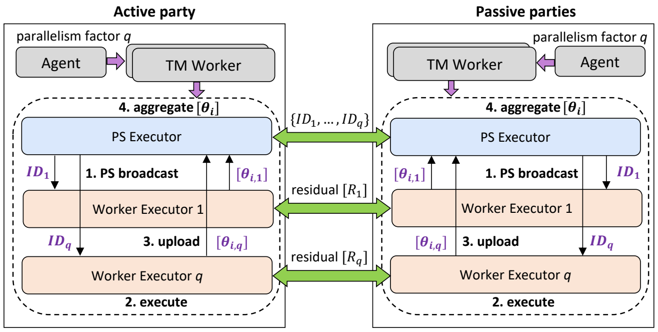


Figure 6: The intra-party PS architecture in Falcon.

## 6 OPTIMIZATIONS

### 6.1 Data Parallelism

We employ the parameter server (PS) architecture [37] within each party for data parallelism. However, we need to ensure that the PS and workers on each party match with the corresponding PS and workers on other parties during the execution. This is because the parties conduct a number of SSS operations, which require the exchanged information to be consistent with their private shares. Figure 6 shows the PS architecture in Falcon. Given a task, each party $P_i (i \in [1, m])$ creates a PS and a set of workers, each of which runs a Falcon executor (see Section 3.2) that executes the task. Specifically, the coordinator determines a parallelism factor $q$ for a task, i.e., the number of workers to be created. It then informs the agent on each party to start a PS and $q$ workers. Let $PS_i$ and $W_{i,j}$ denote the PS and the $j$-th worker of party $P_i$, respectively, where $i \in [1, m]$ and $j \in [1, q]$. In addition to the inter-party peer-to-peer (P2P) communication among PSs (i.e., $\{PS_1, \cdots, PS_m\}$) and among workers (i.e., $\{W_{1,j}, \cdots, W_{m,j}\}$) from different parties, there are intra-party communication between $PS_i$ and each worker $W_{i,j} (j \in [1, q])$.

Take the distributed linear model training as an example. In each iteration, the active party's $PS_1$ selects a batch and partitions the instance IDs into $q$ subsets, say $\mathrm{ID} = \{\mathrm{ID}_1, \cdots, \mathrm{ID}_q\}$. Next, $PS_1$ broadcasts them to the passive parties' PSs, and each $PS_i (i \in [1, m])$ distributes a subset $\mathrm{ID}_j$ to each worker $W_{i,j}$. This ensures that each group of workers across multiple parties uses the same instances. Then, each group of workers securely executes the forward and backward computations, and obtains the updated model parameters $[\boldsymbol{\theta}_{i,j}]$. Afterward, each worker uploads $[\boldsymbol{\theta}_{i,j}]$ to $PS_i$ for aggregation. Finally, $PS_i$ obtains $[\boldsymbol{\theta}_i]$ and broadcasts it to its workers. The distributed workflow of the stages in Falcon-INP is similar, except for WPCC-based feature selection, which needs feature-level partition as each WPCC calculation requires all sampled instances.

### 6.2 Auto-Parallelism in Falcon-INP

Now we introduce an auto-parallelism method that finds effective parallelism factors for the stages in Falcon-INP to reduce the overall computation time under resource constraints.

Suppose we can create at most $Q$ containers (including PS and workers) for a submitted job. We note that the original model prediction and instance weighing stages are independent, and therefore can be executed in parallel. Similarly, when there are multiple

**Table 1: Evaluation of training accuracy**

| Dataset | Energy | Bike | Bank | Credit | News | Connect |
|---|---|---|---|---|---|---|
| Falcon-LR | 0.0869 | 0.0363 | 90.49% | 78.56% | - | - |
| Falcon-MLP | 0.0845 | 0.0355 | 90.49% | 78.57% | 42.28% | 71.08% |
| NP-LR | 0.0869 | 0.0341 | 90.49% | 78.56% | - | - |
| NP-MLP | 0.0844 | 0.0358 | 90.49% | 78.57% | 42.30% | 71.14% |

**Table 2: Evaluation of Falcon-INP explanation results**

| Dataset | Energy | Bike | Bank | Credit | News | Connect |
|---|---|---|---|---|---|---|
| Weighing | 6.4e-10 | 3.1e-09 | 5.2e-10 | 9.5e-09 | 1.7e-07 | 3.6e-09 |
| Feature sel. | 2.2e-07 | 2.9e-07 | 2.1e-07 | 1.5e-07 | 9.1e-08 | 1.9e-09 |
| Falcon (LR-L1) | 7.1e-05 | 3.1e-05 | 3.7e-05 | 4.7e-05 | 5.9e-04 | 5.2e-05 |
| Falcon (LR-L2) | 1.5e-09 | 5.2e-10 | 5.6e-09 | 4.4e-09 | 1.8e-09 | 2.1e-09 |

classes to be explained, their computation can be done in parallel (see Figure 5). Also, at least one worker needs to be allocated to each stage and each label. Let $\boldsymbol{t} = (t_1, t_2, t_3, t_4, t_5)$ be the execution time of the five stages without parallelism, and $\boldsymbol{q} = (q_1, q_2, q_3, q_4, q_5)$ be the parallelism factors of the corresponding stages, respectively. We set $q_1 = 1$ by default. We find that the speedup cannot increase linearly with the number of workers due to communication overhead under data parallelism; thus, we estimate the decay of each stage with separated functions. Let $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5)$ be the decay functions, where $\sigma_i (i \in [1, 5])$ takes $q_i$ as input and outputs an inverse of the speedup ratio. Since the cryptographic computations are relatively stable, we can estimate the time $\boldsymbol{t}$ and fit the decay functions by profiling operations (e.g., execute dummy computations). The profiling is invoked before the interpretability calculation service is enabled for handling new prediction requests. Given a trained original VFL model $f$, the profiling only needs to be executed once if the number of sampled instances $n$, the number of explained features $d_{\text{sel}}$, and the batch size $b$ for interpretable model training are fixed. Finally, we define the objective function w.r.t. execution time as follows.

$$\begin{aligned}
\min_{\boldsymbol{q}} \quad & T = t_1 + \max\{t_2\sigma_2(q_2), t_3\sigma_3(q_3)\} + t_4\sigma_4(q_4) + t_5\sigma_5(q_5) \\
\text{s.t.} \quad & 1 + \textstyle\sum_{i=2}^{3}(q_i + \alpha(q_i)) + \sum_{i=4}^{5} c(q_i + \alpha(q_i)) \le Q, \qquad (10) \\
& \forall q_i \in \boldsymbol{q}, \ \ \alpha(q_i) = 0 \text{ if } q_i = 1, \ \alpha(q_i) = 1 \text{ otherwise}, \\
& \forall q_i \in \boldsymbol{q}, 1 \le q_i \le Q.
\end{aligned}$$

The $\alpha(\cdot)$ function checks whether a parallelism factor $q_i$ is 1. If so, no PS will be created. Eqn 10 is a mixed integer nonlinear programming problem; we use Scipy-Optimize [65] to solve it.

# 7 PERFORMANCE EVALUATION

## 7.1 Methodology

**Implementation.** We implemented Falcon in 27K lines of code (LoC). It consists of 9K LoC in Golang for the coordinator and agent, 17K LoC in C++ for the executor, and 1K LoC in MP-SPDZ for SSS programs. We use libhcs [41] for the PHE primitives and MP-SPDZ [20] for SSS programs.

**Experimental setup.** We conduct experiments on Amazon EC2 [4] using c5.4xlarge instances. The instances are created in a local area network (LAN) cluster in Northern Virginia. Each machine has 16 vCPUs and 32GB memory, running Ubuntu 20.04 LTS. We set the *keysize* of PHE as 2048 bits and the default security parameter of SSS programs as 128 bits. We use 16 bits to encode the floating-point values by default. Besides, the default number of parties is 3 unless specified otherwise.

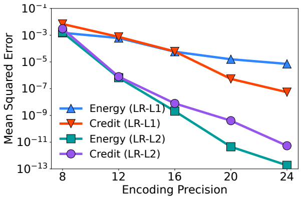


**Figure 7: Effect of precision on Falcon-INP's accuracy.**

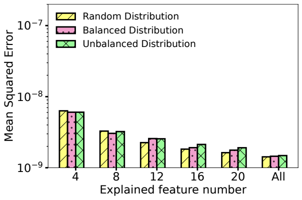


**Figure 8: Effect of the feature distribution (Energy).**

**Datasets.** We use six real-world datasets: (i) Energy dataset, with 19735 samples and 29 features for regression; (ii) Bike dataset, with 17389 samples and 14 features for regression; (iii) Bank dataset, with 4521 samples and 17 features with two classes; (iv) Credit Card dataset, with 30000 samples and 23 features with two classes; (v) News dataset, with 39797 samples and 59 features with five classes; and (vi) Connect-4 dataset, with 67557 samples and 42 features with three classes. In addition, we generate synthetic datasets by varying the number of parties ($m$), the number of features per party ($d'$), the number of samples ($n$), and the number of classes ($c$). We normalize the data into [0,1] to transform features into the same scale.

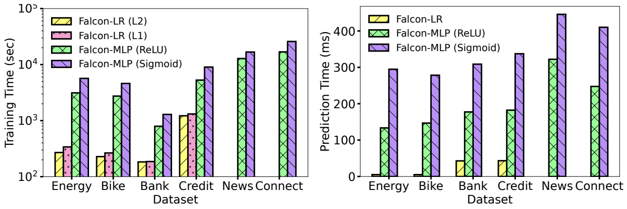
**Baselines.** We compare Falcon with four sets of baselines:

- **Non-private (NP)** algorithm is used to evaluate the accuracy of the trained model and explanation and the efficiency. We implement it based on the same logic of Falcon's secure algorithms.
- **MP-SPDZ** [20] is an MPC library that supports various secure protocols. We use its SSS protocol to implement training and interpretability calculation for efficiency evaluation.
- **FATE** [71] is an industrial-grade federated learning system developed by WeBank. We adopt its logistic regression model training algorithm for efficiency comparison.
- **SecureML** [54] is a secure ML system based on secure two-party computation. Since the system is not open-sourced, we implement its secure logistic regression algorithm for comparison.
- **Average-Parallelism** is a baseline method for generating parallelism factors. It assigns one container to the instance sampling stage and equally splits the remaining containers to other stages.

**Metrics.** For model accuracy, we use prediction accuracy for classification tasks and mean square error (MSE) for regression tasks. To evaluate Falcon-INP's effectiveness, we measure the MSE of each stage between the results computed by Falcon-INP and NP algorithms. For efficiency, we measure the total running time. The training time reported is for one epoch unless noted otherwise.

## 7.2 Accuracy

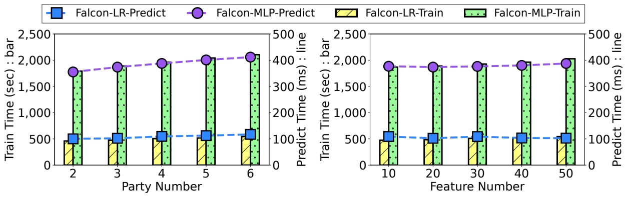
**Model accuracy.** We train the LR (linear regression and logistic regression for regression and classification tasks, respectively) and MLP models on the six datasets, and compare their performance with those trained with NP algorithms. Table 1 reports the comparison results. Since we use cryptographic primitives (i.e., PHE and SSS) to build our training algorithms, the performance is comparable to the NP algorithms. There is a slight loss of accuracy due to the precision loss caused by fixed-point representation. Note that the MLP model accuracy on News is low because this classification task is relatively complex.



(a) **Training time**

(b) **Prediction time**

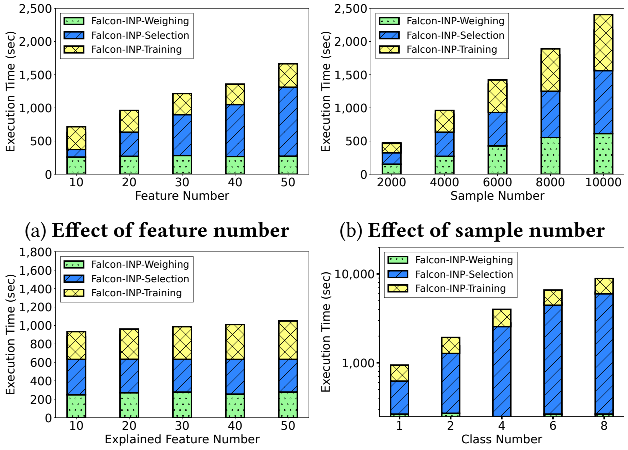
**Figure 9: The efficiency evaluation on real-world datasets.**



(a) **Effect of party number**

(b) **Effect of feature number**

**Figure 10: Training and prediction time w.r.t. parameters.**



(a) **Effect of feature number**

(b) **Effect of sample number**

(c) **Effect of explained number**

(d) **Effect of class number**

**Figure 11: Falcon-INP efficiency evaluation.**

**Interpretability accuracy.** We use the VFL model trained in the model training stage as $f$ in Eqn 2 to compute predictions. We employ linear regression with L1/L2 regularization (LR-L1/LR-L2) for training the interpretable models. Table 2 summarizes the MSEs of each stage's outputs between Falcon-INP and NP. The MSEs of all the stages are small, demonstrating that the privacy-preserving calculations are accurate. Note that the MSE of LR with L2 regularization is smaller than that of LR with L1 regularization. The reason is that, unlike L1 regularization which requires parties to convert each encrypted parameter $[\boldsymbol{\theta}_i]$ into secret shares for calculating the sign in each iteration, L2 regularization allows the parties to directly calculate the encrypted gradient (i.e., $2\lambda \odot [\boldsymbol{\theta}_i]$) and update the encrypted parameter. Conversion is only required when the precision reaches the maximum allowed by PHE. Thus, the precision loss of L1 regularization has a larger impact on the trained parameters, particularly when the parameters tend to be close to zero, as the sign may be mistakenly calculated, e.g., leading the regularized gradient to be $[\lambda]$ while the actual gradient is $[-\lambda]$.

To further investigate the impact of fixed-point encoding precision on interpretability accuracy, we conduct experiments with different precisions $\{8, 12, \mathbf{16}, 20, 24\}$. Figure 7 presents the MSEs of the interpretable models (with LR-L1 and LR-L2) between Falcon-INP and NP on Energy and Credit. We see that the MSEs of both LR-L1 and LR-L2 decrease as the precision level increases. This is expected because, with higher encoding precision, the effect of the truncation loss is reduced, resulting in higher accuracy.

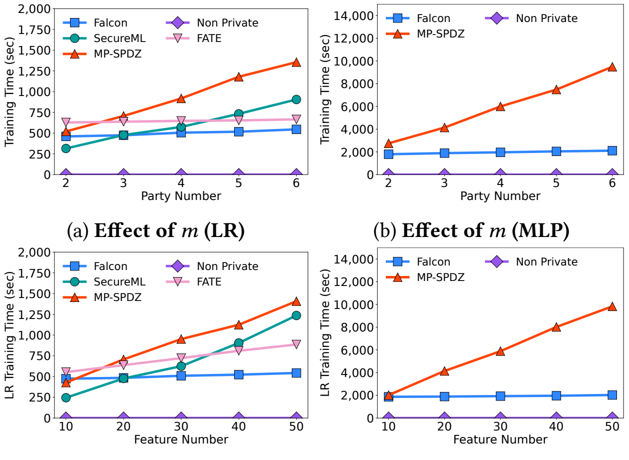
We also examine how the distribution of important features among parties affects interpretability accuracy. We use three partition methods to split the important features: random, balanced (the important features are evenly distributed among parties), and unbalanced (e.g., party 1 holds the most important features). Figure 8 compares the results of Falcon-INP and NP w.r.t. LR-L2, with varying numbers of explained features $d_{\text{sel}}$ on the Energy dataset. The MSEs of the three methods are similar (around 1e-8). The rationale is that the feature distribution has little effect on the aggregation result and the model parameters. Additionally, the MSEs slightly decrease as $d_{\text{sel}}$ increases, as including more features with smaller parameter values leads to lower MSEs of all explained features.

## 7.3 Efficiency

*7.3.1 Model Training and Prediction.* We first evaluate the model training and prediction efficiency of Falcon in the centralized mode. **Evaluation on real-world datasets.** Figure 9a reports the training time of Falcon-LR and Falcon-MLP. For the Energy and Bike datasets, Falcon-LR indicates linear regression, while for others, it indicates logistic regression. For Falcon-MLP models, we use one hidden layer with eight neurons.

There are three observations. First, Falcon-MLP is much slower than Falcon-LR as the MLP model consists of more neurons and therefore requires more computations. Second, Falcon-LR with L2 regularization is slightly faster than Falcon-LR with L1 regularization. The reason is that Falcon-LR (L1) relies on the SSS program to check whether a parameter is larger than 0, incurring additional conversions and computations, whereas Falcon-LR (L2) training does not need this step. Third, training Falcon-MLP with ReLU is faster than that with Sigmoid because the secure computation of the Sigmoid function (requiring exponential calculations) is more complex than the ReLU function (requiring simple comparison).

Figure 9b summarizes the prediction time per sample of the trained models. We see that the prediction time of linear regression models is the fastest as each party only needs to compute a local dot product operation with PHE, and the parties aggregate the ciphertexts. For example, it takes about 5.1$ms$ to predict a sample on Bike. In comparison, the logistic regression model prediction requires PHE2SSS conversions and calculates the logistic function using secure operations, incurring a longer prediction time (e.g., around 43.3$ms$ on Credit). Also, Falcon-MLP models take much longer to compute the prediction, and Falcon-MLP (ReLU) is relatively faster than Falcon-MLP (Sigmoid) due to the reasons explained above.



(a) **Effect of** $m$ **(LR)**

(b) **Effect of** $m$ **(MLP)**

(c) **Effect of** $d'$ **(LR)**

(d) **Effect of** $d'$ **(MLP)**

**Figure 12: The training time comparison with the baseline.**

**Evaluation on synthetic datasets.** We further investigate the effects of the party number $m$ and the feature number per party $d'$ on the efficiency. We generate a set of synthetic datasets with two classes by varying the number of participating parties $m \in \{2, \mathbf{3}, 4, 5, 6\}$ and the number of features each party holds $d' \in \{10, \mathbf{20}, 30, 40, 50\}$. Figure 10a shows that the training and prediction time increases slightly when $m$ grows for both models as it requires more communications for SSS executions. Nevertheless, the increased time is insignificant, indicating that Falcon can scale well with more parties. Similarly, Figure 10b evidences that the time is stable when $d'$ increases. This is because $d'$ only affects each party's local aggregation and encrypted parameter update, which are less dominant operations in the overall execution time.

*7.3.2 Interpretability Computation.* Next, we evaluate Falcon-INP's efficiency in the centralized mode. We generate synthetic datasets by varying the number of features per party $d' \in \{10, \mathbf{20}, 30, 40, 50\}$, the number of sampled instances $n \in \{2000, \mathbf{4000}, 6000, 8000, 10000\}$, the number of explained features $d_{\text{sel}} \in \{10, \mathbf{20}, 30, 40, 50\}$, and the number of classes $c \in \{\mathbf{1}, 2, 4, 6, 8\}$. We use these datasets as sampled instances $S$ and report the execution time of instance weighing, feature selection, and interpretable training with ten epochs.

**Effect of feature number.** Figure 11a shows the execution time for varying $d'$. The execution time increases linearly with $d'$. We observe that the time of instance weighing and interpretable model training stages are stable when $d'$ increases. This is because the most time-consuming part in instance weighing is the distance and kernel calculation using SSS, which is on the aggregated statistics. Meanwhile, the training time is roughly the same as they only train on $d_{\text{sel}}$ features. The feature selection time is dominant as $d'$ grows since it needs to compute WPCC for more features, and each WPCC calculation involves all instances, consuming more time.

**Effect of sample number.** Figure 11b shows the execution time for varying $n$, which goes up linearly when $n$ increases. Specifically, we can see that the three stages have a similar trend when $n$ grows, because they need to handle more sampled instances for calculating weights, WPCC, and iterative model training.

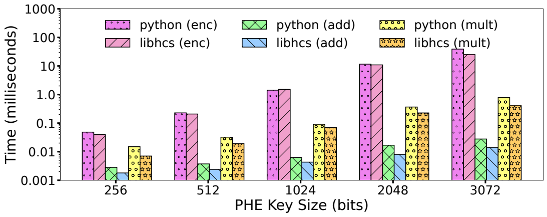
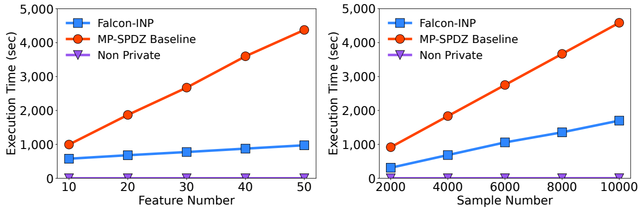


**Figure 13: Comparison of python-paillier and libhcs-pailler.**



(a) **Effect of feature number**

(b) **Effect of sample number**

**Figure 14: The time comparison of with the baseline.**

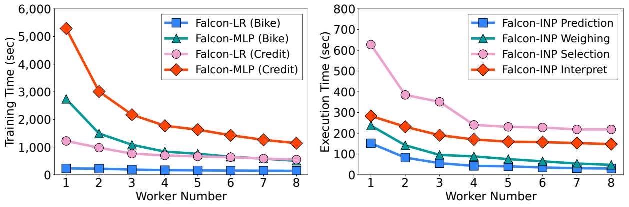
**Effect of explained feature number.** Figure 11c shows the execution time for varying $d_{\text{sel}}$. Note that $d_{\text{sel}}$ only affects the interpretable model training stage, and it is a minor factor in the LR model training, as discussed in Section 7.3.1. Thus, the execution time only increases slightly when $d_{\text{sel}}$ increases.

**Effect of class number.** Figure 11d shows the execution time for varying $c$. We observe that the execution time increases significantly with $c$, because we need to execute the feature selection and interpretable model training stages for each class label. In the centralized mode, we execute the stages for different classes sequentially; therefore, the execution time doubles when $c$ doubles.

*7.3.3 Baseline Comparison.* We compare Falcon with four baselines: MP-SPDZ [36], FATE [71], SecureML [54], and NP. We do not compare to SplitNN-based solutions (PySyft and FATE MLP), as they disclose too much intermediate information during training. Similarly, we do not compare to PaddleFL, which uses ABY3, as it assumes an honest majority [53] setting (i.e., two of the three servers are honest), relaxing the security to achieve performance.

**Model training.** We use synthetic datasets with the number of parties $m \in \{2, \mathbf{3}, 4, 5, 6\}$ and the number of features per party $d' \in \{10, \mathbf{20}, 30, 40, 50\}$. We set the number of instances $n = 10000$.

Figures 12a and 12b show the time comparison for varying $m$ w.r.t. LR and MLP models, respectively. There are four main observations. First, NP algorithms are efficient (less than 3 seconds) in all experiments, but it sacrifices data privacy. Second, when $m = 2$, Falcon's execution time is comparable to those of SecureML and MP-SPDZ, with SecureML being slightly faster for the LR model. This is because two-party MPC is relatively efficient, and SecureML uses a lightweight piecewise function approximation of the logistic function, while PHE operations in Falcon are time-intensive. Third, as $m$ increases, MP-SPDZ and SecureML experience significant time increases compared to Falcon. This is expected because Falcon executes many computations locally, while MP-SPDZ and SecureML require more communication rounds, resulting in lower



(a) **Falcon training**

(b) **Falcon-INP stages**

**Figure 15: The execution time w.r.t. worker number.**

**Table 3: Parallelism strategy comparison**

| Q | 10 | 15 | 20 | 25 | 30 | 35 | 40 |
|---|---|---|---|---|---|---|---|
| Average | 1162.7 | 1003.7 | 736.3 | 594.2 | 565.9 | 505.5 | 473.4 |
| Auto | 1073.6 | 816.1 | 673.1 | 563.1 | 512.1 | 472.4 | 445.3 |

performance. Falcon achieves up to 2.5x and 1.7x speedup over MP-SPDZ and SecureML for the LR model, and up to 4.5x speedup over MP-SPDZ for the MLP model. Fourth, Falcon outperforms FATE-LR, which also uses PHE for privacy, by up to 1.4x, due to its use of the C++-implemented libhcs library for Paillier instead of the Python-based Paillier library used by FATE. Specifically, a micro-benchmark (see Figure 13) comparing the two libraries for encryption, homomorphic addition, and homomorphic multiplication shows that libhcs-Paillier is up to 1.56x, 2.09x, and 2.12x faster than Python-Paillier for the three operators. Moreover, Falcon provides stronger privacy protection, while FATE allows parties to decrypt some information for easier computations.

Figures 12c and 12d show the comparison for varying $d'$. Falcon is stable as $d'$ only affects each party's local computations, a minor factor in execution time. Specifically, Falcon achieves up to 2.7x, 2.4x, and 1.7x speedup over MP-SPDZ, SecureML, and FATE for LR, and up to 4.9x speedup over MP-SPDZ for MLP.

**Interpretability calculation comparison.** We compare Falcon-INP with the MP-SPDZ and NP baselines. We consider the computation for one class label, and measure the time of the instance weighing and interpretable model training stages. Figure 14 shows the comparison by varying $d'$ and $n$. Similarly, NP's time is very small due to no privacy protection. The execution time of Falcon-INP slightly increases as $d'$ increases, while that of MP-SPDZ increases significantly, which is similar to the comparison in Section 7.3.1. Besides, when $n$ increases, both Falcon-INP and MP-SPDZ have longer execution times, since each stage handles more sampled instances. In all cases, Falcon-INP is more efficient and is up to 4.49x and 2.95x compared to MP-SPDZ for varying $d'$ and $n$, respectively.

*7.3.4 Distributed Data Parallelism.* Finally, we evaluate the efficiency of our optimizations in Section 6. Figure 15a-15b present the execution time w.r.t. the different number of workers $q \in [1, 8]$ for Falcon training and Falcon-INP stages, respectively. When $q = 1$, the execution is the same as in the centralized mode. In Figure 15a, we observe that Falcon-MLP results in a higher speedup than Falcon-LR on both Bike and Credit. The reason is that Falcon-LR is already fast, and there are communication and computation overheads in the distributed mode, which dominate the execution time, leading to a lower speedup. For Falcon-INP stages in Figure 15b, we can see that the improvements for the original model prediction and instance weighing stages are more pronounced. For example, when $q = 8$, the speedup of these two stages are 5.055x and 4.973x, while the feature selection and model training stages are 2.872x and 1.921x, respectively. Similarly, this is due to the overheads introduced.

Table 3 compares the auto-parallelism method with the average-parallelism baseline w.r.t. the different number of total containers $Q \in \{10, 15, 20, 25, 30, 35, 40\}$. The time reported is the estimation of the total execution time of all five stages. We use the logistic regression model for prediction. Given $n = 4000$, $d_{\text{sel}} = 20$, and $b = 512$, it takes around $3925s$ for profiling the five stages, and the result is used for this set of experiments. We see that the execution time gradually decreases as $Q$ increases, because given more resources, we can benefit more from the parallel execution. Further, auto-parallelism results in a faster execution time, indicating that it finds more effective parallelism factors for scheduling.

## 8 RELATED WORK

Existing VFL systems, including SecureML [54], FATE [71], PaddleFL [5], and PySyft [59], share a similar goal with Falcon, that is, to provide privacy for model training and prediction. However, they either disclose intermediate information during training (e.g., FATE and PySyft) or assume a weaker security model (e.g., SecureML and PaddleFL require the parties to outsource private data to non-colluding servers). Furthermore, none of existing systems considers interpretability, which is important and challenging in VFL.

Model-agnostic interpretable methods [13, 21] require black-box access to the target models and interpret model predictions by analyzing the difference in model outputs w.r.t. different inputs. State-of-the-art methods, such as LIME [64, 78] and SHAP [45, 67] are not designed for security and cannot be applied directly to VFL. We propose a privacy-preserving framework Falcon-INP that addresses this challenge. Besides, [44] considers model debugging when a VFL model produces unexpected predictions. However, it does not provide interpretability to the prediction.

## 9 CONCLUSIONS

In this paper, we presented a privacy-preserving and interpretable vertical federated learning system Falcon, which allows multiple parties to train VFL models and make interpretable predictions in a privacy-preserving manner. To the best of our knowledge, Falcon is the first system that supports prediction interpretability in VFL. The system consists of a series of novel algorithms based on partial homomorphic encryption and secret sharing schemes. It introduces system-level optimizations that improve performance by supporting parallel execution of the model training and interpretability. We fully implement Falcon and evaluate against both real-world and synthetic datasets. The results show that Falcon achieves high accuracy, and it outperforms secure baselines in terms of efficiency.

## ACKNOWLEDGMENTS

We thank Kaiyuan Yang, Xinjian Luo, and Can Cui for their early contributions. This work is supported by Singapore Ministry of Education Academic Research Fund Tier 3 under MOEs official grant number MOE2017-T3-1-007. The work of Meihui Zhang is supported by National Natural Science Foundation of China (62072033).

## REFERENCES

[1] 2016. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation). (2016).

[2] 2018. California Consumer Privacy Act. Bill No. 375 privacy: personal information: businesses. https://leginfo.legislature.ca.gov/. (2018).

[3] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *CCS*. 308–318.

[4] Amazon. 2022. Amazon Elastic Computing Cloud (Amazon EC2). https://www.amazonaws.cn/en/ec2/. Accessed: 2022-12.

[5] Baidu. 2023. PaddleFL: Federated Deep Learning in PaddlePaddle, https://github.com/PaddlePaddle/PaddleFL. Accessed: 2023-04.

[6] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*.

[7] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag.

[8] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *CCS*. 1175–1191.

[9] Niklas Bussmann, Paolo Giudici, Dimitri Marinelli, and Jochen Papenbrock. 2021. Explainable machine learning in credit risk management. *Computational Economics* 57, 1 (2021), 203–216.

[10] Octavian Catrina and Sebastiaan de Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *SCN*. 182–199.

[11] Octavian Catrina and Amitabh Saxena. 2010. Secure Computation with Fixed-Point Numbers. In *FC*. 35–50.

[12] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *CCS*. 1243–1255.

[13] Jianbo Chen, Le Song, Martin Wainwright, and Michael Jordan. 2018. Learning to explain: An information-theoretic perspective on model interpretation. In *ICML*. PMLR, 883–892.

[14] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, and Qiang Yang. 2019. SecureBoost: A Lossless Federated Learning Framework. *CoRR* abs/1901.08755 (2019).

[15] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. 2020. Crypt?: Crypto-Assisted Differential Privacy on Untrusted Servers. In *SIGMOD*. 603–619.

[16] Martine De Cock, Rafael Dowsley, Caleb Horst, Raj S. Katti, Anderson C. A. Nascimento, Wing-Sea Poon, and Stacey Truex. 2019. Efficient and Private Scoring of Decision Trees, Support Vector Machines and Logistic Regression Models Based on Pre-Computation. *IEEE TDSC* 16, 2 (2019), 217–230.

[17] Jian Dai, Meihui Zhang, Gang Chen, Ju Fan, Kee Yuan Ngiam, and Beng Chin Ooi. 2018. Fine-grained concept linking using neural networks in healthcare. In *SIGMOD*. 51–66.

[18] Ivan Damgård and Mads Jurik. 2001. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography*. 119–136.

[19] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*. 643–662.

[20] CSIRO Data61. 2021. data61/MP-SPDZ: https://github.com/data61/MP-SPDZ. Accessed: 2021-08-26.

[21] Anupam Datta, Shayak Sen, and Yair Zick. 2016. Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems. In *IEEE S&P*. 598–617.

[22] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*.

[23] David Froelicher, Juan R Troncoso-Pastoriza, Jean Louis Raisaro, Michel A Cuendet, Joao Sa Sousa, Hyunghoon Cho, Bonnie Berger, Jacques Fellay, and Jean-Pierre Hubaux. 2021. Truly privacy-preserving federated analytics for precision medicine with multiparty homomorphic encryption. *Nature communications* 12, 1 (2021), 1–10.

[24] Chong Fu, Xuhong Zhang, Shouling Ji, Jinyin Chen, Jingzheng Wu, Shanqing Guo, Jun Zhou, Alex X. Liu, and Ting Wang. 2022. Label Inference Attacks Against Vertical Federated Learning. In *USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 1397–1414.

[25] Fangcheng Fu, Yingxia Shao, Lele Yu, Jiawei Jiang, Huanran Xue, Yangyu Tao, and Bin Cui. 2021. VF$^2$Boost: Very Fast Vertical Federated Gradient Boosting for Cross-Enterprise Learning. In *SIGMOD*. 563–576.

[26] Fangcheng Fu, Huanran Xue, Yong Cheng, Yangyu Tao, and Bin Cui. 2022. BlindFL: Vertical Federated Machine Learning without Peeking into Your Data. In *SIGMOD*. 1316–1330.

[27] Rui Fu, Yuncheng Wu, Quanqing Xu, and Meihui Zhang. 2023. FEAST: A Communication-efficient Federated Feature Selection Framework for Relational Data. *Proc. ACM Manag. Data* 1, 1 (2023), 107:1–107:28.

[28] Chang Ge, Ihab F. Ilyas, and Florian Kerschbaum. 2019. Secure Multi-Party Functional Dependency Discovery. *PVLDB* 13, 2 (2019), 184–196.

[29] Robin C. Geyer, Tassilo Klein, and Moin Nabi. 2017. Differentially Private Federated Learning: A Client Level Perspective. *CoRR* abs/1712.07557 (2017).

[30] Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Richard Nock, Giorgio Patrini, Guillaume Smith, and Brian Thorne. 2017. Private federated learning on vertically partitioned data via entity resolution and additively homomorphic encryption. *arXiv preprint arXiv:1711.10677* (2017).

[31] John F. Hart. 1978. *Computer Approximations*. Krieger Publishing Co., Inc., Melbourne, FL, USA.

[32] Kai He, Liu Yang, Jue Hong, Jinghua Jiang, Jieming Wu, Xu Dong, and Zhuxun Liang. 2019. PrivC - A Framework for Efficient Secure Two-Party Computation. In *SecureComm*, Vol. 305. Springer, 394–407.

[33] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. 2017. Composing Differential Privacy and Secure Computation: A Case Study on Scaling Private Record Linkage. In *CCS*. 1389–1406.

[34] Yaochen Hu, Di Niu, Jianming Yang, and Shengping Zhou. 2019. FDML: A Collaborative Machine Learning Framework for Distributed Features. In *SIGKDD*. 2232–2240.

[35] Ziyue Huang, Yuan Qiu, Ke Yi, and Graham Cormode. 2022. Frequency Estimation Under Multiparty Differential Privacy: One-shot and Streaming. *Proc. VLDB Endow.* 15, 10 (2022), 2058–2070.

[36] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS*. 1575–1590.

[37] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. OSDI. 583–598.

[38] Oscar Li, Jiankai Sun, Xin Yang, Weihao Gao, Hongyi Zhang, Junyuan Xie, Virginia Smith, and Chong Wang. 2022. Label Leakage and Protection in Two-party Split Learning. In *ICLR*.

[39] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. 2020. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine* 37, 3 (2020), 50–60.

[40] Zitao Li, Bolin Ding, Ce Zhang, Ninghui Li, and Jingren Zhou. 2021. Federated Matrix Factorization with Privacy Guarantee. *PVLDB* 15, 4 (2021), 900–913.

[41] Libhcs. 2022. A partially Homomorphic C library. https://github.com/tiehuis/libhcs. Accessed: 2022-12.

[42] Junxu Liu, Jian Lou, Li Xiong, Jinfei Liu, and Xiaofeng Meng. 2021. Projected Federated Averaging with Heterogeneous Differential Privacy. *PVLDB* 15, 4 (2021), 828–840.

[43] Yang Liu, Yingting Liu, Zhijie Liu, Junbo Zhang, Chuishi Meng, and Yu Zheng. 2019. Federated Forest. *CoRR* abs/1905.10053 (2019).

[44] Yejia Liu, Weiyuan Wu, Lampros Flokas, Jiannan Wang, and Eugene Wu. 2021. Enabling SQL-based Training Data Debugging for Federated Learning. *PVLDB* 15, 3 (2021), 388–400.

[45] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *NIPS*. 4765–4774.

[46] Xinjian Luo, Yuncheng Wu, Xiaokui Xiao, and Beng Chin Ooi. 2021. Feature Inference Attack on Model Predictions in Vertical Federated Learning. In *ICDE*. IEEE, 181–192.

[47] Zhaojing Luo, Shaofeng Cai, Jinyang Gao, Meihui Zhang, Kee Yuan Ngiam, Gang Chen, and Wang-Chien Lee. 2018. Adaptive lightweight regularization tool for complex analytics. In *ICDE*. 485–496.

[48] Zhaojing Luo, Shaofeng Cai, Yatong Wang, and Beng Chin Ooi. 2023. Regularized Pairwise Relationship Based Analytics for Structured Data. *Proc. ACM Manag. Data*, Article 82 (2023), 27 pages.

[49] Zhaojing Luo, Sai Ho Yeung, Meihui Zhang, Kaiping Zheng, Lei Zhu, Gang Chen, Feiyi Fan, Qian Lin, Kee Yuan Ngiam, and Beng Chin Ooi. 2021. MLCask: Efficient Management of Component Evolution in Collaborative Data Analytics Pipelines. In *ICDE*. 1655–1666.

[50] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*. 1273–1282.

[51] Catherine A. Meadows. 1986. A More Efficient Cryptographic Matchmaking Protocol for Use in the Absence of a Continuously Available Third Party. In *IEEE S&P*. 134–137.

[52] Olga Mierzwa-Sulima. 2019. "Please, explain." Interpretability of black-box machine learning models, https://appsilon.com/please-explain-black-box/. Accessed: 2023-06.

[53] Payman Mohassel and Peter Rindal. 2018. ABY$^3$: A Mixed Protocol Framework for Machine Learning. In *CCS*. ACM, 35–52.

[54] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*. 19–38.

[55] Christoph Molnar. 2019. *Interpretable Machine Learning*. https://christophm.github.io/interpretable-ml-book/. Accessed: 2022-11.

[56] Richard Nock, Stephen Hardy, Wilko Henecka, Hamish Ivey-Law, Giorgio Patrini, Guillaume Smith, and Brian Thorne. 2018. Entity Resolution and Federated

Learning get a Federated Resolution. *CoRR* abs/1803.04035 (2018).

[57] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium.* 619–636.

[58] Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony K. H. Tung, Yuan Wang, Zhongle Xie, Meihui Zhang, and Kaiping Zheng. 2015. SINGA: A Distributed Deep Learning Platform. In *ACM MM.* 685–688.

[59] OpenMined. 2023. PySyft: A library for answering questions using data you cannot see, https://github.com/OpenMined/PySyft. Accessed: 2023-04.

[60] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT.* 223–238.

[61] Dario Pasquini, Giuseppe Ateniese, and Massimo Bernaschi. 2021. Unleashing the tiger: Inference attacks on split learning. In *CCS.* 2113–2129.

[62] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. 2015. Phasing: Private Set Intersection Using Permutation-based Hashing. In *USENIX Security Symposium.* 515–530.

[63] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2018. Scalable Private Set Intersection Based on OT Extension. *ACM Trans. Priv. Secur.* 21, 2 (2018), 7:1–7:35.

[64] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *SIGKDD.* 1135–1144.

[65] SciPy. 2022. Fundamental algorithms for scientific computing in Python. https://docs.scipy.org/doc/scipy/reference/optimize.html. Accessed: 2022-12.

[66] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ASPLOS.* 955–970.

[67] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *ICML.* 3145–3153.

[68] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. 2019. A Hybrid Approach to Privacy-Preserving Federated Learning. In *AISec@CCS.* ACM, 1–11.

[69] Jaideep Vaidya, Basit Shafiq, Wei Fan, Danish Mehmood, and David Lorenzi. 2014. A Random Decision Tree Framework for Privacy-Preserving Data Mining. *IEEE TDSC* 11, 5 (2014), 399–411.

[70] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. 2018. Split learning for health: Distributed deep learning without sharing raw patient data. *CoRR* abs/1812.00564 (2018).

[71] WeBank. 2019. FATE: Federated AI Technology Enabler, https://github.com/FederatedAI/FATE. Accessed: 2022-12.

[72] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. 2016. Privacy-Preserving Shortest Path Computation. In *NDSS.*

[73] Yuncheng Wu, Shaofeng Cai, Xiaokui Xiao, Gang Chen, and Beng Chin Ooi. 2020. Privacy Preserving Vertical Federated Learning for Tree-based Models. *PVLDB* 13, 11 (2020), 2090–2103.

[74] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE S&P.* 640–656.

[75] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *ACM TIST* 10, 2 (2019), 12:1–12:19.

[76] Kaiping Zheng, Shaofeng Cai, Horng Ruey Chua, Wei Wang, Kee Yuan Ngiam, and Beng Chin Ooi. 2020. TRACER: A Framework for Facilitating Accurate and Interpretable Analytics for High Stakes Applications. In *SIGMOD.* 1747–1763.

[77] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2019. Helen: Maliciously Secure Coopetitive Learning for Linear Models. In *IEEE S&P.* 915–929.

[78] Zhengze Zhou, Giles Hooker, and Fei Wang. 2021. S-lime: Stabilized-lime for model explanation. In *SIGKDD.* 2429–2438.