POSEIDON: Privacy-Preserving Federated Neural Network Learning

Sinem Sav, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza,
David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux
Laboratory for Data Security, EPFL
e-mail: name.surname@epfl.ch

Abstract—**In this paper, we address the problem of privacy-preserving training and evaluation of neural networks in an N-party, federated learning setting. We propose a novel system, POSEIDON, the first of its kind in the regime of privacy-preserving neural network training. It employs multiparty lattice-based cryptography to preserve the confidentiality of the training data, the model, and the evaluation data, under a passive-adversary model and collusions between up to $N-1$ parties. To efficiently execute the secure backpropagation algorithm for training neural networks, we provide a generic packing approach that enables Single Instruction, Multiple Data (SIMD) operations on encrypted data. We also introduce arbitrary linear transformations within the cryptographic bootstrapping operation, optimizing the costly cryptographic computations over the parties, and we define a constrained optimization problem for choosing the cryptographic parameters. Our experimental results show that POSEIDON achieves accuracy similar to centralized or decentralized non-private approaches and that its computation and communication overhead scales linearly with the number of parties. POSEIDON trains a 3-layer neural network on the MNIST dataset with 784 features and 60K samples distributed among 10 parties in less than 2 hours.**

I. INTRODUCTION

In the era of big data and machine learning (ML), neural networks (NNs) are the state-of-the-art models, as they achieve remarkable predictive performance in various domains such as healthcare, finance, and image recognition [10], [77], [106]. However, training an accurate and robust deep learning model requires a large amount of diverse and heterogeneous data [121]. This phenomenon raises the need for data sharing among multiple data owners who wish to collectively train a deep learning model and to extract valuable and generalizable insights from their joint data. Nonetheless, data sharing among entities, such as medical institutions, companies, and organizations, is often not feasible due to the sensitive nature of the data [117], strict privacy regulations [2], [7], or the business competition between them [104]. Therefore, solutions that enable privacy-preserving training of NNs on the data of multiple parties are highly desirable in many domains.

A simple solution for collective training is to outsource the data of multiple parties to a *trusted party* that is able to train the NN model on their behalf and to retain the data and model's confidentiality, based on established stringent non-disclosure agreements. These confidentiality agreements, however, require a significant amount of time to be prepared by legal and technical teams [72] and are very costly [60]. Furthermore, the trusted party becomes a single point of failure, thus both data and model privacy could be compromised by data breaches, hacking, leaks, etc. Hence, solutions originating from the cryptographic community replace and emulate the trusted party with a group of computing servers. In particular, to enable privacy-preserving training of NNs, several studies employ multiparty computation (MPC) techniques and operate on the two [83], [28], three [82], [110], [111], or four [26], [27] server models. Such approaches, however, limit the number of parties among which the trust is split, often assume an honest majority among the computing servers, and require parties to communicate (i.e., secret share) their data outside their premises. This might not be acceptable due to the privacy and confidentiality requirements and the strict data protection regulations. Furthermore, the computing servers do not operate on their own data or benefit from the model training; hence, their only incentive is the reputation harm if they are caught, which increases the possibility of malicious behavior.

A recently proposed alternative for privacy-preserving training of NNs – without data outsourcing – is *federated learning*. Instead of bringing the data to the model, the model is brought (via a coordinating server) to the clients, who perform model updates on their local data. The updated models from the parties are averaged to obtain the global NN model [75], [63]. Although federated learning retains the sensitive input data locally and eliminates the need for data outsourcing, the model, that might also be sensitive, e.g., due to proprietary reasons, becomes available to the coordinating server, thus placing the latter in a position of power with respect to the remaining parties. Recent research demonstrates that sharing intermediate model updates among the parties or with the server might lead to various privacy attacks, such as extracting parties' inputs [53], [113], [120] or membership inference [78], [86]. Consequently, several works employ differential privacy to enable privacy-preserving exchanges of intermediate values and to obtain models that are free from adversarial inferences in federated learning settings [67], [101], [76]. Although differentially private techniques partially limit attacks to federated learning, they decrease the utility of the data and the resulting ML model. Furthermore, training robust and accurate models requires high privacy budgets, and as such, the level of privacy achieved in practice remains unclear [55]. Therefore, a distributed privacy-preserving deep learning approach requires strong cryptographic protection of the intermediate model updates during the training, as well as of the final model weights.

Recent cryptographic approaches for private distributed learning, e.g., [119], [42], not only have limited ML functionalities, i.e., regularized or generalized linear models, but also employ traditional encryption schemes that make them vulnerable to post-quantum attacks. This should be cautiously considered, as recent advances in quantum computing [47], [87], [105], [116], increase the need for deploying quantum-resilient cryptographic schemes that eliminate

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24119
www.ndss-symposium.org

potential risks for applications with long-term sensitive data. Froelicher et al. recently proposed SPINDLE [41], a generic approach for the privacy-preserving training of ML models in an N-party setting that employs multiparty lattice-based cryptography, thus achieving post-quantum security guarantees. However, the authors [41] demonstrate the applicability of their approach only for generalized linear models, and their solution lacks the necessary protocols and functions that can support the training of complex ML models, such as NNs.

In this work, we extend the approach of SPINDLE [41] and build POSEIDON, a novel system that enables the training and evaluation of NNs in a distributed setting and provides end-to-end protection of the parties' training data, the resulting model, and the evaluation data. Using multiparty lattice-based homomorphic encryption [84], POSEIDON enables NN executions with different types of layers, such as fully connected, convolution, and pooling, on a dataset that is distributed among N parties, e.g., a consortium of tens of hospitals, that trust only themselves for the confidentiality of their data and of the resulting model. POSEIDON relies on mini-batch gradient descent and protects, from any party, the intermediate updates of the NN model by maintaining the weights and gradients encrypted throughout the training phase. POSEIDON also enables the resulting encrypted model to be used for privacy-preserving inference on encrypted evaluation data.

We evaluate POSEIDON on several real-world datasets and various network architectures such as fully connected and convolutional neural network structures and observe that it achieves training accuracy levels on par with centralized or decentralized non-private approaches. Regarding its execution time, we find that POSEIDON trains a 2-layer NN model on a dataset with 23 features and 30,000 samples distributed among 10 parties, in 8.7 minutes. Moreover, POSEIDON trains a 3-layer NN with 64 neurons per hidden-layer on the MNIST dataset [66] with 784 features and 60K samples shared between 10 parties, in 1.4 hours, and a NN with convolutional and pooling layers on the CIFAR-10 [65] dataset (60K samples and 3,072 features) distributed among 50 parties, in 175 hours. Finally, our scalability analysis shows that POSEIDON's computation and communication overhead scales linearly with the number of parties and logarithmically with the number of features and the number of neurons in each layer.

In this work, we make the following contributions:

- We present POSEIDON, a novel system for privacy-preserving, quantum-resistant, federated learning-based training of and inference on NNs with N parties with unbounded N, that relies on multiparty homomorphic encryption and respects the confidentiality of the training data, the model, and the evaluation data.
- We propose an alternating packing approach for the efficient use of single instruction, multiple data (SIMD) operations on encrypted data, and we provide a generic protocol for executing NNs under encryption, depending on the size of the dataset and the structure of the network.
- We improve the distributed bootstrapping protocol of [84] by introducing arbitrary linear transformations for optimizing computationally heavy operations, such as pooling or a large number of consecutive rotations on ciphertexts.
- We formulate a constrained optimization problem for choosing the cryptographic parameters and for balancing the number of costly cryptographic operations required for training and evaluating NNs in a distributed setting.
- POSEIDON advances the state-of-the-art privacy-preserving solutions for NNs based on MPC [110], [83], [82], [12], [27], [111], by achieving better flexibility, security, and scalability:
  Flexibility. POSEIDON relies on a federated learning approach, eliminating the need for communicating the parties' confidential data outside their premises which might not be always feasible due to privacy regulations [2], [7]. This is in contrast to MPC-based solutions which require parties to distribute their data among several servers, and thus, fall under the cloud outsourcing model.
  Security. POSEIDON splits the trust among multiple parties, and guarantees its data and model confidentiality properties under a passive-adversarial model and collusions between up to $N-1$ parties, for unbounded N. On the contrary, MPC-based solutions limit the number of parties among which the trust is split (typically, 2, 3, or 4 servers) and assume an honest majority among them.
  Scalability. POSEIDON's communication is linear in the number of parties, whereas MPC-based solutions scale quadratically.
- Unlike differential privacy-based approaches for federated learning [67], [101], [76], POSEIDON does not degrade the utility of the data, and the impact on the model's accuracy is negligible.

To the best of our knowledge, POSEIDON is the first system that enables quantum-resistant distributed learning on neural networks with N parties in a federated learning setting, and that preserves the privacy of the parties' confidential data, the intermediate model updates, and the final model weights.

II. RELATED WORK

Privacy-Preserving Machine Learning (PPML). Previous PPML works focus exclusively on the training of (generalized) linear models [17], [57], [23], [34], [61], [62]. They rely on *centralized* solutions where the learning task is securely outsourced to a server, notably using homomorphic encryption (HE) techniques. As such, these works do not solve the problem of privacy-preserving *distributed* ML, where multiple parties collaboratively train an ML model on their data. To address the latter, several works propose multi-party computation (MPC) solutions where several tasks, such as clustering and regression, are distributed among 2 or 3 servers [54], [25], [89], [43], [44], [13], [100], [21], [31]. Although such solutions enable multiple parties to collaboratively learn on their data, the trust distribution is limited to the number of computing servers that train the model, and they rely on assumptions such as non-collusion, or an honest majority among the servers. There exist only a few works that extend the distribution of ML computations to N parties ($N \geq 4$) and that remove the need for outsourcing [33], [119], [42], [41]. For instance, Zheng et al. propose Helen, a system for privacy-preserving learning of linear models that combines HE with MPC techniques [119]. However, the use of the Paillier additive HE scheme [90] makes their system vulnerable to post-quantum attacks. To address this issue, Froelicher et al. introduce SPINDLE [41], a system that provides support for generalized linear models and security against post-quantum attacks. These works have paved the way for PPML computations in the N-party setting, but none of them addresses the challenges associated with the privacy-preserving training of and inference on neural networks (NNs).

Privacy-Preserving Inference on Neural Networks. In this research direction, the majority of works operate on the following setting: a central server holds a trained NN model and clients communicate their evaluation data to obtain predictions-as-a-service [45], [73], [59]. Their aim is to protect both the confidentiality of the server's model and the clients' data. Dowlin et al. propose the use of a ring-based leveled HE scheme to enable the inference phase on encrypted data [45]. Other works rely on hybrid approaches by employing two-party computation (2PC) and HE [59], [73], or secret sharing and garbled circuits to enable privacy-preserving inference on NNs [92], [97], [81]. For instance, Riazi et al. use garbled circuits to achieve constant round communication complexity during the evaluation of binary neural networks [97], whereas Mishra et al. propose a similar hybrid solution that outperforms previous works in terms of efficiency, by tolerating a small decrease in the model's accuracy [81].

Boemer et al. develop a deep-learning graph compiler for multiple HE cryptographic libraries [19], [20], such as SEAL [1], HElib [49], and Palisade [98]. Their work enables the deployment of a model, which is trained with well-known frameworks (e.g., Tensorflow [9], PyTorch [91]), and enables predictions on encrypted data. Dalskov et al. use quantization techniques to enable efficient privacy-preserving inference on models trained with Tensorflow [9] by using MP-SPDZ [5] and demonstrate benchmarks for a wide range of adversarial models [35].

All aforementioned solutions enable only privacy-preserving inference on NNs, whereas our work focuses on both the privacy-preserving training of and the inference on NNs, protecting the training data, the resulting model, and the evaluation data.

Privacy-Preserving Training of Neural Networks. A number of works focus on *centralized* solutions to enable privacy-preserving learning of NNs [103], [8], [115], [109], [85], [51]. Some of them, e.g., [103], [8], [115], employ differentially private techniques to execute the stochastic gradient descent while training a NN in order to derive models that are protected from inference attacks [102]. However, they assume that the training data is available to a *trusted* party that applies the noise required during the training steps. Other works, e.g., [109], [85], [51], rely on HE to outsource the training of multilayer perceptrons to a central server. These solutions either employ cryptographic parameters that are far from realistic [109], [85], or yield impractical performance [51]. Furthermore, they do not support the training of NNs in the N-party setting, which is the main focus of our work.

A number of works that enable privacy-preserving *distributed* learning of NNs employ MPC approaches where the parties' confidential data is distributed among two [83], [12], three [82], [110], [111], [52], [28], or four servers [26], [27] (2PC, 3PC, and 4PC, resp.). For instance, in the 2PC setting, Mohassel and Zhang describe a system where data owners process and secret-share their data among two non-colluding servers to train various ML models [83], and Agrawal et al. propose a framework that supports discretized training of NNs by ternarizing the weights [12]. Then, Mohassel and Rindal extend [83] to the 3PC setting and introduce new fixed-point multiplication protocols for shared decimal numbers [82]. Wagh et al. further improve the efficiency of privacy-preserving NN training on secret-shared data [110] and provide security against malicious adversaries, assuming an honest majority among 3 servers [111]. More recently, 4PC honest-majority malicious frameworks for PPML have been proposed [26], [27]. These works split the trust between more servers and achieve better round complexities than previous ones, yet they do not address NN training among N-parties. Note that 2PC, 3PC, and 4PC solutions fall under the *cloud outsourcing* model, as the data of the parties has to be transferred to several servers among which the majority has to be trusted. Our work, however, focuses on a distributed setting where the data owners maintain their data locally and iteratively update the collective model, yet data and model confidentiality is ensured in the existence of a dishonest majority in a semi-honest setting, thus withstanding passive adversaries and up to $N-1$ collusions between them. We provide a comparison with these works in Section VII-F.

Another widely employed approach for training NNs in a distributed manner is that of federated learning [75], [64], [63]. The main idea is to train a global model on data that is distributed across multiple clients, with the assistance of a server that coordinates model updates on each client and averages them. This approach does not require clients to send their local data to the central server, but several works show that the clients' model updates *leak* information about their local data [53], [113], [120]. To counter this, some works focus on secure aggregation techniques for distributed NNs, based on HE [93], [94] or MPC [22]. Although encrypting the gradient values prevents the leakage of parties' confidential data to the central server, these solutions do not account for potential leakage from the *aggregate* values themselves. In particular, parties that decrypt the received model before the next iteration are able to infer information about other parties' data from its parameters [53], [78], [86], [120]. Another line of research relies on differential privacy (DP) to enable privacy-preserving federated learning for NNs. Shokri and Shmatikov [101] apply DP to the parameter update stages, and Li et al. design a privacy-preserving federated learning system for medical image analysis where the parties exchange differentially private gradients [67]. McMahan et al. propose differentially private federated learning [76], by employing the moments accountant method [8], to protect the privacy of all the records belonging to a user. Finally, other works combine MPC with DP techniques to achieve better privacy guarantees [56], [108]. While DP-based learning aims to mitigate inference attacks, it significantly degrades model utility, as training accurate NN models requires high privacy budgets [96]. As such, it is hard to quantify the level of privacy protection that can be achieved with these approaches [55]. To account for these issues, our work employs multiparty homomorphic encryption techniques to achieve *zero-leakage* training of NNs in a distributed setting where the parties' intermediate updates and the final model remain under encryption.

III. Preliminaries

We provide background information about NNs and the multiparty homomorphic encryption (MHE) scheme on which POSEIDON relies to achieve privacy-preserving training of and inference on NN models in a federated N-party setting.

A. *Neural Networks*

Neural networks (NNs) are machine learning algorithms that extract complex non-linear relationships between the input and output data. Typical NNs are composed of a pipeline of layers where feed-forward and backpropagation steps for linear and non-linear transformations (activations) are applied to the input data iteratively [48]. Each training iteration consists of one forward pass and one backward pass, and the term epoch refers to processing once all the samples in a dataset.

Multilayer perceptrons (MLPs) are fully-connected deep NN structures widely used in the industry [58]. MLPs are composed of an input layer, one or more hidden layer(s), and an output layer; each neuron is connected to all neurons in the following layer. At iteration k, the weights between layers j and $j+1$, are denoted by a matrix W_j^k, and the matrix L_j represents the activation of the neurons in the j^{th} layer. The forward pass first linearly combines each layer's weights with the activation values of the previous layer, i.e., $U_j = W_j^k \times L_{j-1}$. Then, an activation function yields the values of each layer as $L_j = \varphi(U_j)$.

Backpropagation, a gradient descent-based method, is then used to update the weights in the backward pass. Here, we describe the update rules for mini-batch gradient descent, where a random batch of sample inputs of size B is used in each iteration. The aim is to minimize each iteration's error based on a cost function E and update the weights. The update rule is $W_j^{k+1} = W_j^k - \frac{\eta}{B}\nabla W_j^k$, where η is the learning rate and ∇W_j^k denotes the gradient of the cost function with respect to the weights and calculated as $\nabla W_j^k = \frac{\partial E}{\partial W_j^k}$. Backpropagation requires several transpose operations applied to matrices/vectors; we denote the transpose of a matrix/vector as W^T.

Convolutional neural networks (CNNs) are trained similarly and typically consist of convolutional (CV), pooling, and fully connected (FC) layers. CV layer operations can be expressed as FC layer ones by representing them as matrix multiplications; in our protocols, we simplify CV layer operations with this representation [110], [3]. Finally, pooling layers are downsampling layers where a kernel (a matrix that moves over the input matrix with a stride of a) is convolved with the current sub-matrix. For a kernel of size $k \times k$, a pooling layer yields the minimum, maximum, or average of each $k \times k$ sub-matrix of its input.

B. Distributed Deep Learning

We employ the well-known MapReduce abstraction to describe the training of data-parallel NNs in a distributed setting where multiple data providers hold their respective datasets [122], [32]. We rely on parallel gradient descent [122], where each party performs b local iterations and calculates each layer's partial gradients. These gradients are aggregated over all parties and the reducer updates the model with their average [32]. This process is repeated for m global iterations. Averaging the gradients from N parties is equivalent to performing batch gradient descent with a batch size of $b \times N$; we differentiate between the local batch size as b and the global batch size as $B = b \times N$.

C. Multiparty Homomorphic Encryption (MHE)

Our system relies on the Cheon-Kim-Kim-Song (CKKS) [29] variant of the MHE scheme proposed by Mouchet et al. [84]. In this scheme, a public collective key is known by all parties while the corresponding secret key is distributed among them. Thus, decryption is only possible with the participation of *all* parties. We choose this scheme as: (i) it is well suited for floating-point arithmetic, (ii) it is based on the ring learning with errors (RLWE) problem [74], making our system secure against post-quantum attacks [11], (iii) it enables secure and flexible collaborative computations between parties without sharing their respective secret key, and (iv) it enables a secure collective key-switch functionality, that is, changing the encryption key of a ciphertext without decryption. We provide a brief description of the scheme's functionalities that we use throughout our protocols. The cyclotomic polynomial ring of dimension $\mathcal{N}$, with $\mathcal{N}$ a power-of-two integer, defines the plaintext and ciphertext space as $R_{Q_L} = \mathbb{Z}_{Q_L}[X]/(X^{\mathcal{N}}+1)$, with $Q_L = \prod_0^L q_i$ in our case. Each q_i is a unique prime, and Q_L is the ciphertext modulus at an initial level L. A plaintext encodes a vector of up to $\mathcal{N}/2$ values. Below, we introduce the main functions used in our system in Scheme III.1. We denote by $\boldsymbol{c} = (c_0, c_1) \in R^2_{Q_L}$ and $p \in R_{Q_L}$, a ciphertext (indicated as boldface) and a plaintext, respectively. $\bar{p}$ denotes an encoded (packed) plaintext. We denote by $L_{\boldsymbol{c}}$, $S_{\boldsymbol{c}}$, L, and S, the current level of a ciphertext $\boldsymbol{c}$, the current scale of $\boldsymbol{c}$, the initial level, and the initial scale (precision) of a fresh ciphertext respectively, and we use the equivalent notations for plaintexts. The functions (in Scheme III.1) that start with 'D' are distributed, and executed among all the secret-key-holders, whereas the others can be executed by anyone with the public key.

$\mathsf{Res}(\cdot)$ is applied to a resulting ciphertext after each multiplication. For a ciphertext at an initial level L, at most an L-depth circuit can be evaluated. To enable more homomorphic operations, the ciphertext must be re-encrypted to its original level L. This is done by the bootstrapping functionality ($\mathsf{DBootstrap}(\cdot)$). $\mathsf{Encode}(\cdot)$ enables us to pack several values into one ciphertext and operate on them in parallel. We differentiate between the functionality of the collective key-switch ($\mathsf{DKeySwitch}(\cdot)$), that requires interaction between all the parties, and a local key-switch ($\mathsf{KS}(\cdot)$) that uses a special public-key. The former is used to decrypt the results or change the encryption key of a ciphertext. The latter, which does not require interactivity, is used during the local computation for slot rotations or relinearization after each multiplication.

$\mathsf{SecKeyGen}(1^\lambda)$: Returns the set of secret keys $\{sk_i\}$,
i.e., sk_i for each party P_i, for security parameter λ.
$\mathsf{DKeyGen}(\{sk_i\})$: Returns the collective public key pk.
$\mathsf{Encode}(msg)$: Returns a plaintext $\bar{p} \in R_{Q_L}$ with scale S, encoding msg.
$\mathsf{Decode}(\bar{p})$: For $\bar{p} \in R_{Q_{L_p}}$ and scale S_p, returns the decoding of p.
$\mathsf{DDecrypt}(\boldsymbol{c}, \{sk_i\})$: For $\boldsymbol{c} \in R^2_{Q_{L_{\boldsymbol{c}}}}$ and scale $S_{\boldsymbol{c}}$, returns the plaintext
$p \in R_{Q_{L_{\boldsymbol{c}}}}$ with scale $S_{\boldsymbol{c}}$.
$\mathsf{Enc}(pk, \bar{p})$: Returns $\boldsymbol{c}_{pk} \in R^2_{Q_L}$ with scale S such that
$\mathsf{DDecrypt}(\boldsymbol{c}_{pk}, \{sk_i\}) \approx \bar{p}$.
$\mathsf{Add}(\boldsymbol{c}_{pk}, \boldsymbol{c}'_{pk})$: Returns $(\boldsymbol{c}+\boldsymbol{c}')_{pk}$ at level $\min(L_{\boldsymbol{c}}, L_{\boldsymbol{c}'})$
and scale $\max(S_{\boldsymbol{c}}, S_{\boldsymbol{c}'})$.
$\mathsf{Sub}(\boldsymbol{c}_{pk}, \boldsymbol{c}'_{pk})$: Returns $(\boldsymbol{c}-\boldsymbol{c}')_{pk}$ at level $\min(L_{\boldsymbol{c}}, L_{\boldsymbol{c}'})$
and scale $\max(S_{\boldsymbol{c}}, S_{\boldsymbol{c}'})$.
$\mathsf{Mul}_{\mathsf{pt}}(\boldsymbol{c}_{pk}, \bar{p})$: Returns $(\boldsymbol{c}p)_{pk}$ at level $\min(L_{\boldsymbol{c}}, L_p)$, scale $(S_{\boldsymbol{c}} \times S_p)$.
$\mathsf{Mul}_{\mathsf{ct}}(\boldsymbol{c}_{pk}, \boldsymbol{c}'_{pk})$: Returns $(\boldsymbol{c}\boldsymbol{c}')_{pk}$ at level $\min(L_{\boldsymbol{c}}, L_{\boldsymbol{c}'})$, scale $(S_{\boldsymbol{c}} \times S_{\boldsymbol{c}'})$.
$\mathsf{RotL/R}(\boldsymbol{c}_{pk}, k)$: Homomorphically rotates $\boldsymbol{c}_{pk}$ to the left/right by k times.
$\mathsf{Res}(\boldsymbol{c}_{pk})$: Returns $\boldsymbol{c}_{pk}$ with scale $S_{\boldsymbol{c}}/q_{L_{\boldsymbol{c}}}$ at level $L_{\boldsymbol{c}}-1$.
$\mathsf{SetScale}(\boldsymbol{c}_{pk}, S)$: Returns $\boldsymbol{c}_{pk}$ with scale S at level $L_{\boldsymbol{c}}-1$.
$\mathsf{KS}(\boldsymbol{c}_{pk} \in R^3)$: Returns $\boldsymbol{c}_{pk} \in R^2$.
$\mathsf{DKeySwitch}(\boldsymbol{c}_{pk}, pk', \{sk_i\})$: Returns $\boldsymbol{c}_{pk'}$.
$\mathsf{DBootstrap}(\boldsymbol{c}_{pk}, L_{\boldsymbol{c}}, S_{\boldsymbol{c}}, \{sk_i\})$: Returns $\boldsymbol{c}_{pk}$ with initial level L, scale S.

Scheme III.1: *Frequently used cryptographic operations.*

IV. SYSTEM OVERVIEW

We introduce POSEIDON's system and threat model, as well as its objectives (Sections IV-A and IV-B). Moreover, we provide a high level description of its functionality (Sections IV-C and IV-D).

A. System and Threat Model

We introduce POSEIDON's system and threat model below.

System Model. We consider a setting where N parties, each locally holding its own data X_i and a one-hot vector of labels y_i, collectively train a neural network (NN) model. At the end of the training process, a querier – which can be one of the N parties or an external entity – queries the model and obtains prediction results y_q on its evaluation data X_q. The parties involved in the training process are interested in preserving the privacy of their local data, the intermediate model updates, and the resulting model. The querier obtains prediction results on the trained model and keeps its evaluation data confidential. We assume that the parties are interconnected and organized in a tree-network structure for communication efficiency. However, our system is fully distributed and does not assume any hierarchy, therefore remaining agnostic of the network topology, e.g., we can consider a fully-connected network.

Threat Model. We consider a *passive-adversary model* with collusions of up to $N-1$ parties: I.e., the parties follow the protocol but up to $N-1$ parties might share among them their inputs and observations during the training phase of the protocol, to extract information about the other parties' inputs through membership inference or federated learning attacks [78], [86], [53], [120], [113], prevented by our work. Inference attacks on the model's *prediction* phase, such as membership [102] or model inversion [39], exploit the final prediction result and are out-of-the-scope of this work. We discuss complementary security mechanisms that can limit the information a querier infers from the prediction results and an extension to the active-adversary model in Appendix D-A.

B. Objectives

Poseidon's main objective is to enable the privacy-preserving training of and the evaluation on NNs in the above system and threat model. During the training process, Poseidon protects both the intermediate updates and the final model weights — that can potentially leak information about the parties' input data [53], [78], [86], [120] — from any party. In the inference step, the parties holding the protected model should not learn the querier's data, or the prediction results, and the querier should not obtain the model's weights. Therefore, Poseidon's objective is to protect the parties' and querier's **data confidentiality**, as well as the trained **model confidentiality**, as defined below:

- **Data Confidentiality.** During training and prediction, no party P_i (including the querier P_q) should learn more information about the input data X_j of any other honest party P_j ($j \neq i$, including the querier P_q), other than what can be deduced from its own input data X_i, y_i (or the input X_q and output y_q, for the querier).
- **Model Confidentiality.** During training and prediction, no party P_i (including the querier P_q) should gain more information about the trained model weights, other than what can be deduced from its own input data X_i, y_i (or X_q, y_q for the querier).

C. Overview of Poseidon

Poseidon achieves its objectives by using the MHE scheme described in Section III-C. In particular, the model weights are kept encrypted, with the parties' collective public key, throughout the training process. The operations required for the communication-efficient training of NNs are enabled by the scheme's computation homomorphic properties, which enables the parties to perform operations between their local data and the encrypted model weights. To enable oblivious inference on the trained model, Poseidon utilizes the scheme's key-switching functionality that allows the parties to collectively re-encrypt the prediction results with the querier's public key.

Poseidon employs several packing schemes to enable SIMD operations on the weights of various NN layers and uses approximations that enable the evaluation of multiple activation functions (e.g., Sigmoid, Softmax, ReLU) under encryption. Furthermore, to account for the complex operations required during the training of a neural network, Poseidon uses the scheme's distributed (collective) bootstrapping capability that enables us to refresh ciphertexts. In the following subsection, we provide a high-level description of Poseidon's phases, the cryptographic operations and optimizations are described in Section V.

Throughout the paper, we present Poseidon as a synchronous distributed learning protocol. An extension to asynchronous distributed NNs is presented in Appendix D-B.

D. High-Level Protocols

To describe the distributed training of and evaluation on NNs, we employ the extended MapReduce abstraction for privacy-preserving ML computations introduced in SPINDLE [41]. The overall learning procedure is composed of four phases: **PREPARE**, **MAP**, **COMBINE**, and **REDUCE**. Protocol 1 describes the steps required for the federated training of a neural network with N parties. The bold terms denote encrypted values and $\boldsymbol{W}_{j,i}^{k}$ represents the weight matrix of the j^{th} layer, at iteration k, of the party P_i. When there is no ambiguity or when we refer to the global model, we replace the sub-index i with $\cdot$ and denote weights by $\boldsymbol{W}_{j,\cdot}^{k}$. Similarly, we denote the local gradients at party P_i by $\boldsymbol{\nabla W}_{j,i}^{k}$, for each network layer j and iteration k. Throughout the paper, the n^{th} row of a matrix that belongs to the i^{th} party is represented by $X_i[n]$ and its encoded (packed) version as $\bar{X}_i[n]$.

Protocol 1 Collective Training

```
Inputs: $X_i, y_i$ for $i \in \{1,2,...,N\}$
Outputs: $\boldsymbol{W}_{1,\cdot}^{m}, \boldsymbol{W}_{2,\cdot}^{m}, ..., \boldsymbol{W}_{\ell,\cdot}^{m}$
  PREPARE:
    Parties collectively agree on $\ell, h_1, ..., h_\ell, \eta, \varphi(\cdot), m, b$
    Each $P_i$ generates $sk_i \leftarrow \mathsf{SecKeyGen}(1^\lambda)$
    Parties collectively generate $pk \leftarrow \mathsf{DKeyGen}(\{sk_i\})$
    Each $P_i$ encodes its local data as $\bar{X}_i$, $\bar{y}_i$
    $P_1$ initializes $\boldsymbol{W}_{1,\cdot}^{0}, \boldsymbol{W}_{2,\cdot}^{0}, ..., \boldsymbol{W}_{\ell,\cdot}^{0}$
  for $k = 0 \rightarrow m-1$ do
    MAP:
      $P_1$ sends $\boldsymbol{W}_{1,\cdot}^{k}, \boldsymbol{W}_{2\cdot}^{k}, ..., \boldsymbol{W}_{\ell,\cdot}^{k}$ down the tree
      Each $P_i$ does:
        Local Gradient Descent Computation:
        $\boldsymbol{\nabla W}_{1,i}^{k}, \boldsymbol{\nabla W}_{2,i}^{k}, ..., \boldsymbol{\nabla W}_{\ell,i}^{k}$
    COMBINE:
      Parties collectively aggregate: $\boldsymbol{\nabla W}_{1,\cdot}^{k}, ..., \boldsymbol{\nabla W}_{\ell,\cdot}^{k} \leftarrow$
    $\sum_{i=1}^{N} \boldsymbol{\nabla W}_{1,i}^{k}, ..., \boldsymbol{\nabla W}_{\ell,i}^{k}$
      $P_1$ obtains $\boldsymbol{\nabla W}_{1,\cdot}^{k}, \boldsymbol{\nabla W}_{2,\cdot}^{k}, ..., \boldsymbol{\nabla W}_{\ell,\cdot}^{k}$
    REDUCE (performed by $P_1$) :
      for $j = 1 \rightarrow \ell$ do
        $\boldsymbol{W}_{j,\cdot}^{k+1} += \eta \frac{\boldsymbol{\nabla W}_{j,\cdot}^{k}}{b \times N}$
      end for
  end for
```

1) PREPARE: In this offline phase, the parties collectively agree on the learning parameters: the number of hidden layers (l), the number of neurons (h_j) in each layer $j, \forall j \in \{1,2,...,l\}$, the learning rate ($\eta$), the number of global iterations (m), the activation functions to be used in each layer ($\varphi(\cdot)$) and their approximations (see Section V-B), and the local batch size (b). Then, the parties generate their secret keys sk_i and collectively generate the public key pk. Subsequently, they collectively normalize or standardize their input data with the secure aggregation protocol described in [42]. Each P_i encodes (packs) its input data samples X_i and output labels y_i (see Section V-A) as $\bar{X}_i, \bar{y}_i$. Finally, the root of the tree (P_1) initializes and encrypts the global weights.

Weight Initialization. To avoid exploding or vanishing gradients, we rely on commonly used techniques: (i) Xavier initialization for the sigmoid or tanh activated layers: $W_j = r \times h_{j-1}$ where r is a random number sampled from a uniform distribution in the range $[-1,1]$ [46], and (ii) He initialization [50] for ReLU activated layers, where the Xavier-initialized weights are multiplied twice by their variance.

2) MAP: The root (P_1) communicates the current encrypted weights, to every other party for their local gradient descent (LGD) computation.

LGD Computation: Each P_i performs b forward and backward passes to compute and aggregate the local gradients, by processing each sample of its respective batch. Protocol 2 describes the LGD steps performed by each party P_i, at iteration k; $\odot$ represents an element-wise product and $\varphi'(\cdot)$ the derivative of an activation function. As the protocol refers to one local iteration for a specific party, we omit k and i from the weight and gradient indices. This protocol describes the typical operations for the forward and backward pass using gradient descent with the $L2$ loss (see Section III). We note that the operations in this protocol are performed over encrypted data.

3) COMBINE: In this phase, each party communicates its encrypted local gradients to their parent, and each parent homomorphically sums the received gradients with their own ones. At the end of this phase, the root of the tree (P_1) receives the globally aggregated gradients.

4) REDUCE: P_1 updates the global model weights by using the averaged aggregated gradients. The averaging is done with respect

Protocol 2 Local Gradient Descent (LGD) Computation

```
Inputs: $\boldsymbol{W}^k_{1,\cdot}, \boldsymbol{W}^k_{2,\cdot}, ..., \boldsymbol{W}^k_{\ell,\cdot}$
Outputs: $\boldsymbol{\nabla W}^k_{1,i}, \boldsymbol{\nabla W}^k_{2,i}, ..., \boldsymbol{\nabla W}^k_{\ell,i}$. Note that $i$ and $k$ indices are omitted in this protocol.
for $t = 1 \rightarrow b$ do                                    ▷ Forward Pass
    $L_0 = \bar{X}[t]$
    for $j = 1 \rightarrow \ell$ do
        $\boldsymbol{U}_j = \boldsymbol{L}_{j-1} \times \boldsymbol{W}_j$
        $\boldsymbol{L}_j = \varphi(\boldsymbol{U}_j)$
    end for
    $\boldsymbol{E}_\ell = \bar{y}[t] - \boldsymbol{L}_\ell$                         ▷ Backpropagation
    $\boldsymbol{E}_\ell = \varphi'(\boldsymbol{U}_\ell) \odot \boldsymbol{E}_\ell$
    $\boldsymbol{\nabla W}_\ell += \boldsymbol{L}^{\boldsymbol{T}}_{\ell-1} \times \boldsymbol{E}_\ell$
    for $j = \ell - 1 \rightarrow 1$ do
        $\boldsymbol{E}_j = \boldsymbol{E}_{j+1} \times \boldsymbol{W}^{\boldsymbol{T}}_{j+1}$
        $\boldsymbol{E}_j = \varphi'(\boldsymbol{U}_j) \odot \boldsymbol{E}_j$
        $\boldsymbol{\nabla W}_j += \boldsymbol{L}^{\boldsymbol{T}}_{j-1} \times \boldsymbol{E}_j$
    end for
end for
```

to the global batch size $|B| = b \times N$, as described in Section III-B.

Training Termination: In our system, we stop the learning process after a predefined number of epochs. However, we note that several early-stop techniques [95] for the NN training termination can be straightforwardly integrated to POSEIDON.

Prediction: At the end of the training phase, the model is kept in an encrypted form such that no individual party or the querier can access the model weights. To enable oblivious inference, the querier encrypts its evaluation data X_q with the parties' collective key. We note that an oblivious inference is equivalent to one forward pass (see Protocol 2), except that the first plaintext multiplication ($\mathsf{Mul}_{\mathsf{pt}}(\cdot)$) of L_0 with the first layer weights is substituted with a ciphertext multiplication ($\mathsf{Mul}_{\mathsf{ct}}(\cdot)$). At the end of the forward pass, the parties collectively re-encrypt the result with the querier's public key by using the key-switch functionality of the underlying MHE scheme. Thus, only the querier is able to decrypt the prediction results. Note that any party P_i can perform the oblivious inference step, but the collaboration between all the parties is required to perform the distributed bootstrap and key-switch functionalities.

V. CRYPTOGRAPHIC OPERATIONS AND OPTIMIZATIONS

We first present the alternating packing (AP) approach that we use for packing the weight matrices of NNs (Section V-A). We then explain how we enable activation functions on encrypted values (Section V-B) and introduce the cryptographic building blocks and functions employed in POSEIDON (Section V-C), together with their execution pipeline and their complexity (Sections V-D and V-E). Finally, we formulate a constrained optimization problem that depends on a cost function for choosing the parameters of the cryptoscheme (Section V-F).

A. Alternating Packing (AP) Approach

For the efficient computation of the steps in Protocol 2, we rely on the packing capabilities of the cryptoscheme that enables SIMD operations on ciphertexts. Packing enables coding a vector of values in a ciphertext and parallelizing the computations across its different slots, thus significantly improving the overall performance.

Existing packing strategies that are commonly used for ML operations on encrypted data [41], e.g., the row-based [61] or diagonal [49], require a high number of rotations for the execution of the matrix-matrix multiplications and matrix transpose operations, performed during the forward and backward pass of the local gradient descent computation (see Protocol 2). We here remark that the number of rotations has a significant effect on the overall training time of a NN on encrypted data, as they require costly key-switch operations (see Section V-E). As an example, the diagonal approach scales linearly with the size of the weight matrices, when it is used for batch-learning of NNs, due to the matrix transpose operations in the backpropagation. We follow a different packing approach and process each batch sample one by one, making the execution embarrassingly parallelizable. This enables us to optimize the number of rotations, to eliminate the transpose operation applied to matrices in the backpropagation, and to scale logarithmically with the dimension and number of neurons in each layer.

We propose an "alternating packing (AP) approach" that combines row-based and column-based packing, i.e., rows or columns of the matrix are vectorized and packed into one ciphertext. In particular, the weight matrix of every FC layer in the network is packed following the opposite approach from that used to pack the weights of the previous layer. With the AP approach, the number of rotations scales logarithmically with the dimension of the matrices, i.e., the number of features (d), and the number of hidden neurons in each layer (h_i). To enable this, we pad the matrices with zeros to get power-of-two dimensions. In addition, the AP approach reduces the cost of transforming the packing between two consecutive layers.

Protocol 3 describes a generic way for initializing the encrypted weights for an ℓ-layer MLP by P_1 and for encoding the input matrix (X_i) and labels (y_i) of each party P_i. It takes as inputs the NN parameters: The dimension of the data (d) that describes the shape of the input layer, the number of hidden neurons in the j^{th} layer (h_j), and the number of outputs (h_ℓ). We denote by gap a vector of zeros, and by $|\cdot|$ the size of a vector or the number of rows of a matrix. $\mathsf{Replicate}(v,k,gap)$ returns a vector that replicates v, k times with a gap in between each replica. $\mathsf{Flatten}(W,gap,dim)$, flattens the rows or columns of a matrix W into a vector and introduces gap in between each row/column. If a vector is given as input to this function, it places gap in between all of its indices. The argument dim indicates flattening of rows ('r') or columns ('c') and $dim =$ '$\cdot$' for the case of vector inputs.

We observe that the rows (or columns) packed into one ciphertext, must be aligned with the rows (or columns) of the following layer for the next layer multiplications in the forward pass and for the alignment of multiplication operations in the backpropagation, as depicted in Table I (e.g., see steps F1, F6, B3, B5, B6). We enable this alignment by adding gap between rows or columns and using rotations, described in the next section. Note that these steps correspond to the weight initialization and to the input preparation steps of the **PREPARE** (offline) phase.

Convolutional Layer Packing. To optimize the SIMD operations for CV layers, we decompose the n^{th} input sample $X_i[n]$ into t smaller matrices according to the kernel size $h = f \times f$. We pack these decomposed flattened matrices into one ciphertext, with a gap in between each matrix that is defined based on the number of neurons in the next layer ($h_2 - h_1$), similarly to the AP approach. The weight matrix is then replicated t times with the same gap between each replica. If the next layer is another convolutional or downsampling layer, the gap is not needed and the values in the slots are re-arranged during the training execution (see Section V-C). Lastly, we introduce the average-pooling operation to our bootstrapping function ($\mathsf{DBootstrapALT}(\cdot)$, see Section V-C), and we re-arrange almost for free the slots for any CV layer that comes after average-pooling.

Protocol 3 Alternating Packing (AP) Protocol

```
Inputs: $X_i, y_i, d, \{h_1, h_2, ..., h_\ell\}, \ell$
Outputs: $\boldsymbol{W}^0_{1,\cdot}, \boldsymbol{W}^0_{2,\cdot}, ..., \boldsymbol{W}^0_{\ell,\cdot}, \bar{X}_i, \bar{y}_i$
for $i = 1 \rightarrow N$ each $P_i$ do
    Initialize $|gap| = max(h_1 - d, 0)$                ▷ Input Preparation
    for $n = 1 \rightarrow |X_i|$ do
        $X_i[n] = \mathsf{Replicate}(X_i[n], h_1, gap)$
        $\bar{X}_i[n] = \mathsf{Encode}(X_i[n])$
    end for
    if $\ell\%2! = 0$ then                               ▷ Labels Preparation
        Initialize $|gap| = h_\ell$
        $y_i = \mathsf{Flatten}(y_i, gap, '\cdot')$
    end if
    $\bar{y}_i = \mathsf{Encode}(y_i)$
    if i==1 then                    ▷ $P_1$ performs Weight Initialization:
        Initialize $W^0_{1,\cdot}, W^0_{2,\cdot}, ..., W^0_{\ell,\cdot}$
        for $j = 1 \rightarrow \ell$ do
            if $j\%2 == 0$ then                          ▷ Row Packing
                if $h_{j-2} > h_j$ then
                    Initialize $|gap| = h_{j-2} - h_j$
                end if
                $W^0_{j,\cdot} = \mathsf{Flatten}(W^0_{j,\cdot}, gap, 'r')$
                $\boldsymbol{W}^0_{j,\cdot} = \mathsf{Enc}(pk, W^0_{j,\cdot})$
            else                                         ▷ Column Packing
                if $h_{j+1} > h_{j-1}$ then
                    Initialize $|gap| = h_{j+1} - h_{j-1}$
                end if
                $W^0_{j,\cdot} = \mathsf{Flatten}(W^0_{j,\cdot}, gap, 'c')$
                $\boldsymbol{W}^0_{j,\cdot} = \mathsf{Enc}(pk, W^0_{j,\cdot})$
            end if
        end for
    end if
end for
```

We note that high-depth kernels, i.e., layers with a large number of kernels, require a different packing optimization. In this case, we alternate row and column-based packing (similar to the AP approach), replicate the decomposed matrices, and pack all kernels in one ciphertext. This approach introduces k multiplications in the **MAP** phase, where k is the number of kernels in that layer, and comes with reduced communication overhead; the latter would be k times larger for **COMBINE**, **MAP**, and $\mathsf{DBootstrap}(\cdot)$, if the packing described in the previous paragraph was employed.

Downsampling (Pooling) Layers. As there is no weight matrix for downsampling layers, they are not included in the offline packing phase. The cryptographic operations for pooling are described in Section V-D.

B. *Approximated Activation Functions*

For the encrypted evaluation of non-linear activation functions, such as Sigmoid or Softmax, we use least-squares approximations and rely on the optimized polynomial evaluation that, as described in [41], consumes $\lceil \log(d_a + 1) \rceil$ levels for an approximation degree d_a. For the piece-wise function ReLU, we approximate the smooth approximation of ReLU, softplus (SmoothReLU), $\varphi(x) = \ln(1 + e^x)$ with least-squares. Lastly, we use derivatives of the approximated functions.

To achieve better approximation with the lowest possible degree, we apply two approaches to keep the input range of the activation function as small as possible, by using (i) different weight initialization techniques for different layers (i.e., Xavier or He initialization), and (ii) collective normalization of the data by sharing and collectively aggregating statistics on each party's local data in a privacy-preserving way [42]. Finally, the interval and the degree of the approximations are chosen based on the heuristics on the data distribution in a privacy-preserving way, as described in [51].

C. *Cryptographic Building Blocks*

We present each cryptographic function that we employ in POSEIDON. We also discuss the optimizations employed to avoid costly transpose operations in the encrypted domain.

Rotations. As we rely on packing capabilities, computation of the inner-sum of vector-matrix multiplications and transpose operation implies a restructuring of the vectors, that can only be achieved by applying slot rotations. Throughout the paper, we use two types of rotation functions: (i) Rotate For Inner Sum ($\mathsf{RIS}(\boldsymbol{c}, p, s)$) is used to compute the inner-sum of a packed vector $\boldsymbol{c}$ by homomorphically rotating it to the left with $\mathsf{RotL}(\boldsymbol{c}, p)$ and by adding it to itself iteratively $\log_2(s)$ times, and (ii) Rotate For Replication ($\mathsf{RR}(\boldsymbol{c}, p, s)$) replicates the values in the slots of a ciphertext by rotating the ciphertext to the right with $\mathsf{RotR}(\boldsymbol{c}, p)$ and by adding to itself, iteratively $\log_2(s)$ times. For both functions, p is multiplied by two at each iteration, thus both yield $\log_2(s)$ rotations. As rotations are costly cryptographic functions (see Table II), and the matrix operations required for NN training require a considerable amount of rotations, we minimize the number of rotations by leveraging a modified bootstrapping operation, that performs some of the rotations.

Distributed Bootstrapping with Arbitrary Linear Transformations. To execute the high-depth homomorphic operations, bootstrapping is required several times to refresh a ciphertext, depending on the initial level L. In POSEIDON, we use a distributed version of bootstrapping [84], [24], as it is several orders of magnitude more efficient than the traditional centralized bootstrapping. Then we modify it, leveraging on the interaction to automatically perform some of the rotations, or pooling operations, embedded as transformations in the bootstrapping.

Mouchet et al. replace the expensive bootstrap circuit by a one-round protocol where the parties collectively switch a Brakerski/Fan-Vercauteren (BFV) [38] ciphertext to secret-shares in $\mathbb{Z}_t^{\mathcal{N}}$. We adapt their protocol to a re-encryption process that extends the ciphertext modulus from Q_ℓ back to Q_L. Because a modular reduction of the plaintext mod Q_ℓ would result in an incorrect re-encryption, we instead collectively switch the ciphertext to a secret-shared plaintext guaranteeing statistical indistinguishability during the re-encryption process.

We define this protocol as $\mathsf{DBootstrapALT}(\cdot)$ (Protocol 4) that takes as inputs a ciphertext $\boldsymbol{c}_{pk}$ at level ℓ encrypting a message msg and returns a ciphertext $\boldsymbol{c}'_{pk}$ at level L encrypting $\phi(msg)$, where $\phi(\cdot)$ is a linear transformation over the field of complex numbers. We denote by $||a||$ the infinity norm of the vector or polynomial a. As the security of the RLWE is based on computational indistinguishability, switching to the secret-shared domain does not hinder security. We refer to Appendix B for technical details and the security proof of our protocol.

Optimization of the Vector-Transpose Matrix Product. The backpropagation step of the local gradient computation at each party requires several multiplications of a vector (or matrix) with the transposed vector (or matrix) (see Lines 11-13 of Protocol 2). The naïve multiplication of a vector $\boldsymbol{v}$ with a transposed weight matrix $\boldsymbol{W^T}$ that is fully packed in one ciphertext, requires converting $\boldsymbol{W}$ of size $g \times k$, from column-packed to row-packed. This is equivalent to applying a permutation of the plaintext slots, that can be expressed with a plaintext matrix $W_{gk \times gk}$ and homomorphically computed by doing a matrix-vector multiplication. As a result, a naïve multiplication requires $\sqrt{g \times k}$ rotations followed by $\log_2(k)$ rotations to obtain the inner sum from the matrix-vector multiplication. We propose several approaches to reduce the number of rotations when computing the

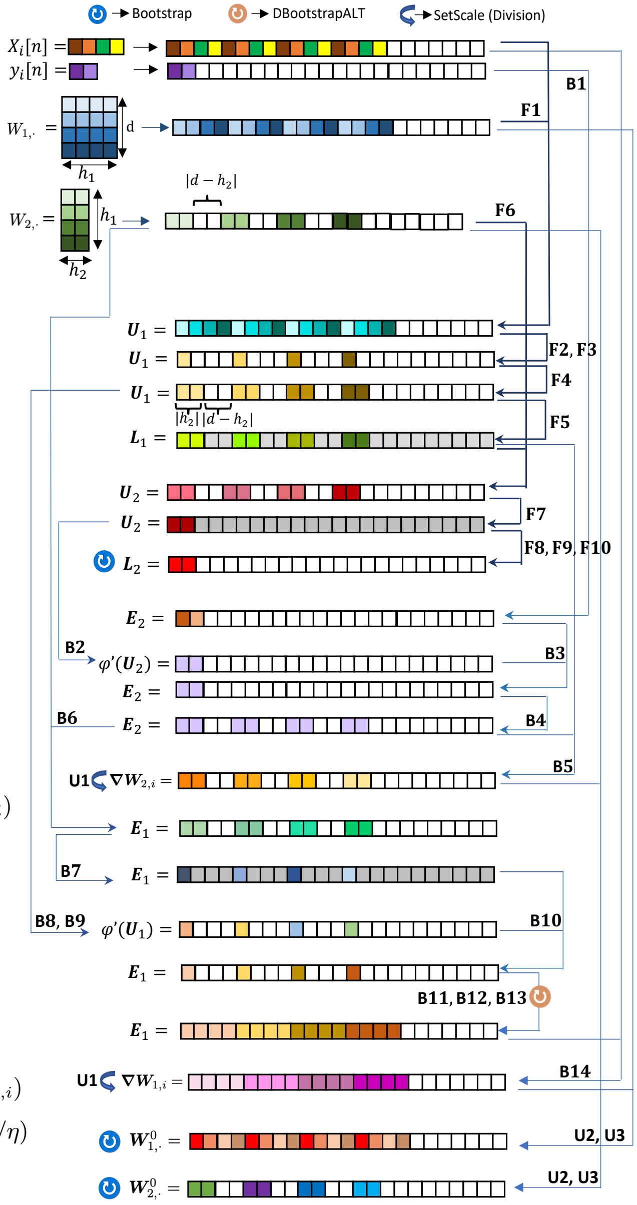
	AP Approach	Representation
PREPARE:		 Bootstrap DBootstrapALT SetScale (Division) $X_i[n] =$ $y_i[n] =$ $W_{1,\cdot} =$ d h_1 $\lvert d-h_2\rvert$ $W_{2,\cdot} =$ h_1 h_2 B1 F1 F6 $U_1 =$ $U_1 =$ $U_1 =$ $\lvert h_2\rvert \lvert d-h_2\rvert$ $L_1 =$ F2, F3 F4 F5 $U_2 =$ $U_2 =$ $L_2 =$ F7 F8, F9, F10 $E_2 =$ B2 $\varphi'(U_2) =$ $E_2 =$ B3 B6 $E_2 =$ B4 U1 $\nabla W_{2,i} =$ B5 $E_1 =$ B7 $E_1 =$ B8, B9 $\varphi'(U_1) =$ B10 $E_1 =$ B11, B12, B13 $E_1 =$ U1 $\nabla W_{1,i} =$ B14 $W_{1,\cdot}^0 =$ U2, U3 $W_{2,\cdot}^0 =$ U2, U3
1. Each P_i prepares $X_i[n]$,$y_i[n]$	Encode $X_i[n]$, $y_i[n] \rightarrow \bar{X}_i[n]$, $\bar{y}_i[n]$ $\bar{L}_0 = \bar{X}_i[n]$	
2. P_1 initializes $W_{1,\cdot}$	Vectorize columns, pack with $\lvert gap\rvert = 0$ $\boldsymbol{W}_{1,\cdot}^0 = \mathsf{Flatten}(W_{1,\cdot}^0, gap, \text{'}c\text{'})$	
3. P_1 initializes $W_{2,\cdot}$	Vectorize rows, pack with $\lvert gap\rvert = d - h_\ell$ $\boldsymbol{W}_{2,\cdot}^0 = \mathsf{Flatten}(W_{2,\cdot}^0, gap, \text{'}r\text{'})$	
4. Each P_i generates masks $\bar{m}_1$,$\bar{m}_2$	$\bar{m}_1 = [1,0,0,0,1,0,0,0,1,0,0,0,1,...]$ $\bar{m}_2 = [1,1,0,0,0,0,0,0,0,0,0,0,0,...]$	
Forward Pass (Each P_i): 1. $\boldsymbol{U}_1 = \bar{L}_0 \times \boldsymbol{W}_{1,\cdot}$ 2. $\boldsymbol{L}_1 = \varphi(\boldsymbol{U}_1)$	F1. $\boldsymbol{U}_1 = \mathsf{Mul}_{\mathsf{pt}}(\bar{L}_0, \boldsymbol{W}_{1,\cdot})$, $\mathsf{Res}(\boldsymbol{U}_1)$ F2. $\boldsymbol{U}_1 = \mathsf{RIS}(\boldsymbol{U}_1, 1, d)$ F3. $\boldsymbol{U}_1 = \mathsf{Mul}_{\mathsf{pt}}(\boldsymbol{U}_1, \bar{m}_1)$, $\mathsf{Res}(\boldsymbol{U}_1)$ F4. $\boldsymbol{U}_1 = \mathsf{RR}(\boldsymbol{U}_1, 1, h_\ell)$ F5. $\boldsymbol{L}_1 = \varphi(\boldsymbol{U}_1)$	
3. $\boldsymbol{U}_2 = \boldsymbol{L}_1 \times \boldsymbol{W}_{2,\cdot}$ 4. $\boldsymbol{L}_2 = \varphi(\boldsymbol{U}_2)$	F6. $\boldsymbol{U}_2 = \mathsf{Mul}_{\mathsf{ct}}(\boldsymbol{L}_1, \boldsymbol{W}_{2,\cdot})$, $\mathsf{Res}(\boldsymbol{L}_2)$ F7. $\boldsymbol{U}_2 = \mathsf{RIS}(\boldsymbol{U}_1, d, h_1)$ F8. $\boldsymbol{L}_2 = \mathsf{Mul}_{\mathsf{pt}}(\boldsymbol{L}_2, \bar{m}_2)$, $\mathsf{Res}(\boldsymbol{L}_2)$ F9. $\mathsf{DBootstrap}(\boldsymbol{U}_2)$ F10. $\boldsymbol{L}_2 = \varphi(\boldsymbol{U}_2)$	
Backpropagation (Each P_i): 1. $\boldsymbol{E}_2 = \bar{y}_i[n] - \boldsymbol{L}_2$	B1. $\boldsymbol{E}_2 = \mathsf{Sub}(\bar{y}_i[n], \boldsymbol{L}_2)$	
2. $\boldsymbol{E}_2 = (\varphi'(\boldsymbol{U}_2)) \odot \boldsymbol{E}_2$	B2. $\boldsymbol{d} = \varphi'(\boldsymbol{U}_2)$ B3. $\boldsymbol{E}_2 = \mathsf{Mul}_{\mathsf{ct}}(\boldsymbol{E}_2, \boldsymbol{d})$, $\mathsf{Res}(\boldsymbol{E}_2)$ B4. $\boldsymbol{E_2} = \mathsf{RR}(\boldsymbol{E_2}, d, h_1)$	
3. $\boldsymbol{\nabla W}_{2,i} = \boldsymbol{L}_1^T \times \boldsymbol{E}_l$	B5. $\boldsymbol{\nabla W}_{2,i} = \mathsf{Mul}_{\mathsf{ct}}(\boldsymbol{L}_1, \boldsymbol{E}_2)$, $\mathsf{Res}(\boldsymbol{\nabla W}_{2,i})$	
4. $\boldsymbol{E}_1 = \boldsymbol{E}_2 \times \boldsymbol{W}_{2,\cdot}^{\boldsymbol{T}}$	B6. $\boldsymbol{E}_1 = \mathsf{Mul}_{\mathsf{ct}}(\boldsymbol{E}_2, \boldsymbol{W}_{2,i})$, $\mathsf{Res}(\boldsymbol{E}_1)$ B7. $\boldsymbol{E}_1 = \mathsf{RIS}(\boldsymbol{E}_1, 1, h_\ell)$	
5. $\boldsymbol{E}_1 = (\varphi'(\boldsymbol{U}_1) \odot \boldsymbol{E}_1)$	B8. $\boldsymbol{d} = \varphi'(\boldsymbol{U}_1)$ B9. $\boldsymbol{d} = \mathsf{Mul}_{\mathsf{pt}}(\boldsymbol{d}, \bar{m}_1)$ B10. $\boldsymbol{E}_1 = \mathsf{Mul}_{\mathsf{ct}}(\boldsymbol{E}_1, \boldsymbol{d})$, $\mathsf{Res}(\boldsymbol{E}_1)$ B11. $\mathsf{DBootstrapALT}(\boldsymbol{E}_1)$	
6. $\boldsymbol{\nabla W}_{1,i} = \bar{L}_0^T \times \boldsymbol{E}_1$	B12. $\boldsymbol{E}_1 = \mathsf{Mul}_{\mathsf{pt}}(\boldsymbol{E}_1, \bar{m}_1)$, $\mathsf{Res}(\boldsymbol{E_1})$ B13. $\boldsymbol{E_1} = \mathsf{RR}(\boldsymbol{E}_1, 1, d)$ B14. $\boldsymbol{\nabla W}_{1,i} = \mathsf{Mul}_{\mathsf{pt}}(\bar{L}_0, \boldsymbol{E}_1)$, $\mathsf{Res}(\boldsymbol{\nabla W}_{1,i})$	
Update (at P_1): 1. $\boldsymbol{W}_{j,\cdot} += \eta \frac{\boldsymbol{\nabla W_{j,\cdot}}}{b \times N}$ $\forall j \in \{1,2,..,l\}$	U1. $\mathsf{SetScale}(\boldsymbol{\nabla W}_{j,\cdot}, S_{\boldsymbol{\nabla W}_{j,\cdot}} \times (b \times N))/\eta)$ U2. $\boldsymbol{W}_{j,\cdot} = \mathsf{Add}(\boldsymbol{W}_{j,\cdot}, \boldsymbol{\nabla W}_{j,\cdot})$ U3. $\mathsf{DBootstrap}(\boldsymbol{W}_{j,\cdot})$	

TABLE I: *Execution pipeline for a 2-layer MLP network with Alternating Packing (AP). Orange steps indicate the operations embedded in* $\mathsf{DBootstrapALT}(\cdot)$.

multiplication of a packed matrix (to be transposed) and a vector: **(i)** For the mini-batch gradient descent, we do not perform operations on the batch matrix. Instead, we process each batch sample in parallel, because having separate vectors (instead of a matrix that is packed into one ciphertext) enables us to reorder them at a lower cost. This approach translates ℓ matrix transpose operations to be transposes in vectors (the transpose of the vectors representing each layer activations in the backpropagation, see Line 13, Protocol 2), **(ii)** Instead of taking the transpose of $\boldsymbol{W}$, we replicate the values in the vector that will be multiplied with the transposed matrix (for the operation in Line 11, Protocol 2), leveraging the gaps between slots with the AP approach. That is, for a vector $\boldsymbol{v}$ of size k and the column-packed matrix $\boldsymbol{W}$ of size $g \times k$, $\boldsymbol{v}$ has the form $[a,0,0,0\ldots,b,0,0,0,\ldots,c,0,0,0,\ldots]$ with at least k zeros in between values (due to Protocol 3). Hence, any resulting ciphertext requiring the transpose of the matrix that will be subsequently multiplied, will also include gaps in between values. We apply $\mathsf{RR}(\boldsymbol{v},1,k)$ that consumes $\log_2(k)$ rotations to generate $[a,a,a,...0...,b,b,b,..,0...,c,c,c,...,0,...]$. Finally, we compute the product $\boldsymbol{\mathcal{P}} = \mathsf{Mul}_{\mathsf{ct}}(\boldsymbol{v},\boldsymbol{W})$ and apply $\mathsf{RIS}(\boldsymbol{\mathcal{P}},1,g)$ to get the inner sum with $\log_2(g)$ rotations, and **(iii)** We further optimize the performance by using $\mathsf{DBootstrapALT}(\cdot)$ (Protocol 4): If the ciphertext before the multiplication must be bootstrapped, we embed the $\log_2(k)$ rotations as a linear transformation performed during the bootstrapping.

D. Execution Pipeline

Table I depicts the pipeline of the operations for processing one sample in LGD computation for a 2-layer MLP. These steps can be extended to an ℓ-layer MLP by following the same operations for

Protocol 4 $\mathsf{DBootstrapALT}(\cdot)$

```
Inputs: $\boldsymbol{c}_{pk} = (c_0, c_1) \in R^2_{Q_\ell}$ encrypting $msg$, $\lambda$ a security parameter, $\phi(\cdot)$ a linear transformation over the field of complex numbers, $a$ a common reference polynomial, $s_i$ the secret-key of each party $P_i$, $\chi_{err}$ a distribution over $R$, where each coefficient is independently sampled from Gaussian distribution with the standard deviation $\sigma = 3.2$, and bound $\lfloor 6\sigma \rfloor$.
    Constraints: $Q_\ell > (N+1) \cdot ||msg|| \cdot 2^\lambda$.
Outputs: $\boldsymbol{c}'_{pk} = (c'_0, c'_1) \in R^2_{Q_L}$
for all $P_i$ do
    $M_i \leftarrow R_{||msg|| \cdot 2^\lambda}$, $e_{0,i}, e_{1,i} \leftarrow \chi_{err}$
    $M'_i \leftarrow \mathsf{Encode}(\phi(\mathsf{Decode}(M_i)))$
    $h_{0,i} \leftarrow s_i c_1 + M_i + e_{0,i} \mod Q_\ell$
    $h_{1,i} \leftarrow -s_i a - M_i + e_{1,i} \mod Q_L$
end for
$h_0 \leftarrow \sum h_{0,i}, h_1 \leftarrow \sum h_{1,i}$
$c'_0 \leftarrow \mathsf{Encode}(\phi(\mathsf{Decode}(c_0 + h_0 \mod Q_\ell)))$
return $\boldsymbol{c}'_{pk} = (c'_0 + h_1 \mod Q_L, a) \in R^2_{Q_L}$
```

multiple layers. The weights are encoded and encrypted using the AP approach, and the shape of the packed ciphertext for each step is shown in the representation column. Each forward and backward pass on a layer in the pipeline consumes one Rotate For Inner Sum ($\mathsf{RIS}(\cdot)$) and one Rotate For Replication ($\mathsf{RR}(\cdot)$) operation, except for the last layer, as the labels are prepared according to the shape of the ℓ^{th} layer output. In Table I, we assume that the initial level $L = 7$. When a bootstrapping function is followed by a masking (that is used to eliminate unnecessary values during multiplications) and/or several rotations, we perform these operations embedded as part of the distributed bootstrapping ($\mathsf{DBootstrapALT}(\cdot)$) to minimize their computational cost. The steps highlighted in orange are the operations embedded in the $\mathsf{DBootstrapALT}(\cdot)$. The complexity of each cryptographic function is analyzed in Section V-E.

Convolutional Layers. As we flatten, replicate, and pack the kernel in one ciphertext, a CV layer follows the exact same execution pipeline as an FC layer. However, the number of $\mathsf{RIS}(\cdot)$ operations for a CV layer is smaller than for an FC layer. That is because the kernel size is usually smaller than the number of neurons in an FC layer. For a kernel of size $h = f \times f$, the inner sum is calculated by $log_2(f)$ rotations. Note that when a CV layer is followed by an FC layer, the output of the i^{th} CV layer (L_i) already gives the flattened version of the matrix in one ciphertext. We apply $\mathsf{RR}(\boldsymbol{L}_i, 1, h_{i+1})$ for the preparation of the next layer multiplication. When a CV layer is followed by a pooling layer, however, the $\mathsf{RR}(\cdot)$ operation is not needed, as the pooling layer requires a new arrangement of the slots of $\boldsymbol{L}_i$. We avoid this costly operation by passing $\boldsymbol{L}_i$ to $\mathsf{DBootstrapALT}(\cdot)$, and by embedding both the pooling and its derivative in $\mathsf{DBootstrapALT}(\cdot)$.

Pooling Layers. We evaluate our system based on average pooling as it is the most efficient type of pooling that can be evaluated under encryption [45]. To do so, we exploit our modified collective bootstrapping to perform arbitrary linear transformations. The average pooling is a linear function, and so is its derivative (as opposed to max pooling). Therefore, in the case of a CV layer followed by a pooling layer, we apply $\mathsf{DBootstrapALT}(\cdot)$ and use it both to rearrange the slots, to compute the convolution of the average pooling, and its derivative, that is used later in the backward pass. For a $h = f \times f$ kernel size, this saves $\log_2(h)$ rotations and additions ($\mathsf{RIS}(\cdot)$) and one level if masking is needed. For max/min pooling, which are non-linear functions, we refer the reader to Appendix A and highlight that evaluating these functions under encryption remains unpractical.

E. Complexity Analysis

Table II displays the communication and *worst-case* computational complexity of POSEIDON's building blocks. This includes the MHE primitives, thus facilitating the discussion on the parameter selection in the following section. We define the complexity in terms of key-switch $\mathsf{KS}(\cdot)$ operations and recall that this is a different operation than $\mathsf{DKeySwitch}(\cdot)$, as explained in Section III-C. We note that $\mathsf{KS}(\cdot)$ and $\mathsf{DBootstrap}(\cdot)$ are 2 orders of magnitude slower than an addition operation, rendering the complexity of an addition negligible.

We observe that POSEIDON's communication complexity depends solely on the number of parties (N), the number of total ciphertexts sent in each global iteration (z), and the size of one ciphertext ($|\boldsymbol{c}|$). The building blocks that do not require communication are indicated as $-$.

In Table II, forward and backward passes represent the per-layer complexity for FC layers, so they are an *overestimate* for CV layers. Note that the number of multiplications differs in a forward pass and a backward pass, depending on the packing scheme, e.g., if the current layer is row-packed, it requires 1 less $\mathsf{Mul}_{\mathsf{ct}}(\cdot)$ in the backward pass, and we have 1 less $\mathsf{Mul}_{\mathsf{pt}}(\cdot)$ in several layers, depending on the masking requirements. Furthermore, the last layer of forward pass and the first layer of backpropagation take 1 less $\mathsf{RR}(\cdot)$ operation that we gain from packing the labels in the offline phase, depending on the NN structure (see Protocol 3). Hence, we save $2\log_2(h_\ell)$ rotations per one LGD computation.

In the **MAP** phase, we provide the complexity of the local computations per P_i, depending on the total number of layers ℓ. In the **COMBINE** phase, each P_i performs an addition for the collective aggregation of the gradients in which the complexity is negligible. To update the weights, **REDUCE** is done by one party (P_1) and divisions do not consume levels when performed with $\mathsf{SetScale}(\cdot)$. The complexity of an activation function ($\varphi(\cdot)$) depends on the approximation degree d_a. We note that the derivative of the activation function ($\varphi'(\cdot)$) has the same complexity as $\varphi(\cdot)$ with degree $d_a - 1$.

For the cryptographic primitives represented in Table II, we rely on the CKKS variant of the MHE cryptosystem in [84], and we report the dominating terms. The distributed bootstrapping takes 1 round of communication and the size of the communication scales with the number of parties (N) and the size of the ciphertext (see [84] for details).

F. Parameter Selection

We first discuss how to optimize the number of $\mathsf{Res}(\cdot)$ operations and give a cost function which is computed by the complexities presented in Table II. Finally, relying on this cost function we formulate an optimization problem for choosing POSEIDON' parameters.

As discussed in Section III-C, we assume that each multiplication is followed by a $\mathsf{Res}(\cdot)$ operation. The number of $\mathsf{Res}(\cdot)$ operations, however, can be reduced by checking the scale of the ciphertext. When the initial scale S is chosen such that $Q/S = r$ for a ciphertext modulus Q, the ciphertext is rescaled after r multiplications. This reduces the level consumption and is integrated into our cost function hereinafter.

Cryptographic Parameters Optimization. We define the overall complexity of an ℓ-layer MLP aiming to formulate a constrained optimization problem for choosing the cryptographic parameters. We first introduce the total number of bootstrapping operations ($\mathcal{B}$) required in one forward and backward pass, depending on the multiplicative depth as

$$\mathcal{B} = \frac{\ell(5 + \lceil \log_2(d_a + 1) + \lceil \log_2(d_a) \rceil \rceil)}{(L - \tau)r},$$

where $r = Q/S$. The number of total bootstrapping operations is calculated by the total number of consumed levels (numerator), the level requiring a bootstrap ($L - \tau$) and r which denotes how many consecutive multiplications are allowed before rescaling (denominator).

	Computational Complexity	#Levels Used	Communication	Rounds
FORWARD P. (FP)	$(\log_2(h_{i-1})+\log_2(h_{i+1}))\cdot\mathsf{KS}+\mathsf{Mul}_{\mathsf{ct}}+\mathsf{Mul}_{\mathsf{pt}}+\varphi$	$2+\lceil\log_2(d_a+1)\rceil$	–	–
BACKWARD P. (BP)	$(\log_2(h_{i-1})+\log_2(h_{i+1}))\cdot\mathsf{KS}+2\mathsf{Mul}_{\mathsf{ct}}+\mathsf{Mul}_{\mathsf{pt}}+\varphi'$	$3+\lceil\log_2(d_a)\rceil$	–	–
MAP	$\ell(\mathsf{FP}+\mathsf{BP})-2\log_2(h_\ell)$	$\ell(5+\lceil\log_2(d_a+1)+\lceil\log_2(d_a)\rceil)$	$z(N-1)\lvert c\rvert$	$1/2$
COMBINE	–	–	$z(N-1)\lvert c\rvert$	$1/2$
REDUCE	$\ell(\mathsf{Mul}_{\mathsf{pt}}+\mathsf{DB})$	–	–	–
DBootstrap (DB)	$N\log_2(N)(L+1)+N\log_2(N)(L_{\boldsymbol{c}}+1)$	–	$(N-1)\lvert c\rvert$	1
Mul Plaintext ($\mathsf{Mul}_{\mathsf{pt}}$)	$2N(L_{\boldsymbol{c}}+1)$	1	–	–
Mul Ciphertext ($\mathsf{Mul}_{\mathsf{ct}}$)	$4N(L_{\boldsymbol{c}}+1)+\mathsf{KS}$	1	–	–
Approx. Activation Function (φ)	$(2^\kappa+\mathsf{m}-\kappa-3+\lceil(d_a+1)/2^\kappa\rceil)\cdot\mathsf{Mul}_{\mathsf{ct}}$	$\lceil\log_2(d_a+1)\rceil$	–	–
$\mathsf{RIS}(\boldsymbol{c},p,s)$, $\mathsf{RR}(\boldsymbol{c},p,s)$	$\log_2(s)\cdot\mathsf{KS}$	–	–	–
Key-switch (KS)	$\mathcal{O}(\mathcal{N}\log_2(\mathcal{N})L_{\boldsymbol{c}}\beta)$	–	–	–

TABLE II: *Complexity analysis of* POSEIDON*'s building blocks.* $\mathcal{N},\alpha,L,L_{\boldsymbol{c}},d_a$ *stand for the cyclotomic ring size, the number of secondary moduli used during the key-switching, maximum level, current level, and the approximation degree, respectively.* $\beta=\lceil L_{\boldsymbol{c}}+1/\alpha\rceil$, $\mathsf{m}=\lceil\log(d_a+1)\rceil$, $\kappa=\lfloor\mathsf{m}/2\rfloor$.

The initial level of a fresh ciphertext L has an effect on the design of the protocols, as the ciphertext should be bootstrapped before the level $L_{\boldsymbol{c}}$ reaches a number $(L-\tau)$ that is close to zero, where τ depends on the security parameters. For a cyclotomic ring size $\mathcal{N}$, the initial level of a ciphertext L, and for the fixed NN parameters such as the number of layers ℓ, the number of neurons in each layer $h_1,h_2,...,h_\ell$, and for the number of global iterations m, the overall complexity is defined as

$$C(\mathcal{N},L)=m(\sum_{i=1}^{\ell}\{(2\log_2(h_{i-1})+\log_2(h_{i+1}))\cdot\mathsf{KS}$$
$$+3\mathsf{Mul}_{\mathsf{ct}}+2\mathsf{Mul}_{\mathsf{pt}}+\varphi+\varphi'\}-2\log_2(h_\ell)+\mathcal{B}\cdot\mathsf{DB}).$$

Note that the complexity of each $\mathsf{KS}(\cdot)$ operation depends on the level of the ciphertext that it is performed on (see Table II), but we use the initial level L in the cost function for the sake of clarity. The complexity of $\mathsf{Mul}_{\mathsf{ct}}$,$\mathsf{Mul}_{\mathsf{pt}}$,$\mathsf{DB}$, and KS is defined in Table II. Then, the optimization problem for a fixed scale (precision) S and a security level λ, which defines the security parameters, can be formulated as

$$\min_{\mathcal{N},L} C(\mathcal{N},L) \tag{1}$$

$$\text{subject to } mc=\{q_1,...,q_L\};L=|mc|;Q=\prod_{i=1}^{L}q_i;Q=kS,\, k\in\mathbb{R}^+;$$
$$Q_{L-\tau}>2^\lambda|plaintext|N;\mathcal{N}\leftarrow\text{postQsec}(Q,\lambda),$$

where postQsec(Q,L,λ) gives the necessary cyclotomic ring size $\mathcal{N}$, depending on the ciphertext modulus (Q) and on the desired security level (λ), according to the homomorphic encryption standard whitepaper [14]. Eq. (1) gives the optimal $\mathcal{N}$ and L for a given NN structure. We then pack each weight matrix into one ciphertext. We note that the solution might give an $\mathcal{N}$ that has fewer slots than the required number to pack the big weight matrices. In this case, we use a *multi-cipher* approach where we divide the flattened weight vector into multiple ciphertexts and carry out the NN operations on them in parallel. E.g., for a weight matrix of size $1{,}024\times64$ and $\mathcal{N}/2=4{,}096$ slots, we divide the weight matrix into $1{,}024\times64/4{,}096=16$ ciphers.

VI. Security Analysis

We demonstrate that POSEIDON achieves the Data and Model Confidentiality properties defined in Section IV-B, under a passive-adversary model with up to $N-1$ colluding parties. We follow the real/ideal world simulation paradigm [70] for the confidentiality proofs.

The semantic security of the CKKS scheme is based on the hardness of the decisional RLWE problem [29], [74], [71]. The achieved practical bit-security against state-of-the-art attacks can be computed using Albrecht's LWE-Estimator [14], [15]. The security of the used distributed cryptographic protocols, i.e., $\mathsf{DKeyGen}(\cdot)$ and $\mathsf{DKeySwitch}(\cdot)$, relies on the proofs by Mouchet et al. [84]. They show that these protocols are secure in a passive-adversary model with up to $N-1$ colluding parties, under the assumption that the underlying RLWE problem is hard [84]. The security of $\mathsf{DBootstrap}(\cdot)$, and its variant $\mathsf{DBootstrapALT}(\cdot)$ is based on Lemma 1, that we state and prove in Appendix B.

Remark 1. Any encryption broadcast to the network in Protocol 1 is re-randomized to avoid leakage about parties' confidential data by two consecutive broadcasts. We omit this operation in Protocol 1 for clarity.

Proposition 1. *Assume that* POSEIDON*'s encryptions are generated using the CKKS cryptosystem with parameters* $(\mathcal{N},Q_L,S)$ *ensuring a post-quantum security level of* λ*. Given a passive adversary corrupting at most* $N-1$ *parties,* POSEIDON *achieves* Data and Model Confidentiality *during training.*

Proof (Sketch). Let us assume a real-world simulator $\mathcal{S}_t$ that simulates the view of a computationally-bounded adversary corrupting $N-1$ parties, as such having access to the inputs and outputs of $N-1$ parties. As stated above, any encryption under CKKS with parameters that ensure a post-quantum security level of λ is semantically secure. During POSEIDON's training phase, the model parameters that are exchanged in between parties are encrypted, and all phases rely on the aforementioned CPA-secure-proven protocols. Moreover, as shown in Appendix B, the $\mathsf{DBootstrap}(\cdot)$ and $\mathsf{DBootstrapALT}(\cdot)$ protocols are simulatable. Hence, $\mathcal{S}_t$ can simulate all of the values communicated during POSEIDON's training phase by using the parameters $(\mathcal{N},Q_L,S)$ to generate random ciphertexts such that the real outputs cannot be distinguished from the ideal ones. The sequential composition of all cryptographic functions remains simulatable by $\mathcal{S}_t$ due to using different random values in each phase and due to Remark 1. As such, there is no dependency between the random values that an adversary can leverage on. Moreover, the adversary is not able to decrypt the communicated values of an honest party because decryption is only possible with the collaboration of *all* the parties. Following this, POSEIDON protects the data confidentiality of the honest party/ies.

Analogously, the same argument follows to prove that POSEIDON protects the confidentiality of the trained model, as it is a function of the parties' inputs, and its intermediate and final weights are always under encryption. Hence, POSEIDON eliminates federated learning attacks [53], [78], [86], [120], that aim at extracting private information about the parties from the intermediate parameters or the final model.

Proposition 2. *Assume that* POSEIDON*'s encryptions are generated using the CKKS cryptosystem with parameters ($\mathcal{N}$,Q_L,S) ensuring a post-quantum security level of λ. Given a passive adversary corrupting at most $N-1$ parties,* POSEIDON *achieves* Data *and* Model Confidentiality *during prediction.*

Proof (Sketch). (a) Let us assume a real-world simulator $\mathcal{S}_p$ that simulates the view of a computationally-bounded adversary corrupting $N-1$ computing nodes (parties). The *Data Confidentiality* of the honest parties and *Model Confidentiality* is ensured following the arguments of Proposition 1, as the prediction protocol is equivalent to a forward-pass performed during a training iteration by a computing party. Following similar arguments to Proposition 1, the encryption of the querier's input data (with the parties common public key pk) can be simulated by $\mathcal{S}_p$. The only additional function used in the prediction step is $\mathsf{DKeySwitch}(\cdot)$ that is proven to be simulatable by $\mathcal{S}_p$ [84]. Thus, POSEIDON ensures *Data Confidentiality* of the querier. (b) Let us assume a real-world simulator $\mathcal{S'}_p$ that simulates a computationally-bounded adversary corrupting $N-2$ parties and the querier. *Data Confidentiality* of the querier is trivial, as it is controlled by the adversary. The simulator has access to the prediction result as the output of the process for P_q, so it can produce all the intermediate (indistinguishable) encryptions that the adversary sees (based on the simulatability of the key-switch/collective decrypt protocol [84]). Following this and the arguments of Proposition 1, *Data and Model Confidentiality* are ensured during prediction. We remind here that the membership inference [102] and model inversion [39] are out-of-the-scope attacks (see Appendix D-A for complementary security mechanisms against these attacks).

VII. EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate POSEIDON's performance and present our empirical results. We also compare POSEIDON to other state-of-the-art privacy-preserving solutions.

A. Implementation Details

We implement POSEIDON in Go [6], building on top of the Lattigo lattice-based library [79] for the multiparty cryptographic operations. We make use of Onet [4] and build a decentralized system where the parties communicate over TCP with secure channels (TLS).

B. Experimental Setup

We use Mininet [80] to evaluate POSEIDON in a virtual network with an average network delay of 0.17ms and 1Gbps bandwidth. All the experiments are performed on 10 Linux servers with Intel Xeon E5-2680 v3 CPUs running at 2.5GHz with 24 threads on 12 cores and 256 GB RAM. Unless otherwise stated, in our *default* experimental setting, we instantiate POSEIDON with $N=10$ and $N=50$ parties. As for the parameters of the cryptographic scheme, we use a precision of 32 bits, number of levels $L=6$, and $\mathcal{N}=2^{13}$ for the datasets with $d<32$ or 32×32 images, and $\mathcal{N}=2^{14}$ for those with $d>32$, following the multi-cipher approach (see Section V-F).

C. Datasets

For the evaluation of POSEIDON's performance, we use the following real-world and publicly available datasets: (a) the Breast Cancer Wisconsin dataset (BCW) [18] with $n=699$,$d=9$,$h_\ell=2$, (b) the hand-written digits (MNIST) dataset [66] with $n=70{,}000$,$d=28\times28$,$h_\ell=10$, (c) the Epileptic seizure recognition (ESR) dataset [37] with $n=11{,}500$,$d=179$,$h_\ell=2$, (d) the default of credit card clients (CREDIT) dataset [114] with $n=30{,}000$,$d=23$,$h_\ell=2$, (d) the street view house numbers (SVHN) dataset [88] with colored images (3 channels), $n=600{,}000$,$d=3\times32\times32$,$h_\ell=10$, and (e) the CIFAR-10 and CIFAR-100 [65] datasets with colored images (3 channels), $n=60{,}000$,$d=3\times32\times32$, $h_\ell=10$, and $h_\ell=100$, respectively. Recall that h_ℓ represents the number of neurons in the last layer of a neural network (NN), i.e., the number of output labels. We convert SVHN to gray-scale to reduce the number of channels. Moreover, since we pad with zeros each dimension of a weight matrix to the nearest power-of-two (see Section V-A), for the experiments using the CREDIT, ESR, and MNIST datasets, we actually perform the NN training with $d=32$, 256, and 1,024 features, respectively. For SVHN, the number of features for a flattened gray-scale image is already a power-of-two ($32\times32=1{,}024$). To evaluate the scalability of our system, we generate synthetic datasets and vary the number of features or samples. Finally, for our experiments we evenly and randomly distribute all the above datasets among the participating parties. We note that the data and label distribution between the parties, and its effects on the model accuracy is orthogonal to this paper (see Appendix D-B for extensions related to this issue).

Dataset	Accuracy					Execution time (s)	
	C1	C2	L	D	POSEIDON	Training	Inference
BCW	97.8%	97.4%	93.9%	97.4%	96.9%	91.06	0.21
ESR	93.6%	91.2%	89.9%	91.1%	90.4%	851.84	0.30
CREDIT	81.4%	80.9%	79.6%	80.6%	80.2%	516.61	0.26
MNIST	92.1%	91.3%	87.8%	90.6%	89.9%	5,283.1	0.38

TABLE III: POSEIDON*'s accuracy and execution times for $N=10$ parties. The model accuracy is compared to several non-private approaches.*

D. Neural Network Configuration

For the BCW, ESR, and CREDIT datasets, we deploy a 2-layer fully connected NN with 64 neurons per layer, and we use the same NN structure for the synthetic datasets used to test POSEIDON's scalability. For the MNIST and SVHN datasets, we train a 3-layer fully connected NN with 64 neurons per-layer. For the CIFAR-10, we train two models: (i) a CNN with 2 CV and 2 average-pooling with a kernel size of 2×2, and 2 FC layers with 128 neurons and 10 neurons, labeled as N1, and (ii) a CNN with 4 CV with a kernel size of 3×4, 2 average-pooling with kernel size of 2×2 and 2 FC layers with 128 and 10 neurons labeled as N2. For CIFAR-100, we train a CNN with 6 CV with a kernel size of 3×4, 2 average-pooling with a kernel size of 2×2 and 2 FC layers with 128 neurons each. For all CV layers, we vary the number of filters between 3 to 16. We use the approximated sigmoid, SmoothReLU, or tanh activation functions (see Section V-B), depending on the dataset. We train the above models for 100, 600, 500, 1,000, 18,000, 25,000, 16,800, and 54,000 global iterations for the BCW, ESR, CREDIT, MNIST, SVHN, CIFAR-10-N1, CIFAR-10-N2, and CIFAR100 datasets, respectively. For the SVHN and CIFAR datasets, we use momentum-based gradient descent or Nesterov's accelerated gradient descent, which introduces an additional multiplication to the update rule (in the **MAP** phase). Finally, we set the local batch size b to 10 and, as such, the global batch size is $B=100$ in our default setting with 10 parties and $B=500$ with 50 parties. For a fixed number of layers, we choose the learning parameters by grid search with 3-fold cross-validation on clear data with the *approximated* activation functions. In a practical FL setting, however, the parties can collectively agree on these parameters by using secure statistics computations [42], [40].

E. Empirical Results

We experimentally evaluate POSEIDON in terms of accuracy of the trained model, execution time for both training and prediction phases, and communication overhead. We also evaluate POSEIDON's scalability with respect to the number of parties N, as well as the number of data samples n and features d in a dataset. We further provide microbenchmark timings and communication overhead for the various functionalities and operations for FC, CV, and pooling layers in Appendix C that can be used to extrapolate POSEIDON' execution time for different NN structures.

Model Accuracy. Tables III and IV display POSEIDON's accuracy results on the used real-world datasets with 10 and 50 parties, respectively. The accuracy column shows four baselines with the following approaches: two approaches where the data is collected to a central party in its clear form: centralized with original activation functions (C1), and centralized with approximated activation functions (C2); one approach where each party trains the model only with its local data (L), and a decentralized approach with approximated activation functions (D), where the data is distributed among the parties, but the learning is performed on cleartext data, i.e., without any protection of the gradients communicated between the parties. For all baselines, we use the same NN structure and learning parameters as POSEIDON, but adjust the learning rate (η) or use adaptive learning rate to ensure the range of the approximated activation functions is minimized, i.e., a smaller interval for an activation-function approximation requires smaller η to prevent divergence while bigger intervals make the choice of η more flexible. These baselines enable us to evaluate POSEIDON's accuracy loss due to the approximation of the activation functions, distribution, encryption, and the impact of privacy-preserving federated learning. We exclude the (D) column from Table IV for the sake of space; the pattern is similar to Table III and POSEIDON's accuracy loss is negligible. To obtain accuracy results for the CIFAR-10 and CIFAR-100 datasets, we simulate POSEIDON in Tensorflow [9] by using its approximated activation functions and a fixed-precision. We observe that the accuracy loss between C1, C2, D, and POSEIDON is $0.9-3\%$ when 32-bits precision is used. For instance, POSEIDON achieves 90.4% training accuracy on the ESR dataset, a performance that is equivalent to a decentralized (D) non-private approach and only slightly lower compared to centralized approaches. Note that the accuracy difference between non-secure solutions and POSEIDON can be further reduced by increasing the number of training iterations, however, we use the same number of iterations for the sake of comparison. Moreover, we remind that CIFAR-100 has 100 class labels (i.e., a random guess baseline of 1% accuracy) and is usually trained with special NN structures (ResNet) or special layers (batch normalization) to achieve higher accuracy than the reported ones: we leave these NN types as future work (see Appendix D-B). We use relatively simpler NNs for CIFAR-10 and CIFAR-100 which is the reason for achieving low accuracy on these datasets.

We compare POSEIDON's accuracy with that achieved by one party using its local dataset (L), that is $1/10$ (or $1/50$) of the overall data, with *exact* activation functions. We compute the accuracy for the (L) setting by averaging the test accuracy of the 10 and 50 locally trained models (Tables III and IV, respectively). We observe that even with the accuracy loss due to approximation and encryption, POSEIDON still achieves $1-3\%$ increase in the model accuracy due to privacy-preserving collaboration (Table III). This increase is more significant when the data is partitioned across 50 parties (Table IV) as the number of training samples *per-party* is further reduced and is not sufficient to learn an accurate model.

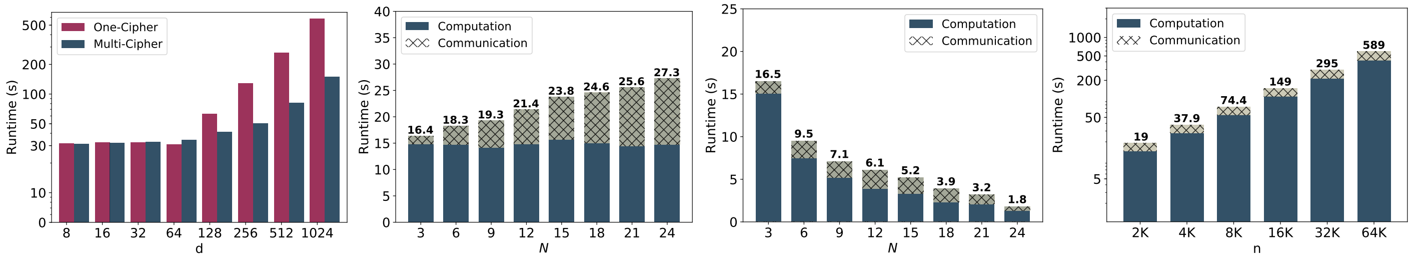
Execution Time. As shown on the right-hand side of Table III, POSEIDON trains the BCW, ESR, and CREDIT datasets in less than 15 minutes and the MNIST in 1.4 hours, when each dataset is evenly distributed among 10 parties. Note that POSEIDON's overall training time for MNIST is less than an hour when the dataset is split among 20 parties that use the same local batch size. We extrapolate the training times of POSEIDON on more complex datasets and architectures for one global iteration (one-GI) in Table IV; these can be used to estimate the training times of these structures with a larger number of global iterations. For instance, CIFAR-10 is trained in 175 hours with 2CV, 2 pooling, 2 FC layers and with dropouts (adding one more multiplication in the dropout layer). Note that it is possible to increase the accuracy with higher run-time or fine-tuned architectures, but we aim at finding a trade-off between accuracy and run-time. For example, POSEIDON's accuracy on SVHN reaches 75% by doubling the training epochs and thus its execution time. The per-sample inference times presented in Tables III and IV include the forward pass, the $\mathsf{DKeySwitch}(\cdot)$ operations that re-encrypt the result with the querier's public key, and the communication among the parties. We note that as all the parties keep the model in encrypted form, any of them can process the prediction query. Hence, taking the advantage of parallel query executions and multi-threading, POSEIDON achieves a throughput of 864,000 predictions per hour on the MNIST dataset with the chosen NN structure.

Dataset	Accuracy				Execution time (hrs)		
	C1	C2	L	POSEIDON	One-GI	Training	Inference
SVHN	68.4%	68.1%	35.1%	67.8%	0.0034	61.2	8.89×10^{-5}
CIFAR-10-N1	54.6%	52.1%	26.8%	51.8%	0.007	175	0.001
CIFAR-10-N2	63.6%	62.0%	28.0%	61.1%	0.011	184.8	0.004
CIFAR-100	43.6%	41.8%	8.2%	41.1%	0.026	1404	0.006

TABLE IV: POSEIDON*'s accuracy and execution times for* $N=50$ *parties (extrapolated). One-GI indicates the execution time of one global iteration.*

Scalability. Figure 1a shows the scaling of POSEIDON with the number of features (d) when the one-cipher and multi-cipher with parallelization approaches are used for a 2-layer NN with 64 hidden neurons. The runtime refers to one epoch, i.e., a processing of all the data from $N=10$ parties, each having 2,000 samples, and employing a batch size of $b=10$. For small datasets with a number of features between 1 and 64, we observe no difference in execution time between the one-cipher and multi-cipher approaches. This is because the weight matrices between layers fit in one ciphertext with $\mathcal{N}=2^{13}$. However, we observe a larger runtime of the one-cipher approach when the number of features increases further. This is because each power-of-two increase in the number of features requires an increase in the cryptographic parameters, thus introducing overhead in the arithmetic operations.

We further analyse POSEIDON's scalability with respect to the number of parties (N) and the number of total samples in the distributed dataset (n), for a fixed number of features. Figures 1b and 1c display POSEIDON's execution time, when the number of parties ranges from 3 to 24, and one training epoch is performed, i.e., all the data of the parties is processed once. For Figure 1b, we fix the number of data samples per party to 200 to study the effect of an increasing number of members in the federation. We observe that POSEIDON's execution time is almost independent of N and is affected only by increasing communication between the parties. When we fix the global number of samples (n), increasing N results in a runtime decrease, as the samples are processed by the parties in parallel (see Figure 1c). Then, we evaluate POSEIDON's runtime with an increasing number of data samples and a fixed number of parties $N=10$, in Figure 1d. We observe that POSEIDON scales linearly with the number of data samples. Finally, we remark that POSEIDON also scales proportionally with the number of layers in the NN structure, if these are all of the same type, i.e, FC, CV, or pooling, and if the number of neurons per layer or the kernel size is fixed.



(a) Increasing number of features (d), $N = 10$, and $n = 2{,}000 * N$.

(b) Increasing number of parties (N), each having 200 samples (n).

(c) Increasing number of parties (N) when n (number of data samples) is fixed to 600.

(d) Increasing number of data samples (n) when N (number of parties) is fixed to 10.

Figure 1: Poseidon*'s training execution time and communication overhead with increasing number of parties, features, and samples, for 1 training epoch.*

F. Comparison with Prior Work

A quantitative comparison of our work with the state-of-the-art solutions for privacy-preserving NN executions is a non-trivial task. Indeed, the most recent cryptographic solutions for privacy-preserving machine learning in the N-party setting, i.e., Helen [119] and SPINDLE [41], support the functionalities of only regularized [119] and generalized [41] linear models respectively. We provide a detailed qualitative comparison with the state-of-the-art privacy-preserving deep learning frameworks in Table V in Appendix and expand on it here.

Poseidon operates in a federated learning setting where the parties maintain their data locally. This is a substantially different setting compared to that envisioned by MPC-based solutions [83], [82], [110], [111], [26], [27], for privacy-preserving NN training. In these solutions, the parties' data has to be communicated (i.e., secret-shared) outside their premises, and the data and model confidentiality is preserved as long as there exists an honest majority among a limited number of computing servers (typically, 2 to 4, depending on the setting). Hence, a similar experimental setting is hard to achieve. Nonetheless, we compare Poseidon to SecureML [83], SecureNN [110], and FALCON [111], when training a 3-layer NN with 128 neurons per layer for 15 epochs, as described in [83], on the MNIST dataset. We set $N = 3$ to simulate a similar setting and use Poseidon's approximated activation functions. Poseidon trains MNIST in 73.1 hours whereas SecureML with 2-parties, SecureNN and FALCON with 3-parties, need 81.7, 1.03, and 0.56 hours, respectively. Depending on the activation functions, SecureML yields $93.1 - 93.4\%$ accuracy, SecureNN 93.4%, and FALCON 97.4%. Poseidon achieves 92.5% accuracy with the approximated SmoothReLU and 96.2% with approximated tanh activation functions. We remind that Poseidon operates under a different system (federated learning-based) and threat model, it supports more parties, and scales linearly with N whereas MPC solutions are based on outsourced learning with limited number of computing servers.

Federated learning approaches based on differential privacy (DP), e.g., [67], [101], [76], train a NN while introducing some noise to the intermediate values to mitigate adversarial inferences. However, training an accurate NN model with DP requires a high privacy budget [96], hence it remains unclear what privacy protection is obtained in practice [55]. We note that DP-based approaches introduce a different tradeoff than Poseidon: they tradeoff privacy for accuracy, while Poseidon decouples accuracy from privacy and tradeoffs accuracy for complexity (i.e., execution time and communication overhead). Nonetheless and as an example, we compare Poseidon's accuracy results with those reported by Shokri and Shmatikov [101] on the MNIST dataset. We focus on their results with the distributed selective SGD configured such that participants download/upload *all* the parameters from/to the central server in each training iteration. We evaluate the same CNN structure used in [101], but with Poseidon's approximated activation functions and average-pooling instead of max-pooling. We compare the accuracy results presented in [101, Figure 13] with $N = 30, N = 90$, and $N = 150$ participants. In all settings, Poseidon yields $> 94\%$ accuracy whereas [101] achieves similar accuracy only when the privacy budget per parameter is ≥ 10. For more private solutions, where the privacy budget is 0.001, 0.01 or 0.1, [101] achieves $\leq 90\%$ accuracy; smaller ϵ yields better privacy but degrades utility.

Finally, existing HE-based solutions [51], [85], [109], focus on a centralized setting where the NN learning task is outsourced to a central server. These solutions, however, employ non-realistic cryptographic parameters [109], [85], and their performance is not practical [51] due to their costly homomorphic computations. Our system, focused on a federated learning-based setting and a multiparty homomorphic encryption scheme, improves the response time 3 to 4 orders of magnitude. The execution times produced by Nandakumar et al. [85] for processing one batch of 60 samples in a single thread and 30 threads for a NN structure with $d = 64$, $h_1 = 32$, $h_2 = 16$, $h_3 = 2$, are respectively 33,840s and 2,400s. When we evaluate the same setting, but with $N = 10$ parties, we observe that Poseidon processes the same batch in 6.3s and 1s, respectively. We also achieve stronger security guarantees (128 bits) than [85] (80 bits). Finally, for a NN structure with 2-hidden layers of 128 neurons each, and the MNIST dataset, CryptoDL [51] processes a batch with $B = 192$ in 10,476.3s, whereas our system in the distributed setting processes the same batch in 34.7s.

Therefore, Poseidon is the only solution that performs both training and inference of NNs in an N-party setting, yet protects data and model confidentiality withstanding collusions up to $N - 1$ parties.

VIII. Conclusion

In this work, we presented Poseidon, a novel system for zero-leakage privacy-preserving federated neural network learning among N parties. Based on lattice-based multiparty homomorphic encryption, our system protects the confidentiality of the training data, of the model, and of the evaluation data, under a passive adversary model with collusions of up to $N - 1$ parties. By leveraging on packing strategies and an extended distributed bootstrapping functionality, Poseidon is the first system demonstrating that secure federated learning on neural networks is practical under multiparty homomorphic encryption. Our experimental evaluation shows that Poseidon significantly improves on the accuracy of individual local training, bringing it on par with centralized and decentralized non-private approaches. Its computation and communication overhead scales linearly with the number of parties that participate in the training, and is between 3 to 4 orders of magnitude faster than equivalent centralized outsourced approaches based on traditional homomorphic encryption. This work opens up the door of practical and secure federated training in passive-adversarial settings. Future work involves extensions to other scenarios with active adversaries and further optimizations to the learning process.

Acknowledgment

We would like to thank all of those who reviewed the manuscript, in particular Sylvain Chatel and the anonymous reviewers. This work was partially supported by the grant #2017-201 of the Strategic Focal Area "Personalized Health and Related Technologies (PHRT)" of the ETH Domain.

References

[1] Microsoft SEAL (release 3.3). https://github.com/Microsoft/SEAL. (Accessed: 2021-01-06).

[2] Centers for Medicare & Medicaid Services. The Health Insurance Portability and Accountability Act of 1996 (HIPAA). https://www.cms.gov/Regulations-and-Guidance/Administrative-Simplification/HIPAA-ACA/PrivacyandSecurityInformation. (Accessed: 2021-01-06).

[3] Convolutional Neural Networks. https://cs231n.github.io/convolutional-networks/. (Accessed: 2021-01-06).

[4] Cothority network library. https://github.com/dedis/onet. (Accessed: 2021-01-06).

[5] Data61. MP-SPDZ - Versatile framework for multi-party computation. https://github.com/data61/MP-SPDZ. (Accessed: 2021-01-06).

[6] Go Programming Language. https://golang.org. (Accessed: 2021-01-06).

[7] The EU General Data Protection Regulation. https://gdpr-info.eu/. (Accessed: 2021-01-06).

[8] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *ACM CCS*, 2016.

[9] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[10] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad. State-of-the-art in artificial neural network applications: A survey. *Elsevier Heliyon*, 4(11):e00938, 2018.

[11] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), July 2018.

[12] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón. QUOTIENT: Two-party secure neural network training and prediction. *ACM CCS*, 2019.

[13] A. Akavia, H. Shaul, M. Weiss, and Z. Yakhini. Linear-regression on packed encrypted data in the two-server model. In *ACM WAHC*, 2019.

[14] M. Albrecht et al. Homomorphic Encryption Security Standard. Technical report, HomomorphicEncryption.org, 2018.

[15] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9:169 – 203, 2015.

[16] S. V. Algesheimer J., Camenisch J. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *CRYPTO*, 2002.

[17] Y. Aono, T. Hayashi, L. Trieu Phong, and L. Wang. Scalable and secure logistic regression via homomorphic encryption. In *ACM CODASPY*, 2016.

[18] Breast cancer wisconsin (original). https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original). (Accessed: 2021-01-06).

[19] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *ACM WAHC*, 2019.

[20] F. Boemer, Y. Lao, and C. Wierzynski. ngraph-he: A graph compiler for deep learning on homomorphically encrypted data. *CoRR*, abs/1810.10121, 2018.

[21] D. Bogdanov, L. Kamm, S. Laur, and V. Sokk. Rmind: a tool for cryptographically secure statistical analysis. *IEEE TDSC*, 15(3):481–495, 2018.

[22] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. Practical secure aggregation for federated learning on user-held data. In *NIPS PPML Workshop*, 2016.

[23] C. Bonte and F. Vercauteren. Privacy-preserving logistic regression training. *BMC Medical Genomics*, 11, 2018.

[24] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. Cryptology ePrint Archive, Report 2020/1203, 2020.

[25] P. Bunn and R. Ostrovsky. Secure two-party k-means clustering. In *ACM CCS*, 2007.

[26] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. *PETS*, 2020.

[27] H. Chaudhari, R. Rachuri, and A. Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS*, 2020.

[28] T. Chen and S. Zhong. Privacy-preserving backpropagation neural network learning. *IEEE Transactions on Neural Networks*, 20(10):1554–1564, Oct 2009.

[29] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

[30] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, and K. Lee. Numerical method for comparison on homomorphically encrypted numbers. In *ASIACRYPT*, 2019.

[31] H. Cho, D. Wu, and B. Berger. Secure genome-wide association analysis using multiparty computation. *Nature Biotechnology*, 36:547–551, 2018.

[32] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2006.

[33] H. Corrigan-Gibbs and D. Boneh. Prio: Private, Robust, and Computation of Aggregate Statistics. In *USENIX NSDI*, 2017.

[34] J. L. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup. Doing real work with fhe: The case of logistic regression. In *ACM WAHC*, 2018.

[35] A. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. *PETS*, 2020.

[36] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*. 2012.

[37] Epileptic Seizure Recognition Dataset. https://archive.ics.uci.edu/ml/datasets/Epileptic+Seizure+Recognition. (Accessed: 2021-01-06).

[38] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012.

[39] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM CCS*, 2015.

[40] D. Froelicher, P. Egger, J. Sousa, J. L. Raisaro, Z. Huang, C. Mouchet, B. Ford, and J.-P. Hubaux. Unlynx: A decentralized system for privacy-conscious data sharing. *PETS*, 2017.

[41] D. Froelicher, J. R. Troncoso-Pastoriza, A. Pyrgelis, S. Sav, J. S. Sousa, J.-P. Bossuat, and J.-P. Hubaux. Scalable privacy-preserving distributed learning. *PETS*, 2021.

[42] D. Froelicher, J. R. Troncoso-Pastoriza, J. S. Sousa, and J. Hubaux. Drynx: Decentralized, secure, verifiable system for statistical queries and machine learning on distributed datasets. *IEEE TIFS*, 15:3035–3050, 2020.

[43] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. Privacy-preserving distributed linear regression on high-dimensional data. *PETS*, 2017.

[44] I. Giacomelli, S. Jha, M. Joye, C. D. Page, and K. Yoon. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In *Springer ACNS*, 2018.

[45] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[46] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

[47] L. Gomes. Quantum computing: Both here and not here. *IEEE Spectrum*, 55(4):42–47, 2018.

[48] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[49] S. Halevi and V. Shoup. HElib - An Implementation of homomorphic encryption. https://github.com/shaih/HElib/. (Accessed: 2021-01-06).

[50] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *IEEE ICCV*, 1502, 2015.

[51] E. Hesamifard, H. Takabi, M. Ghasemi, and R. Wright. Privacy-preserving machine learning as a service. *PETS*, 2018.

[52] B. Hie, H. Cho, and B. Berger. Realizing private and practical pharmacological collaboration. *Science*, 362(6412):347–350, 2018.

[53] B. Hitaj, G. Ateniese, and F. Perez-Cruz. Deep models under the GAN: Information leakage from collaborative deep learning. In *ACM CCS*, 2017.

[54] G. Jagannathan and R. N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *ACM SIGKDD*, 2005.

[55] B. Jayaraman and D. Evans. Evaluating differentially private machine learning in practice. In *USENIX Security*, 2019.

[56] B. Jayaraman, L. Wang, D. Evans, and Q. Gu. Distributed learning without distress: Privacy-preserving empirical risk minimization. In *NIPS*, 2018.

[57] Y. Jiang, J. Hamer, C. Wang, X. Jiang, M. Kim, Y. Song, Y. Xia, N. Mohammed, M. N. Sadat, and S. Wang. Securelr: Secure logistic regression model via a hybrid cryptographic protocol. *IEEE/ACM TCBB*, 2019.

[58] N. P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. *ACM/IEEE ISCA*, 2017.

[59] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. Gazelle: A low latency framework for secure neural network inference. *USENIX Security*, 2018.

[60] Why we shouldn't disregard the NDA. https://www.keystonelaw.com/keynotes/why-we-shouldnt-disregard-the-nda. (Accessed: 2021-01-06).

[61] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC medical genomics*, 2018.

[62] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR Medical Informatics*, 6(2):e19, 2018.

[63] J. Konečnỳ, H. B. McMahan, D. Ramage, and P. Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *CoRR*, abs:1610.02527, 2016.

[64] J. Konecný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. Federated learning: Strategies for improving communication efficiency. *CoRR*, abs/1610.05492, 2016.

[65] A. Krizhevsky. Learning multiple layers of features from tiny images. *Technical Report, University of Toronto*, 2012.

[66] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[67] W. Li, F. Milletarì, D. Xu, N. Rieke, J. Hancox, W. Zhu, M. Baust, Y. Cheng, S. Ourselin, M. J. Cardoso, and A. Feng. Privacy-preserving federated brain tumour segmentation. In *Springer MLMI*, 2019.

[68] X. Li, K. Huang, W. Yang, S. Wang, and Z. Zhang. On the convergence of FedAvg on non-IID data. In *ICLR*, 2020.

[69] X. Lian, W. Zhang, C. Zhang, and J. Liu. Asynchronous decentralized parallel stochastic gradient descent. In *ICML*, 2018.

[70] Y. Lindell. How to simulate it–a tutorial on the simulation proof technique. In *Springer Tutorials on the Foundations of Cryptography*. 2017.

[71] R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In *Springer Topics in Cryptology*, 2011.

[72] Why NDAs often don't work when expected to do so and what to do about it. https://www.linkedin.com/pulse/why-ndas-often-dont-work-when-expected-do-so-what-martin-schweiger. (Accessed: 2021-01-06).

[73] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM CCS*, 2017.

[74] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, 2010.

[75] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas. Federated learning of deep networks using model averaging. *CoRR*, abs/1602.05629, 2016.

[76] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang. Learning differentially private recurrent language models. In *ICLR*, 2018.

[77] Top 15 deep learning applications that will rule the world in 2018 and beyond. https://medium.com/breathe-publication/top-15-deep-learning-applications-that-will-rule-the-world-in-2018-and-beyond-7c6130c43b01. (Accessed: 2021-01-06).

[78] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *IEEE S&P*, 2019.

[79] Lattigo: A library for lattice-based homomorphic encryption in go. https://github.com/ldsec/lattigo. (Accessed: 2021-01-06).

[80] Mininet. http://mininet.org. (Accessed: 2021-01-06).

[81] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*, 2020.

[82] P. Mohassel and P. Rindal. Aby 3: a mixed protocol framework for machine learning. In *ACM CCS*, 2018.

[83] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, 2017.

[84] C. Mouchet, J. R. Troncoso-pastoriza, J.-P. Bossuat, and J. P. Hubaux. Multiparty homomorphic encryption: From theory to practice. In *Technical Report https://eprint.iacr.org/2020/304*, 2019.

[85] K. Nandakumar, N. Ratha, S. Pankanti, and S. Halevi. Towards deep neural network training on encrypted data. In *IEEE CVPR Workshops*, 2019.

[86] M. Nasr, R. Shokri, and A. Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE S&P*, 2019.

[87] C. Neill et al. A blueprint for demonstrating quantum supremacy with superconducting qubits. *Science*, 2018.

[88] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Ng. Reading digits in natural images with unsupervised feature learning. *NIPS*, 2011.

[89] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE S&P*, 2013.

[90] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[91] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[92] A. Patra and A. Suresh. Blaze: Blazing fast privacy-preserving machine learning. In *NDSS*, 2020.

[93] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai. Privacy-preserving deep learning: Revisited and enhanced. In *Springer ATIS*, 2017.

[94] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE TIFS*, 13(5):1333–1345, 2018.

[95] L. Prechelt. Early stopping - but when? In *Springer Neural Networks: Tricks of the Trade*, 1998.

[96] M. A. Rahman, T. Rahman, R. Laganière, and N. Mohammed. Membership inference attack against differentially private deep learning model. *Transactions on Data Privacy*, 11:61–79, 2018.

[97] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. In *USENIX Security*, 2019.

[98] K. Rohloff. The PALISADE Lattice Cryptography Library. https://git.njit.edu/palisade/PALISADE, 2018.

[99] T. P. Schoenmakers B. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *EUROCRYPT*, 2006.

[100] P. Schoppmann, A. Gascon, M. Raykova, and B. Pinkas. Make some room for the zeros: Data sparsity in secure distributed machine learning. In *ACM CCS*, 2019.

[101] R. Shokri and V. Shmatikov. Privacy-preserving deep learning. In *ACM CCS*, 2015.

[102] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. Membership inference attacks against machine learning models. In *IEEE S&P*, 2017.

[103] S. Song, K. Chaudhuri, and A. D. Sarwate. Stochastic gradient descent with differentially private updates. In *IEEE GlobalSIP*, 2013.

[104] I. Stoica, D. Song, R. A. Popa, D. A. Patterson, M. W. Mahoney, R. H. Katz, A. D. Joseph, M. I. Jordan, J. M. Hellerstein, J. E. Gonzalez, K. Goldberg, A. Ghodsi, D. E. Culler, and P. Abbeel. A Berkeley view of systems challenges for AI. *CoRR*, abs/1712.05855, 2017.

[105] B. Terhal. Quantum supremacy, here we come. *Nature Physics*, 14(06), 2018.

[106] Top applications of deep learning across industries. https://www.mygreatlearning.com/blog/deep-learning-applications/. (Accessed: 2021-01-06).

[107] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction APIs. In *USENIX Security*, 2016.

[108] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou. A hybrid approach to privacy-preserving federated learning. In *ACM AISec*, 2019.

[109] A. Vizitu, C. Niţă, A. Puiu, C. Suciu, and L. Itu. Applying deep neural networks over homomorphic encrypted medical data. *Computational and Mathematical Methods in Medicine*, 2020:1–26, 2020.

[110] S. Wagh, D. Gupta, and N. Chandran. Securenn: 3-party secure computation for neural network training. *PETS*, 2019.

[111] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin. FALCON: Honest-majority maliciously secure framework for private deep learning. *PETS*, 2020.

[112] J. Wang and G. Joshi. Cooperative SGD: A unified framework for the design and analysis of communication-efficient SGD algorithms. *CoRR*, abs:1808.07576, 2018.

[113] Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi. Beyond inferring class representatives: User-level privacy leakage from federated learning. In *IEEE INFOCOM*, 2019.

[114] I.-C. Yeh and C. hui Lien. The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients. *Expert Systems with Applications*, 36(2):2473 – 2480, 2009.

[115] L. Yu, L. Liu, C. Pu, M. Gursoy, and S. Truex. Differentially private model publishing for deep learning. In *IEEE S&P*, 2019.

[116] A. Zalcman et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 10 2019.

[117] D. Zhang. Big data security and privacy protection. In *ICMCS*, 2018.

[118] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra. Federated learning with non-IID data. *CoRR*, abs/1806.00582, 2018.

[119] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *IEEE S&P*, 2019.

[120] L. Zhu, Z. Liu, and S. Han. Deep leakage from gradients. In *NIPS*. 2019.

[121] X. Zhu, C. Vondrick, C. C. Fowlkes, and D. Ramanan. Do we need more training data? *Springer IJCV*, 119(1):76–92, Aug. 2016.

[122] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *NIPS*, 2010.

Appendix A
Approximation of the Max/Min Pooling and Its Derivative

For the sake of clarity, we describe the max-pooling operation. Given a vector $x=(x[0],...,x[n-1])$ the challenge is to compute y with $y[0\leq i<n]=\max(x)$. To approximate the index of $\max(x)$, which can then be used to extract the max value of x, we follow an algorithm similar to that presented in [30], described below.

Given two real values a,b, with $0\leq a,b\leq 1$, we observe the following: If $a>b$, then $a-b<a^d-b^d$ for $d>1$, i.e., with increasing d, smaller values converge to zero faster and the ratio between the maximum value and other values increases. The process can be repeated to increase the ratio between a and b but, unless $a=1$, both values will eventually converge to zero. To avoid this, we add a second step that consists in renormalizing a and b by computing $a=a/(a+b)$ and $b=b/(a+b)$. Thus, we ensure that after each iteration, $a+b=1$ and since b will eventually converge to zero, a will tend towards 1. If $a=b$, both values will converge to 0.5. This algorithm can be easily generalized to vectors: Given a vector $x=(x[0],...,x[n-1])$, at each iteration it computes $x[i]=x[i]^d/\sum_{j=0}^{n-1}x[j]^d$, and multiplies the result with the original vector to extract the maximum value.

This max-pooling algorithm is a time-consuming procedure as it requires computing an expensive inverse function, especially if a high accuracy is desired or if the input values are very small. Instead, we employ a direct approach using $\max(a,b)=\frac{1}{2}(a+b+\sqrt{(a-b)^2})$, where the square-root can be approximated by a polynomial. To compute the maximum value for a kernel $f=k\times k$, we iterate $\log(f)$ times $\boldsymbol{c}_{i+1}=\max(\boldsymbol{c}_i,\mathsf{RotL}_{2^i}(\boldsymbol{c}_i))$. As each iteration consumes all levels, we use $\mathsf{DBootstrap}(\cdot)$ $\log(f)$ times. Hence, we suggest using the average-pooling instead, which is more efficient and precise, e.g., Dowlin et al. [45] show that low-degree approximations of max-pooling will converge to a scalar multiple of the mean of k values. We provide microbenchmarks of both max and average-pooling in Appendix C.

Appendix B
Technical details of Distributed Bootstrapping with Arbitrary Linear Transformations ($\mathsf{DBootstrapALT}(\cdot)$)

A linear transformation $\phi(\cdot)$ over a vector of n elements can be described by a $n\times n$ matrix. The evaluation of a matrix-vector multiplication requires a number of rotations proportional to the square-root of its non-zero diagonals, thus, this operation becomes prohibitive when the number of non-zero diagonals is large.

Such a linear transformation can be, however, efficiently carried out *locally* on a secret-shared plaintext, as $\phi(msg+M)=\phi(msg)+\phi(M)$ due to the linearity of $\phi(\cdot)$. Moreover, because of the magnitude of $msg+M$ (100 to 200 bits), arbitrary precision complex arithmetic with sufficient precision should be used for $\mathsf{Encode}(\cdot)$, $\mathsf{Decode}(\cdot)$, and $\phi(\cdot)$ to preserve the lower bits. The collective bootstrapping protocol in [84] is performed through a conversion of an encryption to secret-shared values and a re-encryption in a refreshed ciphertext. We leverage this conversion to perform the aforementioned linear transformation in the secret-shared domain, before the refreshed ciphertext is reconstructed. This is our $\mathsf{DBootstrapALT}(\cdot)$ protocol (Protocol 4).

When the linear transformation is simple, i.e., it does not involve a complex permutation or requires a small number of rotations, the $\mathsf{Encode}(\cdot)$ and $\mathsf{Decode}(\cdot)$ operations in Line 8, Protocol 4 can be skipped. Indeed, those two operations are carried out using arbitrary precision complex arithmetic. In such cases, it is more efficient to perform the linear transformation directly on the encoded plaintext.

Security Analysis of $\mathsf{DBootstrapALT}(\cdot)$. This protocol is a modification of the $\mathsf{DBootstrap}(\cdot)$ protocol of Mouchet et al. [84], with the difference that it includes a product of a public matrix. Both $\mathsf{DBootstrap}(\cdot)$ and $\mathsf{DBootstrapALT}(\cdot)$ for CKKS differ from the BFV version proposed in [84] in which the shares are not unconditionally hiding, but statistically or computationally hiding due to the incomplete support of the used masks. Therefore, the proof follows analogously the passive adversary security proof of the BFV $\mathsf{DBootstrap}(\cdot)$ protocol in [84], with the addition of Lemma 1 which guarantees the statistical indistinguishablity of the shares in $\mathbb{C}$. While the RLWE problem and Lemma 1 do not rely on the same security assumptions, the first one being computational and the second one being statistical, given the same security parameter, they share the same security bounds. Hence, $\mathsf{DBootstrap}(\cdot)$ and $\mathsf{DBootstrapALT}(\cdot)$ provide the same security as the original protocol of Mouchet et al. [84].

Lemma 1. *Given the distribution $P_0=(a+b)$ and $P_1=c$ with $0\leq a<2^\delta$ and $0\leq b,c<2^{\lambda+\delta}$ and b, c uniform, then the distributions P_0 and P_1 are λ-indistinguishable; i.e., a probabilistic polynomial adversary $\mathcal{A}$ cannot distinguish between them with probability greater than $2^{-\lambda}$: $|Pr[\mathcal{A}\to 1|P=P_1]-Pr[\mathcal{A}\to 1|P=P_0]|\leq 2^{-\lambda}$.*

We refer to Algesheimer et. al [16, Section 3.2], and Schoenmakers and Tuyls [99, Appendix A], for the proof of the statistical λ-indistinguishability.

We recall that an encoded message msg of $\mathcal{N}/2$ complex numbers with the CKKS scheme is an integer polynomial of $\mathbb{Z}[X]/(X^{\mathcal{N}}+1)$. Given that $||msg||<2^\delta$, and a second polynomial M of $\mathcal{N}$ integer coefficients with each coefficient uniformly sampled and bounded by $2^{\lambda+\delta}-1$ for a security parameter λ, Lemma 1 suggests that Pr$[||msg^{(i)}+M^{(i)}||\geq 2^{\lambda+\delta}]\leq 2^{-\lambda}$, for $0\leq i<\mathcal{N}$ and where i denotes the i^{th} coefficient of the polynomial. That is, the probability of a coefficient of $msg+M$ to be distinguished from a uniformly sampled integer in $[0,2^{\lambda+\delta})$ is bounded by $2^{-\lambda}$. Hence, during Protocol 4 each party samples its polynomial mask M with uniform coefficients in $[0,2^{\lambda+\delta})$. The parties, however, should have an estimate of the magnitude of msg to derive δ, and a probabilistic upper-bound for the magnitude can be computed by the circuit and the expected range of its inputs.

In Protocol 4, the masks M_i are added to the ciphertext of R_{Q_ℓ} during the decryption to the secret-shared domain. To avoid a modular reduction of the masks in R_{Q_ℓ} and ensure a correct re-encryption in R_{Q_L}, the modulus Q_ℓ should be large enough for the additions of N masks. Therefore, the ciphertext modulus size should be greater than $(N+1)\cdot||M||$ when the bootstrapping is called. For example, for $N=10$, a Q_L composed of a 60 bits modulus, a message msg with $||msg||<2^{55}$ (taking the scaling factor Δ into account) and

		XONN [97]	**Gazelle** [59]	**Blaze** [92]	**MiniONN** [73]	**ABY3** [82]	**SecureML** [83]	**SecureNN** [110]	**FALCON** [111]	**FLASH** [26]	**TRIDENT** [27]	**CryptoNets** [45]	**CryptoDL** [51]	[85]	**POSEIDON**
MPC Setup		2PC	2PC	3PC	2PC	3PC	2PC	3PC	3PC	4PC	4PC	1PC	1PC	1PC	**N-Party**
Private Infer.		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Private Train.		✗	✗	✗	✗	✔	✔	✔	✔	✔	✔	✗	✔	✔	✔
Data Conf.															
Adversarial Model*		1 P	1 P	1 A	1 P	1 A/P	1 P	1 A/P	1 A/P	1 A	1 A/P	1 P’	1 P’	1 P’	$N-1$ P
Collusion*		No	No	No	No	No	No	No	No	No	No	NA	NA	NA	$N-1$
Techniques		GC,SS	HE,GC,SS	GC,SS	HE,GC,SS	GC,SS	HE,GC,SS	SS	SS	SS	GC,SS	HE	HE	HE	HE
Supported Layers	Linear	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Conv.	✔	✔	✗	✔	✔	✔	✔	✔	✔	✔	✔	✗	✗	✔
	Pooling	✔	✔	✗	✔	✔	✔	✔	✔	✔	✔	✔	✗	✗	✔

TABLE V: *Qualitative comparison of private deep learning frameworks. Conf. stands for confidentiality. A and P respectively stand for active and passive adversarial capabilities. GC, SS, HE denote garbled-circuits, secret sharing, and homomorphic encryption. Adversarial model* and collusion* take into account the servers responsible for the training/inference.* 1 *P’ denotes our interpretation, as [45], [85], and [51] do not present an adversarial model. NA stands for not applicable.*

$\lambda = 128$, we should have $||M_i|| \geq 2^{183}$ and $Q_\ell > 11 \cdot 2^{183}$. Hence, the bootstrap should be called at Q_3 because $Q_2 \approx 2^{180}$ and $Q_3 \approx 2^{240}$. Although the aforementioned details suggest that $\mathsf{DBootstrapALT}(\cdot)$ is equivalent to a depth 3 to 4 circuit, depending on the parameters, it is still compelling, as it enables us to refresh a ciphertext and apply an arbitrary complex linear transformation at the same time. Thus, its cost remains negligible compared to a centralized bootstrapping where any transformation is applied via rotations.

APPENDIX C
MICROBENCHMARKS

We present microbenchmark timings for the various functionalities and sub-protocols of POSEIDON in Table VI. These are measured in an experimental setting with $N = 10$ parties, a dimension of $d = 32$ features, $h = 64$ neurons in a layer or kernel size $k = 3 \times 3$, and degree $d_a = 3$ for the approximated activation functions for FC, CV, FC backpropagation, CV backpropagation, and average-pooling benchmarks. These benchmarks represent the processing of 1 sample per party, thus $b = 1$. For max-pooling, we achieve a final precision of 7 bits with a square-root approximated by a Chebyshev interpolant of degree $d_a = 31$. We observe that max-pooling is 6 times slower than average-pooling, has a lower precision, and needs more communication due to the large number of $\mathsf{DBootstrap}(\cdot)$ operations. For 12-bits precision, max-pooling takes 4.72s. This supports our choice of using average-pooling instead of max-pooling in the encrypted domain. The communication column shows the overall communication between the parties in MB. As several HE-based solutions [45], [59], [51], use square activation functions, we also benchmark them and compare them with the approximated activation functions with $d_a = 3$.

We note that **PREPARE** stands for the offline phase and it incorporates the collective generation of the encryption, decryption, evaluation, and rotation keys based on the protocols presented in [84]. Most of the time and bandwidth are consumed by the generation of the rotation keys needed for the training protocol. We refer the reader to [84], [79] for more information about the generation of these keys. Although we present the **PREPARE** microbenchmark to hint about the execution time and communication overhead of this offline phase, we note that it is a non-trivial task to extrapolate its costs for a generic neural network structure. **REDUCE** indicates the reducing step for 1 weight matrix (updating the weight matrix in root) and collectively refreshing it.

Functionality	Execution time (s)	Comm. (MB)
ASigmoid/ASmoothRelu	0.050	-
ASigmoidD/ASmoothReluD	0.022	-
Square / SquareD	0.01 / 0.006	-
ASoftmax	0.07	-
$\mathsf{DBootstrap}(\cdot)$	0.09	6.5
$\mathsf{DBootstrapALT}(\cdot)$ ($\log_2(h)$ rots)	0.18	6.5
$\mathsf{DBootstrapALT}(\cdot)$ with Average Pool	0.33	6.5
MaxPooling	2.08	19.5
FC layer / FC layer-backprop	0.09 / 0.13	-
CV layer / CV layer-backprop	0.03 / 0.046	-
DKeySwitch	0.07	23.13
PREPARE (offline)	18.19	3.8k
MAP (only communication)	0.03	18.35
COMBINE	0.09	7.8
REDUCE	0.1	6.5

TABLE VI: *Microbenchmarks of different functionalities for* $N = 10$ *parties,* $d{=}32$, $h{=}64$, $\mathcal{N}{=}2^{13}$, $d_a{=}3$, $k{=}3{\times}3$.

We show how to use these microbenchmarks to *roughly* estimate the *online* execution time and communication overhead of one global iteration for a chosen neural network structure. We combine the results of Table VI for layers/kernels with specific size, fixed $\mathcal{N}$, d_a, and N, with those of Table II that show POSEIDON’s linear scalability with N for the operations requiring communication, linear scalability with $\mathcal{N}$, and logarithmic scalability with d. We scale the execution time of each functionality for the various parameters depending on the theoretical complexity. Here exemplify the time for computing one global iteration with $N = 50$ parties, for a CNN with 32×32 input images, 1 CV layer with kernel size $k = 6 \times 6$, 1 average-pooling layer with $k = 3 \times 3$, and 1 FC layer with $h = 128$ neurons. We observe that the number of parties N is 5 times bigger than the setting of Table VI, thus yields one round of communication of **MAP** and **COMBINE** as $0.03 \times 5 = 0.15$s and $0{,}09 \times 5 = 0.45$s, respectively. The **REDUCE** microbenchmark is calculated for 1 weight matrix, thus with 2 weight matrices, **REDUCE** will consume 0.2s. For the LGD computation, we start with the CV layer with $k = 6 \times 6$ kernel size. We remind that CV layers are represented by FC layers, thus the kernel size affects the run-time logarithmically; we multiply the CV layer execution time by 2 ($0.03 \times 2 = 0.06$) followed by an activation execution time of 0.05s. For more than 1 filter per CV layer, this number should be multiplied by the number of filters (assuming no parallelization). Then, we use $\mathsf{DBootstrapALT}(\cdot)$ with average pooling to refresh the ciphertext, compute the pooling together with the backpropagation values yielding an execution time of 0.33s scaled to 50 parties as $0.33 \times 5 = 1.65$s. Lastly, since its execution time scales logarithmically with the number of neurons, the FC layer will be executed in $0.09/\log_2(64) * \log_2(128) = 0.105$s followed by another activation of 0.05s. A similar approach is then used for the backward pass and with FC layer-backprop, CV layer-backprop, and using the derivatives of the activation functions. The microbenchmarks are calculated using 1 sample per-party; thus, to extrapolate the time for $b > 1$ *without any parallelization*, the total time for the forward and backward passes should be multiplied by b. Finally, as this example is a CNN, we already refresh the ciphertexts after each CV layer both in the forward and backward pass, to compute pooling or to re-arrange the slots. To extrapolate the times for MLPs, the number of bootstrappings are calculated as described in Section V-F and this is multiplied by the $\mathsf{DBootstrap}(\cdot)$ benchmark. Extrapolating the communication overhead for a global iteration is straightforward: As the communication scales linearly with the number of parties, we scale the overheads given in Table VI with N. For example, with $N = 50$ parties, **MAP** and **COMBINE** consume $18.32 \times 5 = 91.6$MB and $7.8 \times 5 = 39$MB, respectively. Similarly, the total number of $\mathsf{DBootstrap}(\cdot)$, and its variants, should be multiplied by 5. Lastly, in this example, the weight or kernel matrices fit in one ciphertext ($\mathcal{N}/2 = 4{,}096$ slots); if more than 1 cipher per weight matrix is needed, the aforementioned numbers should be multiplied by the number of ciphertexts.

Appendix D
Extensions

We introduce here several security, learning, and optimization extensions that can be integrated to POSEIDON.

A. Security Extensions

Active Adversaries. POSEIDON preserves the privacy of the parties under a passive-adversary model with up to $N - 1$ colluding parties, motivated by the cooperative federated learning scenario presented in Sections I and IV-A.Our work could be extended to an active-adversarial setting by using standard verifiable computation techniques, e.g., resorting to zero-knowledge proofs and redundant computation. This would, though, come at the cost of an increase in the computational complexity, that will be analyzed as future work.

Out-of-the-Scope Attacks. We briefly discuss here out-of-the-scope attacks and countermeasures. By maintaining the intermediate values of the learning process and the final model weights under encryption, during the training process, we protect data and model confidentiality. As such, POSEIDON protects against federated learning attacks [86], [78], [53], [120], [113]. Nonetheless, there exist inference attacks that target the outputs of the model's predictions, e.g., membership inference [102], model inversion [39], or model stealing [107]. Such attacks can be mitigated via complementary countermeasures that can be easily integrated to POSEIDON: (i) limiting the number of prediction queries for the queriers, and (ii) adding noise to the prediction's output to achieve differential privacy guarantees. The choice of the differential privacy parameters in this setting remains an interesting open problem.

B. Learning Extensions

Availability, Data Distribution, and Asynchronous Distributed Neural Networks. In this work, we rely on a multiparty cryptographic scheme that assumes that the parties are always available. We here note that POSEIDON can support asynchronous distributed neural network training [36] without waiting for all parties to send the local gradients. As such, a time threshold could be used for updating the global model. However, we note that the collective cryptographic protocols (e.g., $\mathsf{DBootstrap}(\cdot)$ and $\mathsf{DBootstrapALT}(\cdot)$) require that all the parties be available. Changing POSEIDON's distributed bootstrapping with a centralized one that achieves a practical security level would require increasing the size of the ciphertexts and result in higher computation and communication overhead.

For the evaluation of POSEIDON, we evenly distribute the dataset across the parties; we consider the effects of uneven distributions or the asynchronous gradient descent to the model accuracy — which are studied in the literature [69], [36], [112] — orthogonal to this work. However, a preliminary analysis with the MNIST dataset and the NN structure defined in our evaluation (see Section VII) shows that asynchronous learning decreases the model accuracy between 1 and 4% when we assume that a server is down with a failure probability between 0.4 and 0.8, i.e., when there is between 40 and 80% chance of not receiving the local gradients from a server in a global iteration. Finally, we find that the uneven distribution of the MNIST dataset for $N = 10$ parties with one party holding 90% of the data results to a 6% decrease in the model accuracy. Lastly, we note that the non-iid distribution of the data in federated learning settings causes weight/parameter divergence [68], [118]. The proposed mitigation techniques, however, do not change the working principle of POSEIDON and can be seamlessly integrated, e.g., by adjusting hyperparameters [68] or creating and globally sharing a set of data with uniform distribution among the participants [118].

Other Neural Networks. In this work, we focus on the training of MLPs and CNNs and present our packing scheme and cryptographic operations for these neural networks. For other structures, e.g., long short-term memory (LSTM), recurrent neural networks (RNN), and residual neural networks (ResNet), POSEIDON requires modifications of the LGD-computation phase according to their forward and backward pass operations and of the packing scheme. For example, ResNet has skip connections to jump over some layers, thus the shape of the packed ciphertext after a layer skip should be aligned according to the weight matrix that it is multiplied with. This can be ensured by using the $\mathsf{DBootstrapALT}(\cdot)$ functionality (to re-arrange the slots of the ciphertext). We note that POSEIDON's packing protocols are tailored to MLPs and CNNs and might require adaptation for other neural network structures.

C. Optimization Extensions

Optimizations for Convolutional Neural Networks. We present a scheme for applying the convolutions on the slots, similar to FC layers, by representing them with a matrix multiplication. Convolution on a matrix, however, can be performed with a simple polynomial multiplication by using the coefficients of the polynomial. This operation requires a Fast-Fourier Transform (FFT) from slots (Number Theoretic Transform (NTT)) to coefficients domain, and vice versa (inverseFFT) for switching between CV to pooling or FC layers. Although it achieves better performance for CV layers, domain-switching is expensive. In the case of multiple CV layers before an FC layer, this operation could be embedded into the distributed bootstrapping ($\mathsf{DBootstrapALT}(\cdot)$) for efficiency. The evaluation of the trade-off between the two solutions for larger matrix dimensions is an interesting direction for future work.