Linear-Regression on Packed Encrypted Data in the Two-Server Model

Adi Akavia
akavia@cs.haifa.ac.il
University of Haifa
Haifa, Israel

Hayim Shaul
hayim.shaul@gmail.com
University of Haifa and IDC Herzliya
Haifa, Israel

Mor Weiss
mormorweiss@gmail.com
IDC Herzliya
Herzliya, Israel

Zohar Yakhini
zohar.yakhini@gmail.com
IDC Herzliya and Technion
Herzliya, Israel

ABSTRACT

Developing machine learning models from *federated* training data, containing many independent samples, is an important task that can significantly enhance the potential applicability and prediction power of learned models. Since single users, like hospitals or individual labs, typically collect data-sets that do not support accurate learning with high confidence, it is desirable to combine data from several users without compromising data privacy. In this paper, we develop a *privacy-preserving* solution for learning a linear regression model from data collectively contributed by several parties ("data owners"). Our protocol is based on the protocol of Giacomelli et al. (ACNS 2018) that utilized two non colluding servers and Linearly Homomorphic Encryption (LHE) to learn regularized linear regression models. Our methods use a different LHE scheme that allows us to significantly reduce both the number and runtime of homomorphic operations, as well as the total runtime complexity. Another advantage of our protocol is that the underlying LHE scheme is based on a different (and post-quantum secure) security assumption than Giacomelli et al. Our approach leverages the Chinese Remainder Theorem, and Single Instruction Multiple Data representations, to obtain our improved performance. For a 1000 x 40 linear regression task we can learn a model in a total of 3 seconds for the homomorphic operations, compared to more than 100 seconds reported in the literature. Our approach also scales up to larger feature spaces: we implemented a system that can handle a 1000 x 100 linear regression task, investing minutes of server computing time after a more significant offline pre-processing by the data owners. We intend to incorporate our protocol and implementations into a comprehensive system that can handle secure federated learning at larger scales.

CCS CONCEPTS

• **Security and privacy → Usability in security and privacy**.

KEYWORDS

privacy-preserving machine learning; linear regression; homomorphic encryption; single instruction multiple data; packing; RLWE

ACM Reference Format:
Adi Akavia, Hayim Shaul, Mor Weiss, and Zohar Yakhini. 2019. Linear-Regression on Packed Encrypted Data in the Two-Server Model. In *7th Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC '19), November 11, 2019, London, UK*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3338469.3358942

WAHC '19, November 11, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6829-2/19/11.
https://doi.org/10.1145/3338469.3358942

1 INTRODUCTION

The use of machine-learning (ML) models to support inference has numerous and wide-spread practical implications. Training these models typically requires large volumes of data to support higher confidence statistics. A single data-owner however, such as a health provider clinic or hospital, often has access only to limited data volumes. This motivates sharing data with other data-owners. Such data sharing however poses security risks and may be prohibited, e.g., by regulations or commercial concerns. A vivid field of research is therefore to develop privacy-preserving ML protocols to enable statistical analysis on federated data that simultaneously attain both: (i) Training ML models on large data volumes obtained from multiple data-owners, while (ii) Preserving the privacy of all data-owners. This is known to be possible since the seminal study of secure multi-party computation (MPC) in the 1980's [9, 24]; with much recent efforts on achieving efficiency relevant to practice, as initiated in [1, 7, 10, 13, 15, 17, 18].

Linear-regression is an important and widely-used statistical tool for modeling the relationship between properties of data instances $\vec{x_i} \in \mathbb{R}^d$ (features) and an outcome (response) $y_i \in \mathbb{R}$ using a linear function $y'_i = \vec{w} \cdot \vec{x_i}$. Training a regression model, takes n data instances $(\vec{x_i}, y_i) \in \mathbb{R}^{d+1}$ and returns a model $\vec{w} \in \mathbb{R}^d$ that minimizes the loss function, e.g., the least-square-error (MSE) $\left\|\vec{y'} - \vec{y}\right\|_2^2$. *Ridge-regression* is a prevalent form of regression that reduces over-fitting by adding ℓ_2-regularization $\lambda \|\vec{w}\|_2^2$ to the loss function; See the survey in [16]. Training a ridge regression model, given $(X|y) \in \mathbb{R}^{n\times(d+1)}$, boils down to solving a linear-system $A\vec{w} = \vec{b}$ for $A = X^TX + \lambda I$ and $\vec{b} = X^T\vec{y}$.

A *privacy-preserving ridge regression (PPRR)* is a protocol for training ridge regression models that reveals no information on the training data beyond what can be inferred from the outputted

regression model. A recent line of work initiated by Nikolaenko et al. [19] presented PPRR protocols in the two-server model [14], where the training is conducted by two non-colluding servers. Nikolaenko et al. [19]'s protocol uses Linearly-Homomorphic Encryption (LHE) and Yao's garbled circuit protocol. Giacomelli et al. [6] recently eliminated the need for garbling, and presented a two-server PPRR protocol using only LHE. This has greatly improved overall performance, both in terms of computation and communication. Giacomelli et al. [6] have implemented their protocol using Paillier encryption scheme as the underling LHE to evaluate its concrete performance.

1.1 Our Contribution

Our contribution in this work is as follows.

Linear-regression on Packed Encrypted Data (LoPED). We present a new PPRR-protocol, which we call *"Linear-regression on Packed Encrypted Data (LoPED)"*. Our protocol builds on the method of [6] while modifying the scheme to allow computing on packed data, leading to substantial speedup. "Packing" is in the sense of utilizing the Single Instruction Multiple Data (SIMD) [22] property of RLWE-based schemes [2, 4, 20] to pack hundreds (to thousands) of plaintext data-values in each ciphertext. Our protocol is applicable in the two-server model and uses only LHE, as does [6]. To support SIMD operations, which are not supported when using Paillier encryption [6], we introduce several novel components; See Section 1.2.

Speedup in online time: analytical results. LoPED affords up to $\Theta(d^2)$ speedup over [6] in the number of homomorphic-operations, where d is the number of features. More precisely, the speedup is $\Theta(\mathsf{sl})$, where $\mathsf{sl} \leq d^2$ is the number of slots packed in each SIMD ciphertext. Attaining $\mathsf{sl} \approx d^2$ is essentially for free for up to $d \approx 50$ features, because parameters settings with $2000 - 3000$ slots lead to essentially no complexity overhead in the time of SIMD homomorphic operations (beyond the complexity necessitated by the security and correctness requirements).

Speedup in online time: experimental results. We implemented our protocol and ran extensive experiments demonstrating a substantial speedup in the run-time of homomorphic operations (aka, LHE-comp); See Table 1 (page 12). For example, for $d = 40$ features and $n = 1000$ data instances, LHE-comp takes 102s (seconds) in [6] compared to 3s in LoPED (×33-speedup). The overall speedup in total run-time is from 143s in [6] to 60s in LoPED (×2.4 speedup); "total time" accounts, on top of LHE-comp, also for the time to pre- and post-process the cleartext data, including the time to encrypt and decrypt. Furthermore, while [6] provided run-time results only up to $d = 40$ features (102s spent on LHE-comp), we ran experiments up to $d = 100$ (15s); namely, even when comparing $d = 40$ in [6] to $d = 100$ in LoPED, LoPED's LHE-comp is still ×7 faster.

Security. LoPED is based on a *post-quantum* secure security assumption, in contrast to the Paillier-based implementation used in [6]; this is an added benefit of our system. Security is against a computationally-bounded honest-but-curious adversary that controls at most one of the servers (aka, non-colluding servers) together with any subset of the data-owners. The security guarantee is that corrupted parties (data-owners and servers) learn no new information on the data beyond what is revealed by the output. The latter holds, in both [6] and LoPED, provided that the matrix $A = X^T X + \lambda I$ is invertible in the ring $\mathbb{Z}_N$ into which A was embedded (after appropriate scaling) to allow encryption and homomorphic operations; otherwise server $\mathcal{S}_2$ can distinguish invertible from non-invertible matrices A. We note that in [6], when using Paillier-based LHE, N has two prime divisors. In LoPED, when using RLWE-based LHE together with our CRT representation for the plaintext (see Section 1.2), N has several prime divisors, roughly 40 in our experiments. Nonetheless, in both cases all prime divisors are large ($40-60$ bits long moduli in LoPED), so we expect naturally occurring matrices A to be invertible.

1.2 Our Techniques

To attain run-time speedup and packed-encryption in LoPED we present a variant of [6]'s protocol that achieves the following:

(a) Compatibility with current implementations of RLWE-based LHE. For this purpose we reduce the plaintext size via the Chinese Remainder Theorem (CRT).

(b) Run-time speedup via computation on packed cipehrtexts (SIMD). For this purpose, we employ the SIMD feature available in the RLWE-based LHE schemes. Furthermore, we use Jiang et al. [12]'s protocol for fast matrix-multiplication of packed encrypted matrices, which we (straightforwardly) adapt to our settings where one of the two matrices is given in plaintext, showing that LHE (instead of full-blown fully homomorphic encryption) suffices in this case. In addition, we set the LHE parameters appropriately (when possible) to enforce a roughly optimal number of packed slots ($\mathsf{sl} = d^2$). We also utilize parallel-computing tailored to our use of CRT. We note that SIMD is not known to be available in Paillier. Moreover, CRT cannot be employed to shrink plaintext size in Paillier, because Paillier's security requirement enforces a lower-bound on the plaintext modulus of typically 2048 bit. Next, we elaborate on each of these components.

Compatibility with RLWE-based LHE. Our goal is to use RLWE-based LHE (rather than Paillier in [6]). The motivation is twofold. First, to speedup the homomorphic-computation by utilizing the SIMD feature available in RLWE-based schemes, which is not available for Paillier. Second, to base security on an alternative assumption (RLWE), which is post-quantum secure.

A main challenge we face is that the plaintext moduli required by [6] are incompatible with the current implementations of RLWE-based LHE scheme (e.g., HElib [11] or SEAL [3]). Specifically, for correctness, [6] requires using plaintext modulus $N \sim 2^{d(4\ell+2\log_2 n)}$, which is thousands of bits long for typical parameter settings; See Equation 1 (page 5). For example, for $d = 40$ features, $n = 1000$ data-instances, and $\ell = 3$ decimal-digits of precision, as used in the experiments of [6], N has roughly 2400 bits. The problem is that HElib (similarly, SEAL) supports plaintext moduli only up to 62 (similarly, 60) bits long.

To address this challenge, we use the Chinese Remainder Theorem (CRT).

Plaintext Shrinkage via Chinese Remainder Theorem (CRT). We employ CRT to shrink plaintext space up-to exponentially smaller than in [6]: reducing ring size from roughly $N \sim$

$2^{d(4\ell+2\log_2 n)}$ in [6] to to as low as $p_i \sim d(4\ell + 2\log_2 n)$ in LoPED, albeit using $t = \log_2 N$ rings $\mathbb{Z}_{p_1}, \ldots, \mathbb{Z}_{p_t}$. For example, with parameters $d = 40$, $n = 1000$, $\ell = 10$, we can reduce the moduli magnitudes from roughly 2^{2400} to 2400; equivalently, from 2400 bits to 12 bits. This reduction in plaintext size makes the plaintext compatible for encryption in HElib (similarly, SEAL) LHE implementations.

However, our transition to CRT representation raises several new issues that we need to address, as explained next.

Correctness. First, to ensure correctness, we must reconstruct the integer model $\vec{w}$ from its CRT representation (CRT-reconstruction); this is both straightforward and highly efficient, as the output $\vec{w}$ is in plaintext.

Complexity. Second, complexity involves a trade-off between the number of rings and their sizes, because we require $\prod_i p_i \geq N$ where p_i are the moduli in the CRT representation. As an implementation choice we choose the size to fully utilize our parallel computing resources: with M cores, we choose $p_i \sim N^{1/M}$ conditioned on $p_i < 2^{62}$ for HElib compatibility (otherwise we reduce the size further, and assign several primes to each core). For example, using 40 cores we can reduce $N \sim 2^{2400}$ to $N^{1/40} = 2^{60}$, and assign a single prime p_i to each core for parallel computation.

Security. Third, for security, [6] require that A is invertible in $\mathbb{Z}_N$ (otherwise server $\mathcal{S}_2$ can distinguish invertible from singular matrices A). We make the same requirement in LoPED which, in particular, implies that A is invertible in $\mathbb{Z}_{p_i}$ for all used CRT moduli p_i. Assuming a uniform distribution for the analyzed data, the invertibility requirement holds with high probability. See Sections 3 for further details.

Computing on Packed Encrypted Data. Once we attain compatibility with HElib, the key component for attaining the run-time speedup is using SIMD to pack up to d^2 slots in each ciphertext. This results in speedup by a factor of $\Theta(\mathsf{sl})$, where $\mathsf{sl} \leq d^2$ is the number slots for plaintext packing in each ciphertext.

It is important to note however that naively employing SIMD throughout the computation might not improve efficiency, since operations involving computations on different slots (e.g., matrix multiplication, see below) incur a high overhead that might actually harm efficiency. To address this, we carefully design the various steps in our protocol to be tailored for SIMD computation; this involves two main components as described next; See details in Section 5.

First, as a key component we utilize Jiang et al. [12]'s algorithm for fast multiplication of packed matrices. Specifically, [12] shows how to compute the product of two encrypted $d \times d$ matrices A and B, where: (a) each encrypted matrix is packed in only d ciphertexts (rather than d^2 with entry-by-entry encryption); and (b) the product requires only d homomorphic-multiplications and d homomorphic-additions (rather than $\Theta(d^3)$ homomorphic-operations in the natural matrix-product algorithm). In our protocol only one of the two matrices is encrypted, while the other is known to server $\mathcal{S}_1$ in the clear. Thus, only LHE (and not fully homomorphic encryption) is needed in our use of the algorithm of [12]. SIMD allows to reduce the number of homomorphic operations in the matrix-product step by a factor of up to d^2.

Second, we carefully choose packed representations for all vectors involved in our protocol to make them compatible with the matrix representation of [12], and to support efficient operations of vector-addition and matrix-vector multiplication throughout our protocol.

Parallel computing utilization. The CRT-representation is particularly compatible with parallel computing: we simply run parallel executions of our protocol on the different plaintext moduli p_i. Combining cleartext results is via the well-known CRT-reconstruction algorithm [21], which is highly efficient since it simply computes a linear combination of the results (over the integers) with known integer coefficients.

2 PRELIMINARIES

Notation. We use upper-case letters (e.g., X) to denote matrices, and vector notation (e.g., $\vec{v}$) to denote vectors. For a matrix X, we use X_i to denote its i'th row, and we use X^T to denote the transpose of X. We use boldface letters (e.g., $\mathbf{A}$ for a matrix and $\vec{\mathbf{b}}$ for a vector) to denote ciphertexts. For natural $d, N \in \mathbb{N}$, we use $\mathsf{GL}(d, \mathbb{Z}_N)$ to denote the group of all invertible $d \times d$ matrices with entries in $\mathbb{Z}_N$. We use $\approx$ to denote computational indistinguishability, namely if R, R' are random variables then $\mathsf{R} \approx \mathsf{R}'$ denotes they are computationally indistinguishable. We use $\mathsf{negl}(\sigma)$ to denote a function which is negligible in σ. We use the standard notion of computational indistinguishability (e.g., from [8]). We use PPT as shorthand for Probabilistic Polynomial Time.

Linear-Homomorphic Encryption (LHE) LHE is a public-key encryption scheme that allows one to perform *linear* operations "under the hood" of the encryption, without knowledge of the secret decryption key. We assume that during key generation, one can choose the *plaintext space* by specifying an $N \in \mathbb{N}$, so that homomorphic operations are performed modulo N. This is captured by incorporating the plaintext modulus N explicitly into the syntax of the scheme.

Definition 1. A *Linearly-Homomorphic Encryption (LHE)* scheme $\mathcal{E} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ consists of four algorithms where KG, Enc and Eval are PPT algorithms, and Dec is (deterministic) polynomial time. The algorithms have the following syntax:

- $\mathsf{KG}(1^\sigma, N)$ takes as input a security parameter σ, and an $N \in \mathbb{N}$. It outputs a pair of public and secret encryption keys $(\mathsf{pk}, \mathsf{sk})$. We assume without loss of generality that pk includes N in its description.
- $\mathsf{Enc}(\mathsf{pk}, \mathsf{msg})$ takes as input a public key pk, and a message $\mathsf{msg} \in \mathbb{Z}_N$, and outputs a ciphertext c.
- $\mathsf{Dec}(\mathsf{sk}, \mathsf{c})$ takes as input a secret decryption key sk, and a ciphertext c, and outputs a plaintext message msg'.
- $\mathsf{Eval}(\mathsf{pk}, C, \mathsf{c}_1, \ldots, \mathsf{c}_k)$ takes as input a public key pk, a circuit $C : \mathbb{Z}_N^k \to \mathbb{Z}_N^l$ for some $l, k \in \mathbb{N}$, and k ciphertexts $\mathsf{c}_1, \ldots, \mathsf{c}_k$, and outputs l ciphertexts $\left(\mathsf{c}'_1, \ldots, \mathsf{c}'_l\right)$.

The algorithms are required to satisfy the following semantic properties.

- *Correctness.* For every natural N, every security parameter σ, and every message $\mathsf{msg} \in \mathbb{Z}_p$:

$$\Pr\left[\mathsf{msg} = \mathsf{msg}' : \begin{array}{rl} (\mathsf{pk},\mathsf{sk}) & \leftarrow \mathsf{KG}\left(1^\sigma, N\right) \\ \mathsf{c} & \leftarrow \mathsf{Enc}\left(\mathsf{pk},\mathsf{msg}\right) \\ \mathsf{msg} & = \mathsf{Dec}\left(\mathsf{sk},\mathsf{c}\right) \end{array}\right] = 1 - \mathsf{negl}\left(\sigma\right)$$

  where the probability is over the randomness of KG and Enc.
- *LHE Correctness.* For every natural N, every security parameter σ, every $k, l \in \mathbb{N}$, every circuit $C : \mathbb{Z}_N^k \to \mathbb{Z}_N^l$ consisting only of addition and multiplication by constant gates, and every $\mathsf{msg}_1, \ldots, \mathsf{msg}_k \in \mathbb{Z}_N$, the following probability is at least $1 - \mathsf{negl}\left(\sigma\right)$:

$$\Pr\left[\mathsf{msg} = \mathsf{msg}' : \begin{array}{rl} (\mathsf{pk},\mathsf{sk}) & \leftarrow \mathsf{KG}\left(1^\sigma, N\right) \\ \forall i \in [k],\ \mathsf{c}_i & \leftarrow \mathsf{Enc}\left(\mathsf{pk},\mathsf{msg}_i\right) \\ \mathsf{c} & \leftarrow \mathsf{Eval}\left(\mathsf{pk}, C, (\mathsf{c}_1, \ldots, \mathsf{c}_k)\right) \\ \mathsf{msg} & = \mathsf{Dec}\left(\mathsf{sk},\mathsf{c}\right) \end{array}\right]$$

  where $\mathsf{msg}' = C\left(\mathsf{msg}_1, \ldots, \mathsf{msg}_l\right)$, and the probability is over the randomness of KG, Enc and Eval.
- *Semantic security.* For every σ, N, and every $\mathsf{msg} \in \mathbb{Z}_N$, the joint distribution of pk (i.e., a public key randomly generated by KG) and $\mathsf{c} \leftarrow \mathsf{Enc}\left(\mathsf{pk},\mathsf{msg}\right)$ is computationally indistinguishable from the joint distribution of pk and $\mathsf{c}_0 \leftarrow \mathsf{Enc}\left(\mathsf{pk}, 0\right)$.

Remark on encrypting long messages. Though we define LHE schemes as encrypting a single ring element, we also consider LHE schemes encrypting vectors or matrices of ring elements, namely Enc might take as input a vector or matrix of ring elements, and Dec might take as input a vector or matrix of ciphertexts (each encrypting a field element). See Section 5 for further details.

Remark on Semantic Security Under Evaluations. We note that semantic security (as defined in Definition 1) is preserved under homomorphic evaluation (this follows from a standard hybrid argument). Specifically, semantic security implies that for every σ, N, every $k, l \in \mathbb{N}$, every $\mathsf{msg}_1, \ldots, \mathsf{msg}_k \in \mathbb{Z}_N$, and every $C : \mathbb{Z}_N^k \to \mathbb{Z}_N^l$ which contains only addition and multiplication by constant gates, the following distributions are computationally indistinguishable:

- Sample $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KG}\left(1^\sigma, N\right)$, and $\mathsf{c} \leftarrow \mathsf{Enc}\left(\mathsf{pk}, C\left(\mathsf{msg}_1, \ldots, \mathsf{msg}_k\right)\right)$.[1] Output (pk,c).
- Sample $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KG}\left(1^\sigma, N\right)$, and for every $i \in [k]$, $\mathsf{c}_i \leftarrow \mathsf{Enc}\left(\mathsf{pk}, \mathsf{msg}_i\right)$. Sample $\mathsf{c} \leftarrow \mathsf{Eval}\left(\mathsf{pk}, C, \mathsf{c}_1, \ldots, \mathsf{c}_k\right)$. Output (pk,c).

3 PROBLEM STATEMENT

The Setting. We consider a setting in which m *Data Owners* $DO_1, \ldots, DO_m$ hold private sensitive data, consisting of the labeled examples for a learning algorithm, and wish to execute a ridge regression algorithm (described in Figure 1) on their joint data. Following [14], we focus on the *two-server model* in which the data-owners are aided by two *non-colluding* servers $\mathcal{S}_1$ and $\mathcal{S}_2$ that assist in executing the protocol.

[1] See remark above about encrypting multiple field elements at once.

Naive Solutions. Of course, the hospitals could share their private data, and compute the prediction model based on the aggregated dataset. However, as discussed in Section 1, privacy, business (and even legal) concerns generally forbid this kind of transparent data-sharing arrangement. On the other hand, each hospital could compute a model based only on its own dataset (ignoring the data collected by other hospitals). In this case, all data is trivially kept private, but we expect the resulting models to be less accurate.

Privacy-preserving Ridge Regression is a vivid research area in which protocols for computing prediction models simultaneously (i) make use of all available data (thus optimizing their statistical power), and (ii) preserve data privacy. The participation of the servers reduces the number of communication rounds, and the computational burden on the data owners, which are usually incurred by protocols executed between the data owners alone.

3.1 Security and Threat Model

Our goal is to guarantee correctness of the output, and privacy of the inputs, in the presence of a passive (so-called "honest-but-curious"), computationally-bounded adversary that corrupts a subset of the data owners and at most one server. More specifically, all parties (even corrupted ones) follow the protocol and are restricted to performing PPT computations, but the corrupted parties collude and try to infer as much information as possible from their view of the interaction. Roughly, the security guarantee is that parties (data-owners and servers) learn nothing about the data beyond what is explicitly revealed by the protocol. This explicitly-revealed information is called the "leakage profile", and in our protocols it consists of the shared parameters n, d, ℓ, λ, and the output model $\vec{w}$ (see Figure 1). More formally, we consider k-privacy in the passive setting, for inputs X such that $A = X^T X + \lambda I$ is invertible in Z_N. (As noted in Section 1, this happens with overwhelming probability for naturally-occurring A's.)

Terminology. Let Π be an $m+2$-party protocol executed between PPT data owners $DO_1, \ldots, DO_m$ and PPT servers $\mathcal{S}_1, \mathcal{S}_2$. We assume every pair of parties share a secure point-to-point channel, and all parties share a broadcast channel. We also restrict our attention to protocols in which all parties obtain the same output, and only the data owners have inputs. For inputs $x_1, \ldots, x_m$ of $DO_1, \ldots, DO_m$, we use $\Pi\left(x_1, \ldots, x_m\right)$ to denote the random variable describing the output in a random execution of Π (the probability is over the randomness of *all* participating parties, including the servers). For every party $P \in \{DO_1, \ldots, DO_m, \mathcal{S}_1, \mathcal{S}_2\}$, the *view* of P in Π, denoted $\mathsf{View}_P^\Pi\left(x_1, \ldots, x_m\right)$, is the random variable consisting of the input and randomness of P, as well as the messages P received from the other parties in a random execution of Π with inputs $x_1, \ldots, x_l$. (We note that the messages sent by a passively-corrupted P can be efficiently computed given its view and the description of the protocol.) For a subset $I \subseteq \{DO_1, \ldots, DO_m, \mathcal{S}_1, \mathcal{S}_2\}$ of parties, we use $\mathsf{View}_I^\Pi\left(x_1, \ldots, x_m\right)$ as shorthand for $\left(\mathsf{View}_P^\Pi\left(x_1, \ldots, x_m\right)\right)_{P \in I}$. We say a subset $I \subseteq \{DO_1, \ldots, DO_m, \mathcal{S}_1, \mathcal{S}_2\}$ is *k-permissible* if it contains at most k data owners, and at most one of the servers.

Security notion. We consider standard passive (computational) security (see, e.g., [8]), adapted to the setting of non-colluding servers.

Parties: Data-Owners $DO_1, \ldots, DO_m$ and two-servers $\mathcal{S}_1, \mathcal{S}_2$.

Shared Parameters: Number of data instances n; number of features d; precision ℓ, where all values in $\mathbb{R}$ are normalized to $[-1, 1]$ with a precision of ℓ digits; and regularization parameter $\lambda \in [0, 1]$.

Input: A data-matrix $X \in \mathbb{R}^{n \times d}$ and a response-vector $\vec{y} \in \mathbb{R}^{n \times 1}$, where the input $(X|\vec{y})$ is horizontally-partitioned between the data-owners matrices. That is, each data owner DO_j holds a subset $I_j \subseteq \{1, \ldots, n\}$ of the rows (data instances) of $(X|\vec{y})$, and the I_j's are pairwise disjoint.

Output: A model $\vec{w} \in \mathbb{R}^d$ s.t. $\vec{w} = \text{argmin}_{\vec{w}' \in \mathbb{R}^d} \left\{ \|X \cdot \vec{w}' - \vec{y}\|_2^2 + \lambda \|\vec{w}'\|_2^2 \right\}$.

Leakage Profile: n, d, ℓ, λ, and $\vec{w}$.

Figure 1: Ridge-Regression over federated data (horizontal sharing)

Specifically, following [6] we define correctness with respect to a subset of inputs (where there is no correctness guarantee for inputs not in $\mathcal{T}$), and only require efficient simulateability of k-permissible sets. Formally,

Definition 2 (k-privacy). Let $m, k \in \mathbb{N}$, let σ be a security parameter, let $\mathcal{D}, \mathcal{R}$ be an arbitrary domain and range, let $f : \mathcal{D}^m \to \mathcal{R}$, and let $\mathcal{T} \subseteq \mathcal{D}^m$. We say that an m-party protocol Π realizes f with k-privacy for inputs in $\mathcal{T}$ if:

(1) There exists a negligible function $\text{negl}(\sigma) : \mathbb{N} \to \mathbb{N}$ such that for all inputs $(x_1, \ldots, x_m) \in \mathcal{T}$,

$$\Pr[\Pi(x_1, \ldots, x_m) = f(x_1, \ldots, x_m)] = 1 - \text{negl}(\sigma)$$

where the probability is over the randomness of the parties.

(2) For every k-permissible I there exists a PPT simulator Sim such that for every $(x_1, \ldots, x_m) \in \mathcal{T}$:

$$\text{View}_I^{\Pi}(x_1, \ldots, x_m) \approx \text{Sim}\left(\left(x_j\right)_{DO_j \in I}\right).$$

3.2 Prior State-of-the-Art: PPRR via LHE [6]

The prior state-of-the-art for PPRR in the two-server model is the protocol of Giacomelli et al. [6] that operates on parameters n, d, ℓ, λ as in Figure 1, and with a choice of plaintext ring $\mathbb{Z}_N$ where:

$$N > 2d(d-1)^{\frac{d-1}{2}} 10^{4\ell d} (n^2 + \lambda)^{2d} \tag{1}$$

At a high level, the protocol of [6] operates as follows. During a setup phase, $\mathcal{S}_2$ generates LHE encryption keys (pk, sk), and publishes the public key pk. Then, the protocol trains a ridge-regression model on input $(X|\vec{y}) \in \mathbb{R}^{n \times (d+1)}$ which is horizontally-partitioned between multiple Data Owners as follows. First, each Data Owner computes her share of the $d \times d$ matrix $A = X^T X + \lambda I$, scaled to integer values embedded in $\mathbb{Z}_N$, encrypts this share with the LHE scheme, and sends the ciphertexts to $\mathcal{S}_1$. $\vec{b} = X^T \cdot \vec{y}$ is similarly computed from the contributions of the different Data Owners. Second, $\mathcal{S}_1$ combines all shares to obtain an entry-by-entry encryption $\mathbf{A}$ of A, which is assumed to be invertible in $\mathbb{Z}_N^{d \times d}$. Then, $\mathcal{S}_1$ masks A "under-the-hood" using the homomorphic properties of the LHE scheme. More specifically, using the LHE $\mathcal{S}_1$ computes an encryption $\mathbf{C}$ of $A \cdot R$, where $R \in \text{GL}(d, \mathbb{Z}_N)$ is a random invertible matrix. A masked version of $\vec{b}$ is similarly computed "under-the-hood", where the resulting masked, encrypted, vector $\vec{\mathbf{v}}$ is an encryption of $\vec{v} = \vec{b} + A \cdot \vec{r}$ for a uniformly random $r \in \mathbb{Z}_N^d$. The server $\mathcal{S}_1$ then sends $\mathbf{C}, \vec{\mathbf{v}}$ to $\mathcal{S}_2$. Third, $\mathcal{S}_2$ uses the secret decryption key to decrypt and solves the linear-system $C \cdot \vec{w}^* = \vec{v} \mod N$ to obtain a masked model $\vec{w}^*$ that it sends to $\mathcal{S}_1$. Finally, $\mathcal{S}_1$ removes the masking to obtain the model $\vec{w}' = R \cdot \vec{w}^* - \vec{r} \in \mathbb{Z}_N^d$. The output model $\vec{w} \in \mathbb{Q}^d$ is then obtained from $\vec{w}'$ using *rational reconstruction* [5, 23].[2] After recovering $\vec{w}$, $\mathcal{S}_1$ broadcasts it to all parties.

Remark on correctness and the choice of N. The protocol of [6] performs all intermediate computations (in particular, the computations performed by the ridge regression algorithm) over the ring $\mathbb{Z}_N$ instead of over $\mathbb{R}$. The outcome obtained through this computation is correct in $\mathbb{R}$ when there are no "overflows" in the computation over $\mathbb{Z}_N$. As shown in [6], this holds for N chosen as in Equation 1, for which both the rational reconstruction, and the computation of C^{-1} through $C^{-1} = \text{adj}(C)/\det(C)$ (where $\text{adj}(C)$, $\det(C)$ are the adjoint and determinant of C, respectively) over $\mathbb{Z}_N$ is equivalent to computing over $\mathbb{R}$.

4 OUR PRIVACY-PRESERVING RIDGE-REGRESSION PROTOCOL

In this section we describe our PPRR protocol, which we call LoPED. As discussed in Section 1, it is a variant of the PPRR protocol of [6] with added support for RLWE-based LHE schemes, and improved performance (in terms of runtimes). These properties are obtained by combining the CRT representation (which reduces the plaintext space to one supported by current implementations of RLWE-based LHEs), with SIMD operations (to improve the complexity of homomorphic operations).

Overview. LoPED is a multi-party computation protocol which employs an LHE scheme $\mathcal{E} = (\text{KG}, \text{Enc}, \text{Dec}, \text{Eval})$ as a building block. It is executed between m data owners $DO_1, \ldots, DO_m$, and two servers $\mathcal{S}_1, \mathcal{S}_2$, and operates in three phases. The *Setup* phase establishes cryptographic keys to be used during the execution. During the *Input Uploading* phase, each data owner uploads its (encrypted) data to the servers by sending a single message to $\mathcal{S}_1$. The *Learning* phase is an interactive protocol *between the servers* (with no involvement of the data owners) whose outcome is the (public) model output by the learning algorithm, which is then sent to all data owners. Throughout the protocol, each of the servers has its own designated role: $\mathcal{S}_1$ combines and masks the (encrypted) data contributed by the data owners; whereas $\mathcal{S}_2$ holds the decryption keys and executes the learning algorithm on the un-encrypted, but masked, inputs. We now provide more details on each phase and the roles of all parties.

[2] We use the term "rational reconstruction" to refer to the Lagrange-Gauss algorithm which allows one to recover a rational $q = r/s$ from its representation $q' = r \cdot s^{-1} \in \mathbb{Z}_N$ for sufficiently large N (in particular, this holds for N which satisfies Equation 1).

The Setup Phase (Figure 2) consists of $\mathcal{S}_2$ generating encryption keys for the LHE scheme, and publishing the public keys. Since we use the CRT to represent messages, representing a message in $\mathbb{Z}_N$ using a list of messages in $\mathbb{Z}_{p_1}, \ldots, \mathbb{Z}_{p_t}$ such that $N = \prod_{i=1}^{t} p_i$, independent encryption keys are needed for each modulus p_i. (In contrast, in [6] operations are performed directly in $\mathbb{Z}_N$, and so $\mathcal{S}_2$ generates a single pair (pk, sk) of encryption keys.)

The Input Uploading Phase (Figure 2). Recall from Figure 1 that we assume the input is horizontally partitioned between the data owners. That is, let $X \in \mathbb{R}^{n\times d}, \vec{y} \in \mathbb{R}^{n\times 1}$ denote the data matrix and response vector, respectively, then there exists a partition $I_1, \ldots, I_m$ of $[n]$ such that DO_j holds $X^j = X_{I_j} = \left(X_k \ : \ k \in I_j\right)$ (recall that X_k denotes the k'th row of X) and $\vec{y}^j = \vec{y}_{I_j} = (y_k)_{k\in I_j}$. These inputs are scaled and embedded into $\mathbb{Z}_N$ for a large enough N (for the requirements on N, see Equation 1). For every $1 \le j \le m$, let $A^j = \sum_{k\in I_j} \left(X_k^j\right)^T \cdot X_k^j \in \mathbb{Z}_N^{d\times d}$, and $\vec{b}^j = \sum_{k\in I_j} X_k^j \cdot \vec{y}_k^j \in \mathbb{Z}_N^d$. Each DO_j locally computes $A^j, \vec{b}^j$ from his local inputs, computes

$$A^{j,p_i} = \left(A^j \mod p_i\right) \quad \text{and} \quad \vec{b}^{j,p_i} = \left(\vec{b}^j \mod p_i\right)$$

and sends encryptions of $A^{j,p_i}, \vec{b}^{j,p_i}, i \in [t]$ to $\mathcal{S}_1$.

The Learning Phase (Figure 3). In this phase, $\mathcal{S}_1$ combines and masks (under the hood of the encryption) the data contributed by all data owners, and $\mathcal{S}_2$ performs the learning over the masked data. Following [6, 19], we combine the data to obtain $A = X^T X + \lambda I$ (where λ is the regularization parameter from Figure 1, and I is the identity matrix), and $\vec{b} = X^T \vec{y}$, by summing the contributions of all data owners in the following way. Recall that at the end of the previous phase, $\mathcal{S}_1$ obtained $A^1, \ldots, A^m$ and $\vec{b}^1, \ldots, \vec{b}^m$. We have:

$$A = \sum_{j=1}^{m} A_j + \lambda I \quad \text{and} \quad b = \sum_{j=1}^{m} b_j.$$

Since we use the CRT representation, instead of computing $A, \vec{b}$ we need to compute $A^{p_i} = A \mod p_i$ and $\vec{b}^{p_i} = \vec{b} \mod p_i$ for every $i \in [t]$. Towards that end, $\mathcal{S}_1$ uses the linear-homomorphism of the LHE scheme to compute $A^{p_i}, \vec{b}^{p_i}$ from the $A^{j,p_i}, \vec{b}^{j,p_i}$'s (see Step 1 in Figure 3).

Our masking method is similar to [6] (described in Section 3.2), with the modification that we need to guarantee compatibility with the CRT representation. Specifically, similar to [6], $\mathcal{S}_1$ chooses a random invertible $R \in \mathbb{Z}_N^{d\times d}$, and a random $\vec{r} \in \mathbb{Z}_N^d$, and computes $C = A \cdot R$ and $\vec{v} = \vec{b} + A \cdot \vec{r}$. However, since for every $i \in [t]$, operations on $A^{p_i}, \vec{b}^{p_i}$ are performed in $\mathbb{Z}_{p_i}$, masking is performed with $R^{p_i} = R \mod p_i$ and $\vec{r}^{p_i} = \vec{r} \mod p_i$, namely the masked data matrix is $C^{p_i} = A^{p_i} \cdot R^{p_i} \mod p_i$, and the masked response vector is $\vec{v}^{p_i} = \vec{b}^{p_i} + A^{p_i} \cdot \vec{r}^{p_i} \mod p_i$ (see Step 3a in Figure 3). $C^{p_i}, \vec{v}^{p_i}$ are computed by $\mathcal{S}_1$ using the linear-homomorphism of the LHE scheme, and sent to $\mathcal{S}_2$. $\mathcal{S}_2$ decrypts these ciphertexts to obtain $C^{p_i}, \vec{v}^{p_i}, i \in [t]$, and uses CRT reconstruction [21] to recover $C, \vec{v}$. Using these, $\mathcal{S}_2$ solves the linear system $C \cdot \vec{w} = \vec{v}$ by computing $C^{-1} = \mathrm{adj}(C)/\det(C)$ (where $\mathrm{adj}(C), \det(C)$ denote the adjoint and determinant of C, respectively) to obtain a masked model $\vec{w}^*$, which it sends to $\mathcal{S}_1$. $\mathcal{S}_1$ can now unmask the output model as $\vec{w} = R \cdot \vec{w}^* - \vec{r}$. This gives a model in $\mathbb{Z}_N^d$, from which the corresponding model in $\mathbb{Q}^d$ is reconstructed using rational reconstruction [5, 23] (see discussion in Section 3.2).

SIMD operations. Our protocols can use any LHE scheme. In particular, we describe in Section 5 how to incorporate SIMD operations (when supported by the underlying LHE, as is the case for the LHE scheme used in our experiments) into our protocols. We note that for compatibility with these SIMD operations, the data should be encoded (in a "SIMD-friendly" encoding) before it is encrypted (see, e.g., Step 3a in Figure 3). As described in Section 5, we consider four different encodings (type-L, type-R, type-M and type-A).

4.1 The Security Guarantee of LoPED

We now formalize the security guarantee of our PPRR protocol LoPED . For a natural N, and $\lambda \ge 0$, let $\mathcal{T}_{\mathsf{lnv},N,\lambda}$ denote the subset of $\mathbb{R}^{n\times d} \times \mathbb{R}^{n\times 1}$ consisting of all $(X, \vec{y})$ for which $A = X^T X + \lambda I$ is invertible in $\mathbb{Z}_N^{d\times d}$. Similar to [6], we only consider security for inputs in $\mathcal{T}_{\mathsf{lnv},N,\lambda}$, where λ and N are as defined in Figure 2.

Theorem 3 (LoPED Security). *LoPED is m-private for inputs in $\mathcal{T}_{\mathsf{lnv},N,\lambda}$ for any $\lambda \ge 0$ and any $N \in \mathbb{N}$ that satisfies Equation 1.*

The proof will use the following Lemma from [6, Lemma 1].

Lemma 4 (Lemma 1 from [6]). *Let $N, d \in \mathbb{N}$, and let $A \in GL(d, \mathbb{Z}_N)$ and $\vec{b} \in \mathbb{Z}_N^d$. Then the random variable defined by picking a random $R \leftarrow GL(d, \mathbb{Z}_N)$ and a random $\vec{r} \leftarrow \mathbb{Z}_N^d$ and outputting $\left(A \cdot R, \vec{b} + A \cdot \vec{r}\right)$ is uniformly distributed over $GL(d, \mathbb{Z}_N) \times \mathbb{Z}_N^d$.*

Proof of Theorem 3. Correctness when all parties are honest follows immediately from the description of the protocol, the correctness of the rational reconstruction algorithm for N that satisfies Equation 1, and the correctness of the CRT reconstruction.

As for privacy, let $I \subseteq [m]$ denote the subset of corrupted data owners, and we consider three cases. First, assume that $\mathcal{S}_1$ is corrupted. The simulator Sim, given the inputs $\left\{(X^j, \vec{y}^j)\right\}_{j\in I}$ of the corrupted data owners, and the model $\vec{w} \in \mathbb{Q}^d$, operates as follows.

- During setup, Sim honestly generates encryption keys $\left(\mathsf{pk}_i, \mathsf{sk}_i\right), i \in [t]$.
- During the input-uploading phase, for every honest DO_j and every $i \in [t]$, it generates a random encryption $\mathbf{A}^{j,\mathsf{pk}_i} \leftarrow \mathsf{Enc}\left(\mathsf{pk}_i, I_d\right)$ where I_d denotes the type-L encoding of the $d \times d$ identity matrix, and $\vec{\mathbf{b}}^{j,p_i} \leftarrow \mathsf{Enc}\left(\mathsf{pk}_i, \vec{0}^d\right)$ where $\vec{0}^d$ denotes the type-A encoding of the length-d all-zeros vector. Let $\mathsf{c} = \left\{\left(\mathbf{A}^{j,p_i}, \vec{\mathbf{b}}^{j,p_i}\right)\right\}_{j\notin I, i\in[t]}$.
- During the learning phase, Sim picks $R \leftarrow \mathrm{GL}(d, \mathbb{Z}_N)$ and $\vec{r} \leftarrow \mathbb{Z}_N^d$, uses $\vec{w}$ to determine the corresponding model $\vec{w}' \in \mathbb{Z}_N^d$, and sets $\vec{w}^* = R^{-1} \cdot (\vec{w}' + \vec{r})$.
- Outputs $\left(\left\{(X^j, \vec{y}^j)\right\}_{j\in I}, \vec{w}, \left\{\mathsf{pk}_i\right\}_{i\in[t]}, \mathsf{c}, R, \vec{r}, \vec{w}^*\right)$.

Notice that $R, \vec{r}$ are identically distributed in the real execution and the simulation, and so is $\vec{w}^*$ since it is uniquely determined by $\vec{w}', R$ and $\vec{r}$. Therefore, the only difference between the simulated and real views are the encryptions $\mathbf{A}^{j,p_i}, \vec{\mathbf{b}}^{j,p_i}$ of the honest DO_j's. However, by the semantic security of the LHE scheme, the simulated and real ciphertexts are computationally indistinguishable.

The Setup Phase

Shared parameters: an LHE scheme $\mathcal{E}$ = (KG, Enc, Dec, Eval), a security parameter σ, the number of data owners m, a CRT parameter t, and t distinct primes $p_1, \ldots, p_t$ such that $\prod_i p_i = N$ for N satisfying Equation 1.
Input: the parties have no private inputs.
Output: encryption keys $\left\{\left(\mathsf{pk}_i, \mathsf{sk}_i\right)\right\}_{i\in[t]}$ for $\mathcal{S}_2$, public keys $\left\{\mathsf{pk}_i\right\}_{i\in[t]}$ for all other parties.
Steps:

(1) For every $1 \le i \le t$, $\mathcal{S}_2$ generates encryption keys $\left(\mathsf{pk}_i, \mathsf{sk}_i\right) \leftarrow \mathsf{KG}\left(1^\sigma, p_i\right)$.
(2) $\mathcal{S}_2$ publishes $\mathsf{pk}_1, \ldots, \mathsf{pk}_t$.

The Input-Uploading Phase

Shared parameters: $\mathcal{E}$ = (KG, Enc, Dec, Eval), σ, m, t, and $p_1, \ldots, p_t$ as above. In addition, dimensions $n \times d$ of the input-matrix, and positive input sizes $n_1, n_2, \ldots, n_m > 0$ such that $\sum_{j=1}^{m} n_j = n$.
Input from previous phase: all parties take as input public encryption keys $\left\{\mathsf{pk}_i\right\}_{i\in[t]}$. The input of $\mathcal{S}_2$ additionally includes the corresponding secret decryption keys $\{\mathsf{sk}_i\}_{i\in[t]}$.
Input: for every $j \in [m]$, the input of data owner DO_j includes a matrix $X^j \in \mathbb{R}^{n_j \times d}$, and a vector $\vec{y}^j \in \mathbb{R}^{n_j}$.
Output for the next phase: the output of $\mathcal{S}_1$ is, for every $i \in [t]$ and every $j \in [m]$, the encryptions $\mathbf{A}^{j,p_i}, \vec{\mathbf{b}}^{j,p_i}$ of a matrix $A^{j,p_i} \in \mathbb{Z}_{p_i}^{d\times d}$ and a vector $\vec{b}^{j,p_i} \in \mathbb{Z}_{p_i}^{d}$, respectively. The other parties have no output. (We note that $A^{j,p_i} = \left(X^j\right)^T \cdot X^j \mod p_i$, and $\vec{b}^{j,p_i} = \left(X^j\right)^T \cdot \vec{y} \mod p_i$.)
Steps:

(1) **Data Representation:** for every $1 \le j \le m$, DO_j scales its inputs X^j, $\vec{y}^j$ to have entries in $\mathbb{Z}_N$. Then, DO_j computes $A^j = \sum_{k=1}^{n_j} \left(X_k^j\right)^T \cdot X_k^j$ (recall that X_k^j denotes the k'th row of X^j), and $\vec{b}^j = \sum_{k=1}^{n_j} X_k^j \cdot \vec{y}_k^j$.
(2) **Data Encryption:** for every $1 \le j \le m$ and every $1 \le i \le t$, DO_j encodes A^{j,p_i} as a type-L matrix $A^{j,p_i,\star}$ (see SIMD encoding in Section 5.2.1), and encodes $\vec{b}^{j,p_i}$ as a type-A vector $\vec{b}^{j,p_i,\star}$ (see SIMD encoding in Section 5.2.2). Then, DO_j generates encryptions $\mathbf{A}^{j,p_i} \leftarrow \mathsf{Enc}\left(\mathsf{pk}_i, A^{j,p_i,\star}\right)$ and $\vec{\mathbf{b}}^{j,p_i} \leftarrow \mathsf{Enc}\left(\mathsf{pk}_i, \vec{b}^{j,p_i,\star}\right)$, and sends $\left\{\left(\mathbf{A}^{j,p_i}, \vec{\mathbf{b}}^{j,p_i}\right)\right\}_{j\in[m],i\in[t]}$ to $\mathcal{S}_1$.

Figure 2: LoPED Setup and Input Uploading Phases.

The Learning Phase

Shared parameters: $\mathcal{E}$ = (KG, Enc, Dec, Eval), σ, m, t, n, d, $p_1, \ldots, p_t$ as in Figure 2. In addition, a regularization parameter λ.
Inputs from previous phase: all parties take as input public encryption keys $\left\{\mathsf{pk}_i\right\}_{i\in[t]}$. The input of $\mathcal{S}_2$ additionally includes the corresponding secret decryption keys $\{\mathsf{sk}_i\}_{i\in[t]}$. The input of $\mathcal{S}_1$ includes, for every $i \in [t]$ and every $j \in [m]$, ciphertexts $\mathbf{A}^{j,p_i}, \vec{\mathbf{b}}^{j,p_i}$ of $A^{j,p_i} \in \mathbb{Z}_{p_i}^{d\times d}$, $\vec{b}^{j,p_i} \in \mathbb{Z}^d$.
Output: all parties obtain a model $\vec{w} \in \mathbb{Q}^d$.
Steps:

(1) **Data Merging:** for every $1 \le i \le t$, $\mathcal{S}_1$ computes $\mathbf{A}^{p_i} \leftarrow \mathsf{Eval}\left(\mathsf{pk}_i, \mathsf{Add}_\lambda, \mathbf{A}^{1,p_i}, \ldots, \mathbf{A}^{m,p_i}\right)$, and $\vec{\mathbf{b}}^{p_i} \leftarrow \mathsf{Eval}\left(\mathsf{pk}_i, \mathsf{Add}, \vec{\mathbf{b}}^{1,p_i}, \ldots, \vec{\mathbf{b}}^{m,p_i}\right)$ (see Notation 8 in Section 5). (We note that if X denotes the matrix obtained by concatenating the rows of all X^j, and $\vec{y}$ is the vector obtained by concatenating all the $\vec{y}^j$'s, then $A^{p_i} = X^T \cdot X + \lambda I \mod p_i$ where I denotes the identity matrix, and $\vec{b}^{p_i} = X^T \cdot \vec{y} \mod p_i$.)
(2) **Randomness Generation:** $\mathcal{S}_1$ picks a random invertible $d \times d$ matrix $R \leftarrow \mathsf{GL}(d, \mathbb{Z}_N)$, and a random vector $\vec{r} \leftarrow \mathbb{Z}_N^d$.
(3) for every $1 \le i \le t$: (the computation steps for each of the p_i's can be done in parallel)
   (a) **Data Masking:** $\mathcal{S}_1$ computes $R^{p_i} = R \mod p_i$ and $\vec{r}^{p_i} = \vec{r} \mod p_i$. Then, $\mathcal{S}_1$ encodes R^{p_i} as a type-R matrix (see SIMD encoding in Section 5.2.1), and encodes $\vec{r}^{p_i}$ as a type-M vector (see SIMD encoding in Section 5.2.2). Next, $\mathcal{S}_1$ computes $\mathbf{C}^{p_i} \leftarrow \mathsf{Eval}\left(\mathsf{pk}_i, \mathsf{MatMult}, \mathbf{A}^{p_i}, R^{p_i}\right)$ (see Notation 6 in Section 5.2.1), $\vec{\mathbf{z}}^{p_i} \leftarrow \mathsf{Eval}\left(\mathsf{pk}_i, \mathsf{MatMult}, \mathbf{A}^{p_i}, \vec{r}^{p_i}\right)$, and $\vec{\mathbf{v}}^{p_i} \leftarrow \mathsf{Eval}\left(\mathsf{pk}_i, \mathsf{VecAdd}, \vec{\mathbf{b}}^{p_i}, \vec{\mathbf{z}}^{p_i}\right)$ (see Notation 7 in Section 5.2.2). Finally, $\mathcal{S}_1$ sends $\mathbf{C}^{p_i}, \vec{\mathbf{v}}^{p_i}$, $i \in [t]$ to $\mathcal{S}_2$. (We note that $\mathbf{C}^{p_i}$ encrypts $A^{p_i} \cdot R^{p_i}$, and $\vec{\mathbf{v}}^{p_i}$ encrypts $\vec{b}^{p_i} + A^{p_i} \cdot \vec{r}^{p_i}$.)
   (b) **Decrypting Masked Data:** $\mathcal{S}_2$ decrypts $C^{p_i} = \mathsf{Dec}\left(\mathsf{pk}_i, \mathbf{C}^{p_i}\right)$ and $\vec{v}^{p_i} = \mathsf{Dec}\left(\mathsf{pk}_i, \vec{\mathbf{v}}^{p_i}\right)$.
(4) **Masked learning:** $\mathcal{S}_2$ uses the CRT reconstruction [21] to reconstruct $C, \vec{v}$ from $\{(C^{p_i}, \vec{v}^{p_i})\}_{i\in[t]}$. Then, $\mathcal{S}_2$ computes the masked model $\vec{w}^* = C^{-1} \cdot \vec{v}$ (C^{-1} is computed as $C^{-1} = \mathrm{adj}(C)/\det(C)$, where $\mathrm{adj}(C), \det(C)$ denote the adjoint and determinant of C, respectively), and sends $\vec{w}^*$ to $\mathcal{S}_1$.
(5) **Unmasking:** $\mathcal{S}_1$ computes $\vec{w}' = R \cdot \vec{w}^* - \vec{r}$, recovers from it the model $\vec{w} \in \mathbb{Q}^d$ using rational reconstruction [5, 23] in each coordinate, and sends $\vec{w}$ to all parties.

Figure 3: LoPED Learning Phase.

Second, assume that $\mathcal{S}_2$ is corrupt. The simulator Sim, given the inputs $\left\{\left(X^j, \vec{y}^j\right)\right\}_{j\in I}$ of the corrupted data owners, and the model $\vec{w} \in \mathbb{Q}^d$, operates as follows.

- During setup, Sim honestly generates encryption keys $\left(\mathsf{pk}_i, \mathsf{sk}_i\right), i \in [t]$.
- During the input-uploading, the corrupted parties do not receive any messages, so there is nothing to simulate.
- During the learning phase, Sim picks a random $C \leftarrow \mathsf{GL}(d, \mathbb{Z}_N)$, and a random $\vec{v} \leftarrow \mathbb{Z}_N^d$. Then, for every $i \in [t]$, it computes $C^{p_i} = C \mod p_i$ and $\vec{v}^{p_i} = \vec{v} \mod p_i$. Next, it encrypts $\mathbf{C}^{p_i} \leftarrow \mathsf{Enc}\left(\mathsf{pk}_i, C^{p_i}\right)$ and $\vec{\mathbf{v}}^{p_i} \leftarrow \mathsf{Enc}\left(\mathsf{pk}_i, \vec{v}^{p_i}\right)$

- Outputs $\left(\left\{\left(X^j, \vec{y}^j\right)\right\}_{j\in I}, \vec{w}, \left\{\left(\mathsf{pk}_i, \mathsf{sk}_i\right), \left(\mathbf{C}^{p_i}, \vec{\mathbf{v}}^{p_i}\right)\right\}_{i\in[t]}\right)$.

The only difference between the simulated and real views is the choice of the C^{p_i}'s and the $\vec{v}^{p_i}$'s, which in the real view are generated from the A^{p_i}'s and $\vec{b}^{p_i}$'s, whereas in the simulation they are chosen at random. However, these are identically distributed by Lemma 4 since $A \in \mathsf{GL}(d, \mathbb{Z}_N)$ when $(X, \vec{y}) \in \mathcal{T}_{\mathsf{Inv},N,\lambda}$.

Third, assume that both servers are honest. Since the servers have no input, and the output is public, simulatability follows immediately from simulatability when one of the servers is corrupted. □

4.2 Complexity of LoPED

We state the complexity of LoPED, focusing on the homomorphic operations, which is where LoPED deviates from the prior-art [6]; see Section 5 for the proof.

Theorem 5 (Complexity). *Let $d, m, \ell, n \in \mathbb{N}$ be as in Figure 1. Then:*
(1) Step 1 (data merging) in Figure 3 requires $O(t \cdot m \cdot d\lceil\frac{d^2}{sl}\rceil)$ homomorphic operations.
(2) Step 3a (data masking) in Figure 3 requires $O(t \cdot d\lceil\frac{d^2}{sl}\rceil)$ homomorphic operations.
where sl is the number of values packed in a ciphertext and $t = O(d\log(d \cdot n \cdot \lambda \cdot 2^\ell))$.

5 INCORPORATING SIMD INTO LOPED

In this section, we describe how we incorporate SIMD computations into our protocol to improve efficiency. We start with some background on SIMD.

5.1 Background on SIMD

Encryption scheme and implementation supporting Single Instruction Multiple Data (SIMD) include: the RLWE-based schemes of Brakerski-Gentry-Vaikuntanathan [2] and Fan-Vercauteren [4] (often referred to as BGV/FV) and their implementations in HElib [11] and SEAL [3] libraries. These schemes allow "packing" multiple messages in different "slots" in a *single* ciphertext. We denote the packing parameter, namely the number of messages packed in one ciphertext as sl, and the different slots in the ciphertext by $\mathbf{c} = (\mathbf{c}(1), \ldots, \mathbf{c}(\mathsf{sl}))$. Operations on packed ciphertexts are done in a SIMD (Single Instruction Multiple Data) manner. For example, a SIMD multiplication, denoted by $\mathbf{c} = \mathbf{a} \odot \mathbf{b}$, is defined by $\mathbf{c}(i) = \mathbf{a}(i) \cdot \mathbf{b}(i)$, for $i = 1, \ldots, \mathsf{sl}$.

It is important to note that naively employing SIMD throughout the computation might not improve efficiency, since operations involving computations on different slots (e.g., matrix multiplication, see below) incur a high overhead that might actually harm efficiency.

5.2 Incorporating SIMD into LoPED

Our design of LoPED is tailored to efficient use of SIMD, in the sense that we design all computations to apply only on values in the same SIMD slots ("element-wise" computation). Namely, when applying an instruction, say $\odot$, on packed ciphertexts $\mathbf{c} = (\mathbf{c}(1), \ldots, \mathbf{c}(\mathsf{sl}))$ and $\mathbf{c'} = (\mathbf{c'}(1), \ldots, \mathbf{c'}(\mathsf{sl}))$, the computation produces the ciphertext $\mathbf{c''} = (\mathbf{c}(1) \odot \mathbf{c'}(1), \ldots, \mathbf{c}(\mathsf{sl}) \odot \mathbf{c'}(\mathsf{sl}))$. In this section we specify the details of the SIMD packing we use to attain the goal that all computations are element-wise.

First recall the operations homomorphically evaluated in LoPED:

(1) Adding vectors (Step 1, Figure 3).
(2) Adding matrices (Step 1, Figure 3).
(3) Matrix by matrix multiplication (Step 3a, Figure 3).
(4) Matrix by vector multiplication (Step 3a, Figure 3).

Out of these operations, the most expensive one is operation number 3 (matrix-by-matrix multiplication), and we therefore focus on optimizing it. Towards that end, we use the SIMD matrix multiplication scheme described in [12]. We combine this with new representations for vectors which we introduce, and new protocols which we design for efficiently performing the other operations described above.

We now describe how matrices and vectors are represented, and how the aforementioned operations are performed.

5.2.1 Matrix Encoding and Matrix-by-Matrix Multiplication. As noted above, we use the scheme of [12] for efficient matrix multiplication using SIMD, and therefore use their encoding of matrices. Jiang et al. [12] differentiate between matrices that are multiplied from the left and matrices that are multiplied from the right, and use different encodings for each. Let L and R be two $d \times d$ matrices, where we wish to compute $A = L \cdot R$. We now describe the encodings of L and R.

Type-L and type-R encoding, and Type-N notation. We denote the encoding of the matrix L, which appears as the left matrix in the product, as a *type-L encoding*. It consists of d rotations $L_1, \ldots, L_d$ of L, where each L_i is a $d \times d$ matrix that is computed from L by rotating its rows. Similarly, we denote the encoding of the matrix R, which appears as the right matrix in the product, as a *type-R encoding*. It consists of d rotations $R_1, \ldots, R_d$, where each R_i is a $d \times d$ matrix obtained from R by rotating its columns (see Figure 4 for an example). We call "Type-N" a matrix that is encoded in its *N*ative form, i.e., with no rotations just taking the original matrix, when we want to emphasize it is not in Type-L or Type-R.

We note that though the matrix encoding effectively results in holding d copies of the matrix, these copies can be packed into a smaller number of ciphertexts, resulting in improved overhead (running time and RAM requirements). Specifically, as we show below that the naive matrix representation requires d^2 ciphertexts, whereas our representation requires $d \cdot \lceil\frac{d^2}{\mathsf{sl}}\rceil$ ciphertexts (which improves over d^2 when $\mathsf{sl} > d$).

Matrix-by-matrix multiplication. The type-L and type-R encoding allow for efficient multiplication where operations require only element-wise computation. Indeed, the product $A = L \cdot R$ can be decomposed as $A = \sum_{i=1}^{d} A_i$, where $A_i = L_i \odot R_i$ is the $d \times d$ matrix obtained by computing the element-wise product of the rotations L_i and R_i, namely $(A_i)_{c,l} = (L_i)_{c,l} \cdot (R_i)_{c,l}$, for $1 \le c, l \le d$. An example for $d = 3$ is given in Figure 4. We refer the interested reader to [12] for further details.

In our protocol, we multiply an encrypted matrix with a public matrix (see Step 3a in Figure 3). This is done using the Eval algorithm of the underlying LHE scheme, where Eval is used to evaluate the following circuit.

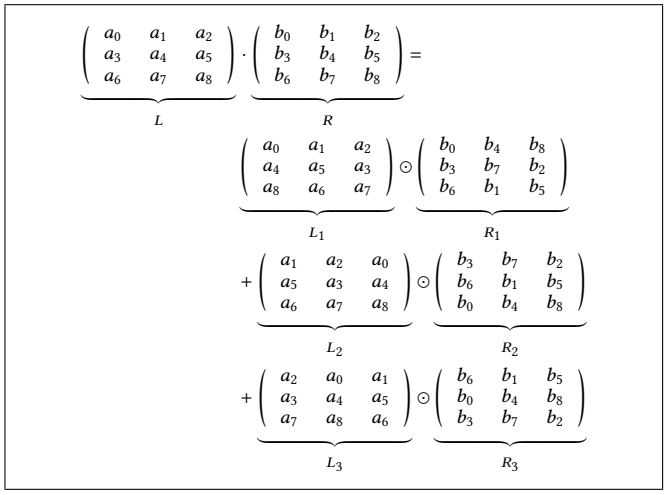

Figure 4: Matrix-by-matrix multiplication example

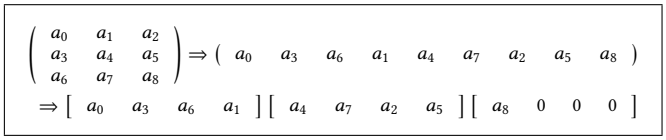

Figure 5: Serialization and packed encryption example. The 3×3 matrix is serialized in the first line; and packed into ciphertexts, each with 4 slots, in the second line.

Figure 6: type-M (left) and type-A (right) encoding example for a 3-dimensional vector $v = (v_0, v_1, v_2)$ with $\mathsf{sl} = 4$. The type-M encoding is then mapped to a type-R matrix encoding as described in Section 5.2.1.

Notation 6 (Matrix-by-Matrix multiplication circuit). *Let $N, d \in \mathbb{N}$, let $R \in \mathbb{Z}_N^{d\times d}$ be a fixed matrix, and let $(R_1, \ldots, R_d)$ be its type-R encoding. The circuit $C_{MatMult,R} : \left(\mathbb{Z}_N^{d\times d}\right)^d \to \mathbb{Z}_N^{d\times d}$ on input a type-L encoding $(L_1, \ldots, L_d)$ of a matrix L, outputs the product $A = L \cdot R$ computed as follows. For every $1 \le i \le d$, and for every $1 \le c, l \le d$, compute $(A_i)_{c,l} = (L_i)_{c,l} \cdot (R_i)_{c,l}$, and output $\sum_{i=1}^{d} A_i$. The output matrix A is in native form (Type-N).*

For an LHE scheme $\mathcal{E} = (KG, Enc, Dec, Eval)$, a public key pk, an encryption $\mathbf{L}$ of a type-L encoding of a matrix L, and a type-R encoding of a public matrix R, we use $Eval(pk, MatMult, \mathbf{L}, R)$ to denote running Eval on input the circuit $C_{MatMult,R}$ and the ciphertext $\mathbf{L}$.

Improved Efficiency through Serialization. Following [12], we improve efficiency (and overcome the overhead introduced by holding multiple rotated copies of each matrix) by packing multiple matrix entries into a single ciphertext. This is done by first representing ("serializing") the $d \times d$ matrix as a vector of length d^2.

We describe a specific serialization method which optimizes the complexity of the four aforementioned operations we care about (whereas [12] do not specify the exact serialization method, since it was of no importance to them). Specifically, we serialize a $d \times d$ matrix A by copying its columns from left to right, where the entries in each column are copied from top to bottom. That is, the entry $A_{c,l}$ of A appears as the $((c-1)d + l)$'th entry of the serialized A. The serialized matrix (padded with zeros if needed) can then be encrypted into a ciphertext (packed into slots). See Figure 5 for an example. Since we use serialization to represent all matrices in our protocol, we use "type-L" ("type-R", respectively) encoding to denote the *serialized* representation of the type-L (type-R, respectively) encoded matrix.

We now explain why serialization improves efficiency, when combined with packing. Let sl denote the number of slots in a single ciphertext, then encrypting a $d \times d$ matrix naively requires d^2 ciphertexts (one for each entry), whereas using serialization and packing as described above, we only need $d\lceil \frac{d^2}{\mathsf{sl}} \rceil$ ciphertexts. Thus, computing the product of two $d \times d$ matrices requires $d\lceil \frac{d^2}{\mathsf{sl}} \rceil$ ciphertext multiplications, as opposed to d^3 in the naive implementation. Consequently, the number of homomorphic operations computed in LoPED is $\Theta(d \cdot \lceil \frac{d^2}{\mathsf{sl}} \rceil)$ (cf. $\Theta(d^3)$ in [6]). Furthermore, the number of communicated ciphertexts, for each $i \in [t]$, is $\lceil \frac{d^2}{\mathsf{sl}} \rceil$ (cf. d^2 in [6]).

5.2.2 Vector Representation. In this section we describe our method of representing vectors, which complements the matrix representation of [12], and supports efficient operations on vectors (which are needed by our protocol). We describe two different encodings for vectors, where each is used to optimize the complexity of a different operation. Our first encoding, which we call *type-M*, allows for efficient matrix-vector multiplication. The second encoding, which we call *type-A*, allows for efficient vector additions.

Looking ahead, our protocol (See Step 3a in Figure 3) computes $\vec{b} + A \cdot \vec{r}$, where A is a $d \times d$ matrix and $\vec{r}, \vec{b}$ are d-dimensional vectors. To compute the matrix-vector multiplication $A \cdot \vec{r}$, we define the type-M encoding of a vector to be consistent with our matrix representation. Specifically, We first map $\vec{r}$ into a $d \times d$ matrix, whose first column is $\vec{r}$ and the rest are zero-columns. This matrix is then encoded as a type-R encoding for matrices, as described in Section 5.2.1. The type-A encoding of a vector $\vec{b}$ pads $\vec{b}$ with zeros to have length divisible by sl (such that, when encrypted, $\vec{b}$ can be packed). See Figure 6 for an example of type-M and type-A encodings.

We note that though addition *can* be performed on type-M encodings, it is less efficient than adding two type-A encodings. For example, when $\mathsf{sl} = d$, adding two type-M vectors requires d operations, but adding two type-A encodings requires only a single operation. We additionally note that operations combining both type-M and type-A encodings are expensive.

5.2.3 Incorporating SIMD into our protocol. We now describe how we incorporate SIMD into our protocol. Recall that the ciphertext operations performed by our protocol are data encryption (Step 2, Figure 2), data merging (Step 1, Figure 3), and masking (Step 3a, Figure 3).

Data Encryption (Figure 2, Step 2 of the input uploading phase). In the input uploading phase of our protocol each data owner DO_j holds an input matrix X^j of size $n_j \times d$, and a vector $\vec{y}^j$ of length n_j. DO_j then computes a $d \times d$ matrix $A^j = \sum_{k=1}^{n_j} \left(X_k^j\right)^T \cdot X_k^j$ (recall that X_k^j denotes the k'th row of X^j), and $\vec{b}^j = \sum_{k=1}^{n_j} X_k^j \cdot \vec{y}_k^j$. A^j is then encoded as a type-L matrix, and $\vec{b}^j$ is encoded as a type-A vector.

Data Merging (Figure 3, Step 1). To merge the contributions $A^1, \ldots, A^m$ from the m data owners, $\mathcal{S}_1$ computes

$$A = \sum_{i=1}^{m} A_i \quad \text{and} \quad b = \sum_{i=1}^{m} b_i.$$

Since the additions are done on ciphertexts with sl slots each, then computing A requires $(m-1)\lceil \frac{d^2}{\mathsf{sl}} \rceil$ operations, and computing $\vec{b}$ requires $(m-1)\lceil \frac{d}{\mathsf{sl}} \rceil$ operations.

Masking A (Figure 3, Step 3a). To mask A, $\mathcal{S}_1$ draws a random invertible matrix $R \leftarrow \mathrm{GL}(d, \mathbb{Z}_N)$, and computes $C = A \cdot R$, where both A and R are $d \times d$ matrices. Recall that $A = \sum_{j=1}^{m} A_j$ where each A_j is a type-L encoding, so A is also type-L. Since R is known to $\mathcal{S}_1$ in plaintext form, $\mathcal{S}_1$ can encode it as a type-R matrix, and then compute $C = A \cdot R$ using the matrix multiplication algorithm described in Section 5.2.1. Using the analysis of Section 5.2.1, this step requires $\lceil \frac{d^2}{\mathsf{sl}} \rceil$ operations.

Masking b (Figure 3, Step 3a). To mask $\vec{b}$, $\mathcal{S}_1$ draws a random $\vec{r} \leftarrow \mathbb{Z}_N^d$, and computes $\vec{v} = \vec{b} + A \cdot \vec{r}$. Recall that $\vec{b} = \sum_{j=1}^{m} \vec{b}^j$ where each $\vec{b}^j$ is a type-A vector, so $\vec{b}$ is also type-A. To be consistent with the encoding of $A, \vec{b}$, and the definition of $\vec{v}$, $\mathcal{S}_1$ encodes $\vec{r}$ as a type-M vector.

An important observation is that when A is type-L encoded and $\vec{r}$ is type-M encoded, then $A \cdot \vec{r}$ can be added to the type-A encoded $\vec{b}$, as we explain next. Let $\vec{r}^*$ denote the type-M encoding of r, i.e., the serialized matrix whose first column is $\vec{r}$ and the rest are zero-columns. Then the first column of $A \cdot \vec{r}^*$ is $A \cdot \vec{r}$, and the rest are zero-columns. Since $A \cdot \vec{r}^*$ is also serialized, it is a vector whose first entries correspond to $A \cdot \vec{r}$ (the first column of $A \cdot \vec{r}^*$), followed by zeros. Thus, if the encryption of $A \cdot \vec{r}^*$ contains more ciphertexts than the encryption of $\vec{b}$, the extra ciphertexts in $A \cdot \vec{r}^*$ can be discarded. Then, $\vec{b}$ can be added to (the possibly truncated) $A \cdot \vec{r}^*$ to get $\vec{v} = \vec{b} + A \cdot \vec{r}$. See Figure 7 for an example.

Recall that in our protocol (Figure 3) $A, \vec{b}$ are encrypted, i.e., $\mathcal{S}_2$ computes $\vec{\mathbf{b}} + \mathbf{A} \cdot \vec{r}$ (see Step 3a in Figure 3). Since $\vec{r}$ is represented as a type-M vector, the product $\mathbf{A} \cdot \vec{r}$ can be computed by applying the Eval algorithm of the underlying LHE scheme to the circuit of Notation 6. We now describe the circuit used to compute the addition of two vectors.

Notation 7 (Vector addition circuit). *Let $sl, N, n, n' \in \mathbb{N}$ such that $n \leq n'$. The circuit $C_{VecAdd} : (\mathbb{Z}_N^{sl})^n \times (\mathbb{Z}_N^{sl})^{n'} \to (\mathbb{Z}_N^{sl})^n$ on input two vectors $\vec{b}, \vec{b}'$ packed as vectors of length-sl sub-vectors operates as follows. First, it truncates $\vec{b}'$ to have length n by discarding the $n' - n$ last vectors. Denote the truncated vector by $\vec{b}''$ (notice that if $n = n'$ then $\vec{b}' = \vec{b}''$). Then, it outputs the sum $\vec{b} + \vec{b}''$.*

For an LHE scheme $\mathcal{E} = (KG, Enc, Dec, Eval)$, a public key pk, and encryptions $\vec{\boldsymbol{b}}, \vec{\boldsymbol{b}}'$ of vectors in $(\mathbb{Z}_N^{sl})^n, (\mathbb{Z}_N^{sl})^{n'}$ (respectively), we use $Eval\left(pk, VecAdd, \vec{\boldsymbol{b}}, \vec{\boldsymbol{b}}'\right)$ to denote running the Eval algorithm on input the circuit C_{VecAdd} and the ciphertexts $\vec{\boldsymbol{b}}, \vec{\boldsymbol{b}}'$.

The following notation is used in the description of our protocol in Figure 3.

Notation 8. *Let $N, d, m \in \mathbb{N}$, and $\lambda \geq 0$. The circuit $C_{Add,\lambda} : \left(\mathbb{Z}_N^{d \times d}\right)^m \to \mathbb{Z}_N^{d \times d}$ on input m type-L encodings of matrices $A_1, \ldots, A_m \in \mathbb{Z}_N^{d \times d}$, outputs the sum $\sum_{j=1}^{m} A_j + \lambda I \mod N$. The circuit $C_{Add} : \left(\mathbb{Z}_N^d\right)^m \to \mathbb{Z}_N^d$ on input m type-A encodings of vectors $\vec{b}_1, \ldots, \vec{b}_m \in \mathbb{Z}_N^d$, outputs the sum $\sum_{j=1}^{m} \vec{b}_j \mod N$. For an LHE scheme $\mathcal{E} = (KG, Enc, Dec, Eval)$, a public key pk, and encryptions $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_m$ of type-L encoding of matrices $A_1, \ldots, A_m$, we use $Eval(pk, \mathrm{Add}_\lambda, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_m)$ to denote running the Eval algorithm on input the circuit $C_{Add,\lambda}$ and the ciphertexts $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_m$. For encryptions $\vec{\boldsymbol{b}}_1, \ldots, \vec{\boldsymbol{b}}_m$ of type-A encoding of vectors $\vec{b}_1, \ldots, \vec{b}_m$, we use $Eval\left(pk, Add, \vec{\boldsymbol{b}}_1, \ldots, \vec{\boldsymbol{b}}_m\right)$ to denote running the Eval algorithm on input the circuit C_{Add} and the ciphertexts $\vec{\boldsymbol{b}}_1, \ldots, \vec{\boldsymbol{b}}_m$.*

Proof of Theorem 5. From Equation 1, $\log N = O(d \log d + \ell \cdot d + d \log(n^2 + \lambda)) = O(d \log(d \cdot n \cdot \lambda \cdot 2^\ell))$. Since $p_i \leq 2^{62}$ for every $i \in [t]$ (i.e., is constant) then $t = O(\log N)$. As discussed above, computing $A, \vec{b}$ (multiplying two matrices, resp.) requires $O(m \cdot d\lceil \frac{d^2}{\mathsf{sl}} \rceil)$ ($O(d\lceil \frac{d^2}{\mathsf{sl}} \rceil)$, resp.) homomorphic operations. The bounds follow because each operation is performed on t copies. □

6 EXPERIMENTAL RESULTS

We implemented our protocol and ran extensive performance benchmark, which we compare against [6]. We note that the machine-learning accuracy is identical to that of [6], because we implement the same functionality and with same precision. Performance difference emanates from using different LHE (RLWE-based rather than Paillier), with our protocol modifications to enable employing the SIMD property of the LHE. In this section we describe our experiments.

6.1 Settings

We tested our protocol with parameter settings similar to [6]:

Parties. We ran experiments with 2 servers and $m = 10$ data owners with horizontally-partitioned data, where each data owner holds 10% of the rows in the data matrix.

Hardware. We assigned $M = 40$ cores for each data owner and server. The cores we used were Xeon E5-2630 v4 running in 2.20GHz. Such CPUs are standard in servers and personal computers.

Software. We implemented the protocol in C++, building on top of HElib [11] for the LHE implementation, with security parameter set to 80-bit as in [6]. We stress that we used HElib only as an LHE (and not as fully-homomorphic encryption), i.e., only to add ciphertexts and multiply ciphertexts by plaintexts.

HElib supports plaintext spaces $\mathbb{Z}_{p^r}$, where p is prime, $r \geq 1$ is an integer, and $p^r < 2^{62}$; as well as packing (SIMD) of sl plaintext

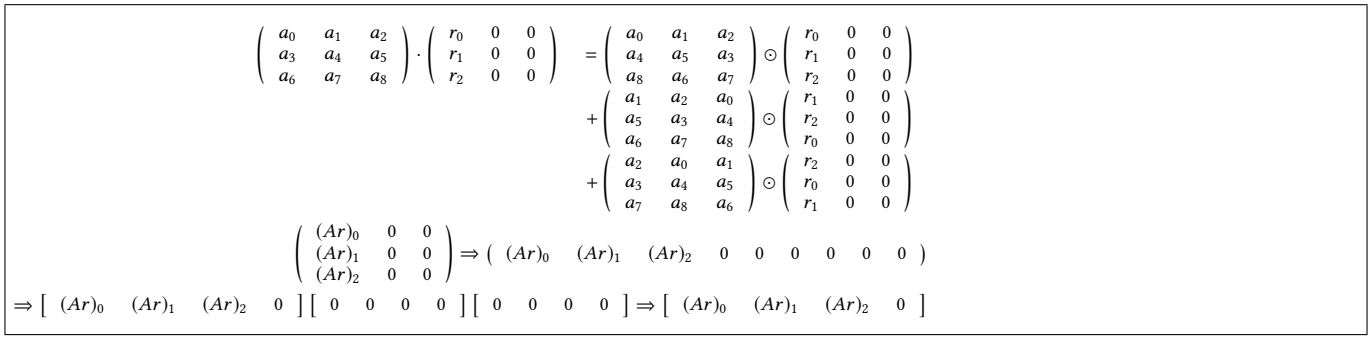

Figure 7: Example of multiplying a type-L encoding by a type-M encoding (top) and truncated serialized output encoding (bottom).

values (aka "slots") in each ciphertext. The number of slots, sl, is determined in HElib during key-generation, and depends on parameters including the plaintext modulus p^r and the security parameter. For our experiments, we changed the code of HElib to enforce sl to be as large as a lower-bound value of our choice (e.g., d^2), thus gaining some control over the value of sl.

Data and parameters. The data matrix and response vector were $X \in [0,1)^{n \times d}$ and $\vec{y} \in [0,1)^n$ with precision of $\ell = 3$ decimal digits. We scaled all values by 1000, and embedded the resulting integers in the HElib plaintext space we used. We ran experiments with $m = 10$ data owners, $n = 1000$ data-instances, $d = 10, 20, 30, 40$ features, $\ell = 3$, and $M = 40$ cores. These choices of m, ℓ, n, d, M are as in [6]. In addition, we test our system on $d = 100$ features (whereas [6] only retport run-times up to $d = 40$). For $d \leq 40$, we used $t = 40$ primes in the CRT representation of plaintexts ($t = 106$ when $d = 100$); each core was allocated one ring from the CRT representation ($2 - 3$ rings when $d = 100$), and executed the computation in that ring.

6.2 Experiments and Results

Correctness. We first experimentally validated the correctness of our implementation by comparing the output model in our protocol to the model computed by ridge-regression on the same data in plaintext. All our experiments returned identical models.

Run-time. A main objective was to measure the run-time performance when focusing on steps where the LHE choice affects run-time, because this is where our protocol differs from [6]. In contrast, steps computing on plaintext are identical to those of [6], and hence we expect identical run-times. We therefore report a fine-grained measurement, reporting the run-time of each step, and focus not only on the total run-time but also on the total time for homomorphic operations (LHE op.); See Table 1, last two columns, where G vs. L in the 2nd column indicate whether the run-time is as reported in [6] (G) vs. as we measured for LoPED (L). Each row indicates run-time on different values of $d = 10, 20, 30, 40, 100$. (For $d = 100$ [6] did not report run-times.)

In Table 1, columns 3–10, we report the run-time in the individual steps: Key Generation (Figure 2, Setup Phase, Step 1); Local Computation by data owner (Figure 2, Input-Uploading Phase, Step 1); Encrypt (Figure 2, Input-Uploading Phase, Step 2); Merge (Figure 3, Step 1); Mask (Figure 3, Step 3a); Decrypt (Figure 3, Step 3b); Solve (Figure 3, Step 4); Unmask (Figure 3, Step 5). We note that [6] reported only the cumulative time for "Local Computation" together with "Encrypt", and "Decrypt" together with "Solve"; we present their reported times in the rightmost of these two steps. We emphasis in bold the faster run-time for each step (except for the Decrypt and Solve steps where [6] reported the cumulative times so we were unable to determine which protocol was faster).

The results demonstrate the speedup gained by LoPED in the homomorphic operations. For example, for $d = 40$, the masking step in [6] took 100.94 seconds, while LoPED took only 2.91 seconds, which is faster by a factor of ×35. The local-computations and encryption steps, on the other hand, were faster for [6] with 26.13 seconds, compared to 34.39 seconds in LoPED. The overall time for the LHE operation was 143.08 seconds for [6] and 60.21 seconds for LoPED. The results support our analytical results, showing that our use of SIMD reduced the number of LHE operations in matrix multiplication (the dominating component in the homomorphic operations) from d^3 to $d \cdot \lceil \frac{d^2}{\mathsf{sl}} \rceil$ operations. A graph comparing the run-time on LHE operations in [6] vs. LoPED, and showing substantial speedup in LoPED, is given in Figure 8.

Communications bandwidth. Packing also reduces the number of communicated ciphertexts from each data owner and from $\mathcal{S}_1$ from d^2 to $d \cdot \lceil d^2/\mathsf{sl} \rceil$. In contrast, our CRT representation increases the number of communicated ciphertexts, because each plaintext value is encrypted in t distinct plaintext rings $\mathbb{Z}_{p_i}$, resulting in $\times t$ increase in the number of transmitted ciphertexts. Additionally, ciphertexts in HElib are larger than Paillier. In our experiments, for example, for $d = 40$ the data owners need to communicate 177MB to $\mathcal{S}_1$ (cf. less than 1MB in [6]). We note that the increase in communication in LoPED stems from the use of both CRT and the (less communication-efficient) HElib, which are unavoidable when using current RLWE-based LHE schemes implementations. In certain scenarios the increase in communication is offset by the advantages of basing security on alternative (and post-quantum secure) security assumptions, and attaining considerably faster homomorphic computation runtime as in LoPED.

6.3 System Configuration Optimization

For correctness, arithmetic operations in our protocol require a large ring of size N, as specified in Equation 1 (page 5). For example, $N > 2^{2501}$ when $d = 40$; see Table 2. Since HElib does not

d	L/G	Key Gen.	Local Comp.	Encrypt	Merge	Mask	Decrypt	Solve	Unmask	LHE op.	Total Processing
10	G	**0.21**	(*)	**1.1**	0.03	1.21	(**)	0.56	0.04	1.24	**2.94**
	L	10.68	0.02	5.25	**0.01**	**0.11**	0.18	same	same	**0.12**	6.16
20	G	**0.32**	(*)	3.88	0.12	7.96	(**)	2.15	0.14	8.08	14.25
	L	10.58	0.08	9.86	**0.02**	**0.41**	1.07	same	same	**0.43**	**13.74**
30	G	**0.18**	(*)	**8.34**	0.26	24.76	(**)	4.8	0.29	25.02	38.45
	L	13.22	0.19	19.71	**0.05**	**1.10**	3.05	same	same	**1.15**	**29.18**
40	G	**0.38**	(*)	**26.13**	0.62	100.94	(**)	14.72	0.67	101.56	143.08
	L	19.81	0.36	34.03	**0.11**	**2.91**	7.42	same	same	**3.01**	**60.21**
100	L	188.88	10,298.72	0.51	11.14	3.75	0.01				

Table 1: . Running times on $n = 1000$, $\ell = 3$, $M = 40$, $m = 10$ and $d = 10, 20, 30, 40, 100$ (rows) of LoPED (L) vs. [6] (G). Solve and Unmask steps operate on plaintexts and are identical to those of [6]; we indicate this by writing "same". The one-but-last column reports the total time of homomorphic computations (LHE op.), i.e., the time to Merge and Mask. The last column reports the total time of all operations other than Key Gen (which occurs offline during Setup). Boldface indicates the faster time. All values are in seconds. [6] reported cumulative time of: (*) Local Comp. and Encrypt; (**) Decrypt and Solve.

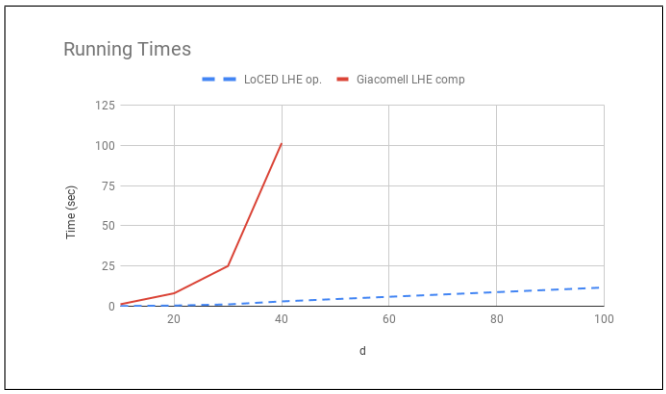

Figure 8: Run-time of LHE operations (y-axis, in seconds), on increasing number of features $d = 10, 20, 30, 40, 100$ (x-axis), in [6] (red solid line) and LoPED (dashed blue line).

d	$\log_2 N$	$\log_2 p_i$	LHE op.(sec)
10	616	16	0.21
20	1241	32	0.89
30	1870	47	2.04
40	2501	62	4.76

Table 2: Ring-size shrinkage examples from $\log_2 N$ bits (2nd col.) to $\log_2 p_i$ bits (3rd col.), and time (in seconds) of a linearly-homomorphic-operation on our packed ciphertexts in ring $\mathbb{Z}_{p_i}$ (4th col.), for parameters: $d = 10, 20, 30, 40$ (rows), and $n = 1000$, $\ell = 3$, $t = 40$.

support plaintext rings larger than 2^{62}, we used CRT to compute our protocol over smaller rings $\mathbb{Z}_{p_i}, i \in [t]$s.t. $\prod p_i = N$; See Section 4.

We next discuss how we set the lower-bound on the number of slots sl, and how we chose t and $p_1, \ldots, p_t$. We stress that the configuration is pre-computed in an offline stage of our HElib parameter configuration, and does not affect the run-time of our protocol. The optimal configuration does not depend on the data, but only on the parameters N, d and the number of cores M available to each server. Our recommended configurations for various N, d, M values will be published in our open-source library upon paper publication. Our configuration was chosen to optimize run-time of $\mathcal{S}_1$ and $\mathcal{S}_2$; albeit our code can compute offline configuration parameters for other objectives (e.g. data owners' running time).

Our configuration for $d \leq 40$ is computed as follow (see discussion on $d > 40$ below). First we set $t = 40$ to be the number of available cores, and set d^2 to be the lower-bound we enforce on the number of slots sl of roughly d^2. Second, we computed the bit-size of t primes s.t. $\prod p_i > N$ for N at least as large as Equation 1; see Table 2 for some examples. Specifically, the bit-size of the primes p_i is computed to be in the range $[\sqrt[40]{N}, 2\sqrt[40]{N}]$. Third, we executed an exploration of the complexity of homomorphic operations for the above setting of t, $|p|$ and the lower-bound on sl. The exploration is done by sampling various primes p of the specified bit-size, and configuring HElib with p as the plaintext ring and the specified lower-bound on the number of slots. This results in various parameter settings. We tested the run-time of homomorphic operations for each, and chose the 40 best candidates.

Figure 9 illustrates our experimental results on $d = 30$, showing that setting sl $\approx d^2$ is indeed desired. Specifically, Figure 9 shows the run-time of homomorphic operations (y-axis) as a function of the number of slots sl (x-axis). It shows that run-time decreases rapidly as sl grows, attaining its minimum essentially at sl $= d^2$ (and very close to the minimum already with sl roughly $d^2/3$).

For $d > 40$ we deviate from the above in two ways. First, $t = 40$ primes no longer suffice when restricted to primes of bit-size up to 62 as in HElib. Instead, we set t to be the smallest multiple of the number of available cores so that $2\sqrt[t]{N} \leq 62$. Second, setting sl $\geq d^2$ is no longer effective, because such a large sl adversely affects the ciphertext size and time for homomorphic operations. Instead we perform the explorations with various sl values of up to a few thousands. Out of the explored values we choose, as before, the t best for our configuration.

7 CONCLUSIONS

We presented a new protocol, LoPED, for privacy-preserving ridge-regression on packed encrypted data. The protocol works in the two-server model, and uses only Linearly Homomorphic Encryption (LHE). The protocol builds on [6], while introducing new components to allow computing on packed encrypted data in a Single Instruction Multiple Data (SIMD) fashion, to gain substantial speedup of the homomorphic computation. Analytically, the speedup is by a factor of up to $\Theta(d^2)$ in the number of homomorphic operations.

We implemented our protocol, with LHE as implemented in the HElib [11] implementation of the BGV [2] encryption with SV [22]

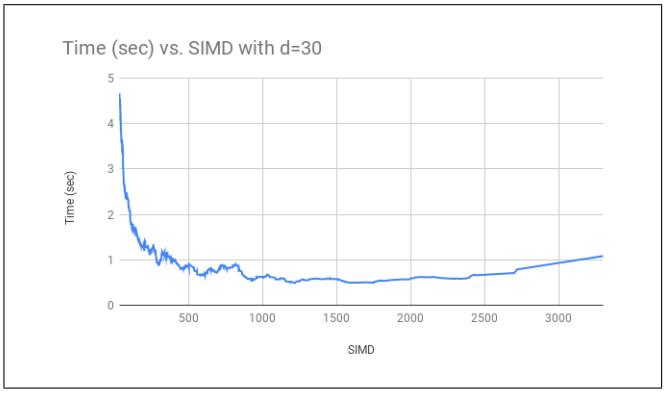

Figure 9: Run-time of LHE operations (y-axis in seconds) on $d = 30$ and increasing number of slots sl (x-axis).

SIMD optimization. We ran extensive experiments, demonstrating run-time speedup in the homomorphic operations. For example, ×33 speedup in run-time of homomorphic computation on $d = 40$ features and $n = 1000$ data-instances; and ×2.4 speedup in total time. We note that the total time accounts, on top of the homomorphic operations, also for computations on plaintext where LoPED is identical to [6], and for the time to encrypt and decrypt; since much of the total time is identical to [6] we expect lesser speedup, as indeed is the case.

In future work we intend to further improve performance and encapsulate our methods to develop an efficient and operable system to support federated privacy-preserving linear regression.

8 ACKNOWLEDGEMENTS AND FUNDING

We thank Irene Giacomelli and Alon Rosen for helpful discussions. This work was partially supported by the BLAVATNIK INTERDISCIPLINARY CYBER RESEARCH CENTER (iCRC) and the BIU Center for Applied Research in Cyber Security (both in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office), ISF grants 1861/16 and 1399/17, and AFOSR Award FA9550-17-1-0069.

REFERENCES

[1] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine Learning Classification over Encrypted Data. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. https://www.ndss-symposium.org/ndss2015/machine-learning-classification-over-encrypted-data

[2] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. ACM, New York, NY, USA, 309–325. https://doi.org/10.1145/2090236.2090262

[3] Hao Chen, Kim Laine, and Rachel Player. 2017. Simple Encrypted Arithmetic Library - SEAL v2.1. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.), Vol. 10323. Springer, 3–18. https://doi.org/10.1007/978-3-319-70278-0_1

[4] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* 2012 (2012), 144.

[5] Pierre-Alain Fouque, Jacques Stern, and Jan-Geert Wackers. 2002. CryptoComputing with Rationals. In *Financial Cryptography, 6th International Conference, FC 2002, Southampton, Bermuda, March 11-14, 2002, Revised Papers*. 136–146. https://doi.org/10.1007/3-540-36504-4_10

[6] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. 2018. Privacy-Preserving Ridge Regression with only Linearly-Homomorphic Encryption. In *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings (Lecture Notes in Computer Science)*, Bart Preneel and Frederik Vercauteren (Eds.), Vol. 10892. Springer, 243–261. https://doi.org/10.1007/978-3-319-93387-0_13

[7] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. JMLR.org, 201–210. http://proceedings.mlr.press/v48/gilad-bachrach16.html

[8] Oded Goldreich. 2004. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press. https://doi.org/10.1017/CBO9780511721656

[9] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. ACM, New York, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[10] Thore Graepel, Kristin Lauter, and Michael Naehrig. 2013. ML Confidential: Machine Learning on Encrypted Data. In *Proceedings of the 15th International Conference on Information Security and Cryptology (ICISC'12)*. Springer-Verlag, Berlin, Heidelberg, 1–21. https://doi.org/10.1007/978-3-642-37682-5_1

[11] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *34rd Annual International Cryptology Conference, CRYPTO 2014*. Springer Verlag.

[12] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. 2018. Secure Outsourced Matrix Computation and Application to Neural Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 1209–1222. https://doi.org/10.1145/3243734.3243837

[13] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 1651–1668. http://dl.acm.org/citation.cfm?id=3277203.3277326

[14] Seny Kamara, Payman Mohassel, and Mariana Raykova. 2011. Outsourcing Multi-Party Computation. Cryptology ePrint Archive, Report 2011/272.

[15] Yehuda Lindell and Benny Pinkas. 2000. Privacy Preserving Data Mining. In *Advances in Cryptology — CRYPTO 2000*, Mihir Bellare (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–54.

[16] Gary C. McDonald. 2009. Ridge regression. *Wiley Interdisciplinary Reviews: Computational Statistics* 1, 1 (2009), 93–100.

[17] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 35–52. https://doi.org/10.1145/3243734.3243760

[18] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 19–38. https://doi.org/10.1109/SP.2017.12

[19] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. 2013. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 334–348. https://doi.org/10.1109/SP.2013.30

[20] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM* 56, 6 (2009), 34:1–34:40.

[21] Kenneth H. Rosen. 1993. *Elementary number theory and its applications (3. ed.)*. Addison-Wesley.

[22] Nigel P Smart and Frederik Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, codes and cryptography* (2014), 1–25.

[23] Paul S. Wang, M. J. T. Guy, and James H. Davenport. 1982. P-adic reconstruction of rational numbers. *ACM SIGSAM Bulletin* 16, 2 (1982), 2–3. https://doi.org/10.1145/1089292.1089293

[24] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*. IEEE Computer Society, Washington, DC, USA, 162–167. https://doi.org/10.1109/SFCS.1986.25