A Report on the Performance of Multi-Secret-Key HE for Private Average Aggregation

A. Pedrouzo-Ulloa

atlanTTic Research Center, Universidade de Vigo
apedrouzo@gts.uvigo.es

Abstract. We describe and evaluate the performance of the solution proposed in [12] for the execution of a private aggregation protocol based on the use of secret multi-key homomorphic encryption. For this type of schemes, input ciphertexts coming from each Data Owner are only encrypted under its individual secret key. However, after aggregation, all the involved Data Owners must collaborate for correctly decrypting. The discussed solution was published in [12] and has been presented at several international venues related to private machine learning, homomorphic encryption, and distributed machine learning. In the GitHub repository `https://github.com/apedrouzoulloa/mkagg/docs`, we include the code used for evaluation and several related presentations.

1 High-level Design of [12]

1.1 Building blocks

Additive Secret Shares of Zero: Given L Data Owners (DOs), we can generate L uniformly random additive shares satisfying that their addition is equal to zero. The protocol is as follows:

1. The i-th DO ($\forall i$) generates a set of $(L-1)$ uniformly random elements $r_{i,j}$ for all $j \neq i$. Next, the i-th DO computes $r_{i,i} = -(\sum_{j:j\neq i} r_{i,j})$. All $r_{i,j}$ satisfy the relation $\sum_j r_{i,j} = 0$.
2. The i-th DO ($\forall i$) sends, $r_{i,j}$ to the j-th DO, $\forall j \neq i$.
3. The j-th DO ($\forall j$) computes $\mathsf{share}_j = r^{(j)} = \sum_i r_{i,j}$.

Rounding polynomial elements: Let $\lfloor \boldsymbol{a} \rceil_p$ be the scaling and rounding of each coefficient of $\boldsymbol{a} \in R_q^N$ to its nearest integer, where R_q denotes the quotient polynomial ring $\mathbb{Z}_q[x]/(x^n+1)$. Here, the use of Lemma [2, Lemma 1] is fundamental for the construction of the private aggregation protocol. We include it next for completeness.

Lemma 1 (Lemma 1 **[2]).** *Let $p|q$, $\boldsymbol{x} \leftarrow R_q^N$ and $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{e} \bmod q$ for some $\boldsymbol{e} \in R_q^N$ with $\|\boldsymbol{e}\|_\infty < B < q/p$. Then $\Pr\left(\lfloor \boldsymbol{y} \rceil_p \neq \lfloor \boldsymbol{x} \rceil_p \bmod p\right) \leq \frac{2npNB}{q}$.*

This lemma can be used to remove the error term associated to each encryption. Given $(a, b = as + e + q/p \cdot m)$, we compute $\lfloor b \rceil_p = \lfloor as + e \rceil_p + m$ which, by Lemma 1, is different than $\lfloor as \rceil_p + m$ with a certain probability $\Pr(\mathsf{Ev})$. The upper bound of the probability $\Pr(\mathsf{Ev})$ depends inversely on q.

Distributed decryption: Given $(a, b = as + e) \in R_q^2$ s.t. $s = \sum_{i=1}^{L} s_i$ where all $s_i \in R_q$, applying modulus switching [12] from q into p, we get

$$\left(\lfloor a \rceil_p, \lfloor b \rceil_p = \left\lfloor \lfloor a \rceil_p s + (p/q \cdot a - \lfloor a \rceil_p) \cdot s + p/q \cdot e \right\rceil\right).$$

By applying now Lemma 1, the error term e is removed with a certain probability, finally having:

$$\lfloor b \rceil_p = \left\lfloor a \underbrace{s}_{\sum_i s_i} \right\rceil_p = \left\lfloor \lfloor a \rceil_p \underbrace{s}_{\sum_i s_i} + \underbrace{(p/q \cdot a - \lfloor a \rceil_p) \cdot}_{e_a} \underbrace{s}_{\sum_i s_i} \right\rceil. \quad (1)$$

From equation (1), we can obtain the magnitude of the difference $e_{\mathsf{distributed}} = \lfloor as \rceil_p - \sum_i \lfloor as_i \rceil_p$. This term must be removed for the correctness of the distributed decryption protocol executed after each aggregation round. Assuming that each s_i is bounded by B, and due to $\|e_a\|_\infty < 1/2$, the magnitude of this remaining error term is bounded by nLB.

1.2 General Description

General Overview: Figure 1 gives a high-level description of the required steps for our protocol. The workflow for a round of the private aggregation protocol is as follows:

1. DOs encrypt their inputs with their individual secret key.
2. The aggregator homomorphically aggregates the encrypted updates.
3. DOs decrypt the aggregated update. This is done collaboratively in version-1 of the protocol and locally in version-2 of the protocol.

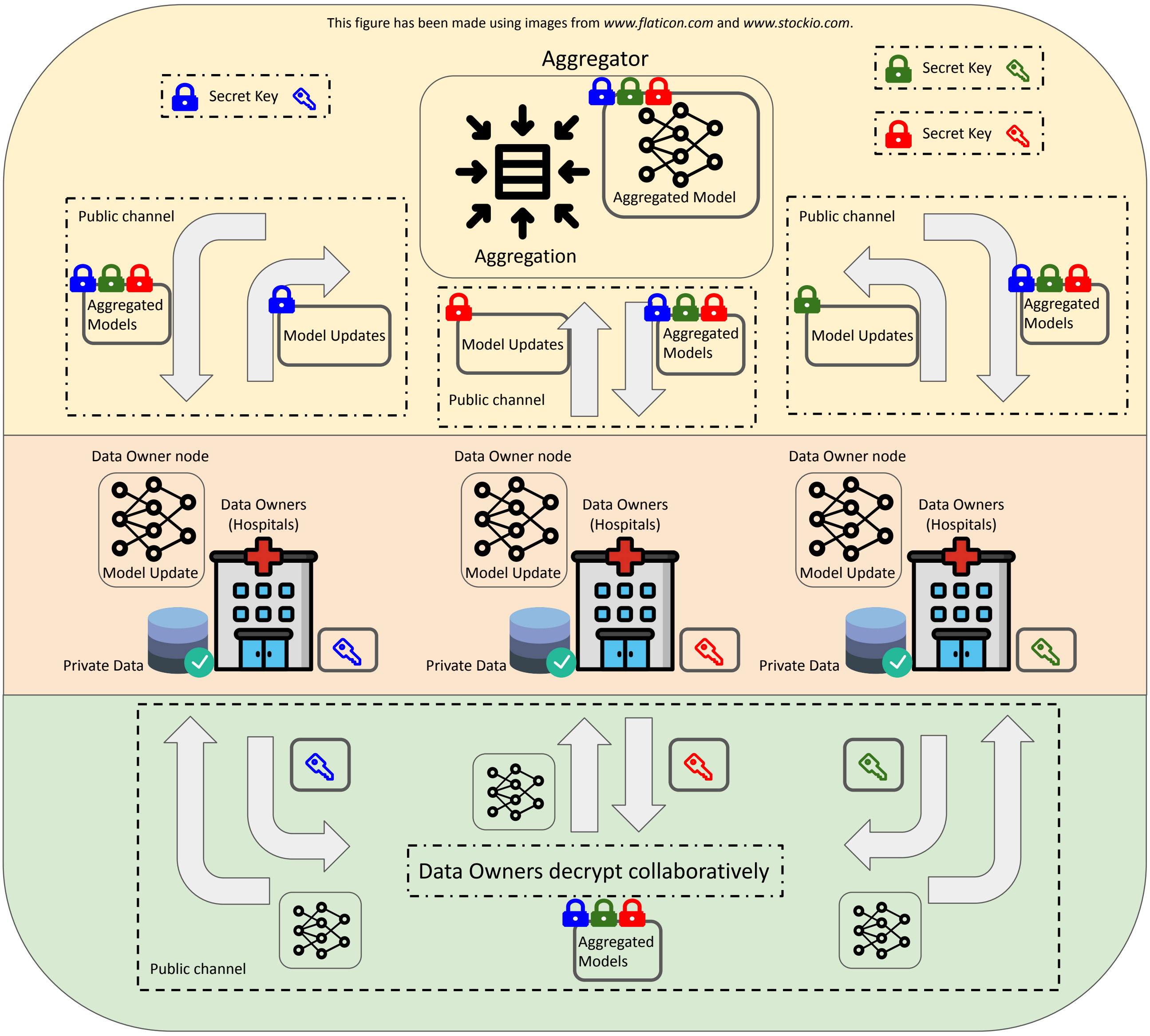


Fig. 1. High-level description of the workflow for the private aggregation protocol based on Multi-Key HE.

Algorithmic description (version-1): Current works making use of Threshold RLWE-based HE [11,1] define a collaborative key setup phase to generate a joint public key pk associated to several secret keys s_i. This results in a pair $(\mathsf{sk} = s, \mathsf{pk} = (a, as + e))$, where the i-th DO has a s_i s.t., $\sum_{i=1}^{L} s_i = s$. In contrast, this protocol optimizes the use of HE for the case of private federated average aggregation: by assuming the CRS model, ciphertexts encrypted under different secret keys can be directly aggregated. We include next a high-level description of our proposed secure aggregation primitive.

In the CRS model, each DO has access to a common random string, allowing us to assume that all DOs share the same secret seed K. If all DOs know the round T in which they are participating, they can utilize a pseudorandom function $\mathsf{PRF}_K(T)$ to generate polynomials a that are computationally indistinguishable from uniformly random polynomials in R_q.

Additionally, we assume that all DOs have run the protocol described in Section 1.1 to generate uniformly random polynomial shares. As a consequence, the i-th DO holds $\mathsf{share}_i = r^{(i)}$. For simplicity of exposition, we assume here that the model updates m_i coming from the i-th DO fit inside of only one secret-key ciphertext (a, b_i). If this is not true, and several ciphertexts are needed to encrypt m_i, then the $\mathsf{PRF}_K(T)$ can also be used to generate a set of N_{Ctxts} different a terms per each round, i.e., $\{a_1, \ldots, a_{N_{\mathsf{Ctxts}}}\}$. Finally, the i-th DO would send its N_{Ctxts} secret-key ciphertexts to the aggregator $\{(a_1, b_{i,1}), \ldots, (a_{N_{\mathsf{Ctxts}}}, b_{i,N_{\mathsf{Ctxts}}})\}$. Here, all $\{a_1, \ldots, a_{N_{\mathsf{Ctxts}}}\}$ can be removed because only the b terms are needed for running the aggregation step.

Each private aggregation round (version-1) of the protocol is as follows:

1. DOs encrypt their inputs: The i-th DO ($\forall i$) encrypts its model update m_i with its secret key s_i as
$$(a, b_i) = (a, a(s_i + r^{(i)}) + e_i + q/p \cdot m_i),$$
which can be compressed by a half by only sending b_i because a is publicly known (i.e., computable with $\mathsf{PRF}_K(T)$ for the T-th round).
2. Aggregation step: After receiving all b_i polynomial terms, a semi-honest aggregator can directly compute:
$$\begin{aligned} b = & \sum_i b_i \\ = & a(s + \underbrace{\sum_i r^{(i)}}_{0}) + e \\ = & a \underbrace{s}_{\sum_i s_i} + \underbrace{e}_{\sum_i e_i} + q/p \cdot \underbrace{m}_{\sum_i m_i}, \end{aligned}$$
which corresponds to $\mathsf{Enc}(\mathsf{sk} = s, m)$, the desired encrypted aggregation. Finally, the aggregator sends back $\mathsf{share}^{(\mathsf{agg})} = \lfloor b \rceil_{p'}$ to the DOs. We elaborate in Section 1.3 on the requirements imposed on $p' > p$ for correctness.
3. Distributed decryption: Given $\mathsf{Enc}(\mathsf{sk} = s, m)$ s.t. $s = \sum_i s_i$. This protocol is as follows:
   (a) The i-th DO ($\forall i$) computes
   $$\mathsf{share}^{(i)} = \lfloor a s_i \rceil_{p'},$$
   and makes it available to the other DOs.
   (b) All DOs compute
   $$\left\lfloor \mathsf{share}^{(\mathsf{agg})} - \sum_i \mathsf{share}^{(i)} \right\rceil_p,$$
   which is equal to m with probability higher than $1 - 2^{-\kappa}$, whenever the encryption parameters are chosen according to the bounds from Section 1.3.

Communication cost (version-1): We include in Table 1 the communication cost per party for this version-1 of the private aggregation protocol. We assume that the number of model parameters $N_{\mathsf{ModelParam}}$ is high enough so that ciphertexts are fully packed. Note that the decryption share of each DO must be sent to the other $L-1$ DOs. At worst-case, if each DO uses $L-1$ party-to-party channels to transmit the decryption share, the communication cost per party during the phase of distributed decryption is $(L-1)N_{\mathsf{ModelParam}} \cdot \log_2 p'$.

Input per DO	Decryption share per DO	Aggregator output	Decrypted result
$N_{\mathsf{ModelParam}} \cdot \log_2 q$	$N_{\mathsf{ModelParam}} \cdot \log_2 p'$	$N_{\mathsf{ModelParam}} \cdot \log_2 p'$	$N_{\mathsf{ModelParam}} \cdot \log_2 p$

Table 1. Communication costs per party in each round of the version-1 of the private aggregation protocol from Section 1.

An interesting variant (version-2): The previously presented variant version-1 of the protocol is characterized by the following communication workflow:

- All DOs send their encrypted inputs to the aggregator.
- After aggregation, DOs engage in a distributed decryption, for which all the partial decryptions $\mathsf{share}^{(i)}$ must be exchanged among all DOs.

While there are different ways of optimizing the communication flow for the distributed decryption, the most straightforward implementation requires that each DO sends its partial decryption to the rest of DOs. This entails a communication cost which grows quadratically with the number of DOs. We explain next how, by considering a simple modification in the protocol, the communication cost can be reduced from quadratic to linear.

First, similarly to version-1, we assume for simplicity of exposition that the model updates m_i fit inside of only one secret-key ciphertext. Then, the same $\mathsf{PRF}_K(T)$ can also be used by the DO_i to generate a uniformly random term $\mathsf{mask}_i \in R_p$ per each round.[1] Now, each private aggregation round (version-2) is as follows:

1. DOs encrypt their inputs: The i-th DO ($\forall i$) encrypts its model update m_i with its secret key s_i as
$$(a, b_i) = (a, a(s_i + r^{(i)}) + e_i + q/p \cdot (m_i + \mathsf{mask}_i),$$
where mask_i is a uniformly random polynomial in R_p which can be generated with the $\mathsf{PRF}_K(T)$ if K remains only known by the DOs.
Finally, DOs send to the aggregator the pair $(b_i, \mathsf{share}^{(i)} = \lfloor a s_i \rceil_{p'})$.
2. Aggregation step: After receiving all b_i and $\mathsf{share}^{(i)}$ polynomial terms, a semi-honest aggregator can directly compute:
$$\begin{aligned} b = & \sum_i b_i \\ = & a(s + \underbrace{\sum_i r^{(i)}}_{0}) + e \\ = & a \underbrace{s}_{\sum_i s_i} + \underbrace{e}_{\sum_i e_i} + q/p \cdot (\underbrace{m}_{\sum_i m_i} + \underbrace{\mathsf{mask}}_{\sum_i \mathsf{mask}_i}), \end{aligned}$$

[1] In contrast, if N_{Ctxts} ciphertexts are needed to encrypt the update m_i of one aggregation round, then we must generate N_{Ctxts} uniformly random mask terms per each DO.

which corresponds to $\mathsf{Enc}(\mathsf{sk} = s, m + \mathsf{mask})$, the desired encrypted aggregation. Afterwards, the aggregator computes $\mathsf{share}^{(\mathsf{agg})} = \lfloor b \rceil_{p'}$ and also

$$\left\lfloor \mathsf{share}^{(\mathsf{agg})} - \sum_i \mathsf{share}^{(i)} \right\rceil_p ,$$

which is equal to $m + \mathsf{mask}$ with probability higher than $1 - 2^{-\kappa}$, whenever the encryption parameters are chosen according to the bounds from Section 1.3.
Finally, the aggregator sends back $m + \mathsf{mask}$ to the DOs.

3. Distributed decryption (locally removing mask): Given $m + \mathsf{mask}$, all DOs can locally compute the polynomial term mask by only having access to the $\mathsf{PRF}_K(T)$, and finally recover m.

Communication cost (version-2): We include in Table 2 the communication cost per party for this version-2 of the private aggregation protocol. Again, we assume that the number of model parameters $N_{\mathsf{ModelParam}}$ is high enough so that ciphertexts are fully packed. It is worth noting how, by slightly increasing the input size per DO, this second version avoids the problem of sharing among all DOs the decryption shares.

Input per DO	Aggregator output	Decrypted result
$N_{\mathsf{ModelParam}} \cdot (\log_2 q + \log_2 p')$	$N_{\mathsf{ModelParam}} \cdot \log_2 p$	$N_{\mathsf{ModelParam}} \cdot \log_2 p$

Table 2. Communication costs per party in each round of the version-2 of the private aggregation protocol from Section 1.

Threat model: Both versions of the protocol provide an effective solution for private average aggregation in FL with a semi-honest server. For its application in a more realistic FL training, this protocol can be paired with differential privacy techniques, as it is done in [13] combining single-key HE and DP, to cover threats coming from $L - 1$ colluding semi-honest DOs (out of L) that aim at gathering information about the remaining DO data.

Considered assumptions: First, the protocol [12] assumes a CRS model, where all Data Owners (DOs) have access to the same PRF. Using $\mathsf{PRF}_K(T)$, for the T-th encryption round and with the same secret uniformly random seed K, ensures that all DOs will be able to generate the same common mask a for their distinct RLWE/LWE samples. We remark that if we follow [12] and the input T is increased after each call to $\mathsf{PRF}_K(\cdot)$, then the same "a" value will never be used more than once to encrypt the local model updates.

Second, we assume that DOs will have distinct secret keys. That is, each DO will encrypt her own data m_i with her own secret key s_i (but using the same mask a per encryption round which was shared with other DOs). Finally, during the aggregation, the semi-honest server will compute the sum $\sum_i m_i$, which is encrypted under the aggregated secret key $\sum_i s_i$.

General linear combinations for the aggregation: The described protocols can be easily adapted to work for any linear combination $\sum_i \lambda_i m_i$ of the encrypted model updates m_i. One possibility is that each DO uses $\lambda_i^{-1} a$ instead of a during the encryption phase. We refer the reader to [12] for more details and other alternative options.

1.3 A discussion of the protocol security and resilience against attacks

Partial decryptions: In the aggregation protocol, DOs make repeated calls to $\mathsf{PRF}_K(T)$ as a means to generate all the a polynomial terms which are needed for encryption. Here, K is a uniformly random seed and the same T is not used more than once during the whole protocol execution. Therefore, by emulating a random oracle with the $\mathsf{PRF}_K(T)$ function, all the generated polynomials a are computationally indistinguishable from a set of independent and uniformly random polynomials.

Following this line of reasoning, given a pair of independent and uniformly random terms $a, u \leftarrow R_q$, then if an algorithm $\mathcal{A}(a, \lfloor u \rceil_{p'}, \lfloor as_i \rceil_{p'})$ can distinguish between $(a, \lfloor u \rceil_{p'})$ and $(a, \lfloor as_i \rceil_{p'})$, $\mathcal{A}$ can be used to distinguish with probability $1 - 2^{-\kappa}$ the RLWE sample $(a, as_i + e)$ from the pair (a, u).

Consequently, the security of the private aggregation protocol [12] relies on the difficulty of breaking the RLWE indistinguishability assumption. Alternatively, the security of the protocol could also rely on the difficulty of breaking the LWE assumption if ciphertexts are defined under the LWE problem. We refer the reader to the original work [12] for more details.

Threshold (L-out-of-L) HE and $\mathsf{IND\text{-}CPA^D}$ (when $L = 1$): Till very recently, the cryptographic community believed that only approximate HE presented this vulnerability and that exact HE schemes like BFV [4,10], BGV [5] or CGGI [8] were invulnerable to this type of attacks, hence being these schemes naturally safe under $\mathsf{IND\text{-}CPA^D}$. However, a couple of new works [6,7] have shown that this vulnerability is also present in practice for exact schemes, as soon as they present a non-negligible probability of incorrect decryption. Both works exemplify how this vulnerability can be exploited to implement effective key-recovery attacks against several mainstraim HE libraries.

In both works [6,7], the authors illustrate how this type of attacks is especially relevant when dealing with threshold HE schemes. Fortunately, the authors also argue several countermeasures which can be applied when making use of threshold HE. In particular, the addition during decryption of an adequate smudging noise with λ-independent variance can do the job.

In view of the countermeasures discussed in those works, we elaborate now on the main reasons which make the proposed protocol in [12] naturally safe in the multi-key setting:

- The primary weakness evidenced by approximate HE schemes is the leakage of internal noise from decrypted ciphertexts. It is worth mentioning that the protocols from Section 1 use the rounding building block to compute the partial decryptions. This guarantees the elimination in the distributed decryption of the internal noise, which only would not be removed with a probability upper bounded by $2^{-\kappa}$.
- The protocols from Section 1 follow the blueprint of the BFV scheme, hence belonging to the category of "exact" schemes. In this situation, the main source of weakness exploited by the key-recovery attacks from [6,7] consists in the existence of a non-negligible probability of having decryption errors. Note how our κ parameter allows us to precisely control the probability of decryption error during the whole FL training pipeline. For example, a choice like $\kappa = 128$ upper bounds by 2^{-128} the probability of having at least one error during decryption in the whole training.

Protocol parameters Table 3 details all the parameters considered for the private aggregation protocols presented in this section. They can be classified into three separate categories:

- FL training $\{N_{\mathsf{AggRounds}}, N_{\mathsf{ModelParam}}, L\}$: A set of parameters related with the federated learning training pipeline.
- RLWE cryptographic parameters $\{p, p', q, n, B_{\mathsf{Init}}\}$: A set of parameters related with the use of homomorphic encryption as the underlying tool for encrypted aggregation.

- Security and correctness parameters $\{\kappa, \lambda\}$: A set of parameters characterizing the security (λ) and the correctness (κ) of the private aggregation protocol.

Parameter	Description
$N_{\mathsf{ModelParam}}$	Number of parameters of the model updates to be aggregated
$N_{\mathsf{AggRounds}}$	Number of aggregation rounds required for FL training
$N_{\mathsf{Ctxts.PerRound}}$	Number of input ciphertext per round sent by a Data Owner to the Aggregator. $N_{\mathsf{Ctxts.PerRound}}$ is equal to $\lceil \frac{N_{\mathsf{ModelParam}}}{n} \rceil$
L	Number of Data Owners
p	Plaintext modulo. All parameters of the aggregated model update must be bounded by p to have correctness; i.e., $p/2 > \|\| \sum_i \lambda_i m_i \|\|_\infty$
p'	Modulo used for the partial decryptions
q	Ciphertext modulo
n	Number of coefficients in the polynomials of R_q. It also corresponds to the maximum number of parameters which can be encrypted in the same ciphertext
B_{Init}	Upper bound for the coefficients of the polynomials sampled from the error distribution; i.e., if $e \leftarrow \chi$, then $\|\|e\|\|_\infty < B_{\mathsf{Init}}$
B_{Agg}	Upper bound for the coefficients of the polynomial error in the encrypted aggregation. If the aggregation considers $\lambda_i = 1$ for all i, then $B_{\mathsf{Agg}} = LB_{\mathsf{Init}}$
κ	$2^{-\kappa}$ upper bounds the probability of having at least one decryption error during the execution of $N_{\mathsf{AggRounds}}$ aggregation rounds with L Data Owners and with $N_{\mathsf{Ctxts.PerRound}}$ ciphertexts per Data Owner in each round
Bit security or λ	Given the RLWE-based cryptosystem parameters $\{n, q, B_{\mathsf{init}}\}$, together with the error and secret distributions, we can follow the indications of [3] to estimate the bit security of the encryption scheme

Table 3. Parameters of the Multi-Key HE-based Protocols from Section 1.

Selection of parameters: The wide number of parameters to define the execution of the protocols makes it hard to select the most efficient and safe choice. In view of this, our objective here is to provide a methodology for the adequate selection of the protocol parameters depending on the needs of the FL training process. We propose the following:

1. Once the FL training task is defined, we first select adequate FL training parameters $\{N_{\mathsf{AggRounds}}, N_{\mathsf{ModelParam}}, L\}$.
2. If we have an upper bound $M > ||m_i||_\infty$ for all i, we choose $p/2 \geq L \cdot M$.
3. We select the κ and λ parameters. For example, we recommend choosing them as $\kappa = 128$ and $\lambda = 128$.
4. We select B_{Init}. For example, the Lattigo library typically uses $\sigma = 3.2$ for the Gaussian error distribution χ with a default bound $B_{\mathsf{Init}} = 6\sigma = 19.2$.[2]
5. To select q and n, we can start by choosing an initial value for n (for example $n = 2048$) and
   (a) make use of the bound for q introduced in [12]:
   $$q \geq 4 \cdot n^2 \cdot N_{\mathsf{AggRounds}} \cdot N_{\mathsf{Ctxts.PerRound}} \cdot p \cdot L^2 \cdot B_{\mathsf{Init}}^2 \cdot 2^\kappa.$$
   (b) Given $\{n, q, B_{\mathsf{Init}}\}$, we can follow the recommendations of [3] to obtain an estimate of the bit security of the encryption scheme.
      - If the bit security is equal to λ, we go to step 6.
      - If the bit security is smaller than λ, we double n and go to step 5. (a).
      - If the bit security is bigger than λ, we can divide n by 2 and repeat again step 5. (a) to check whether q and n can still be reduced a bit more. On the contrary, we go to step 6.
6. Choose $p' > 2nB_{\mathsf{Agg}}p$ and $N_{\mathsf{Ctxts.PerRound}} = \lceil \frac{N_{\mathsf{ModelParam}}}{n} \rceil$.

[2] See `https://github.com/tuneinsight/lattigo/blob/master/core/rlwe/security.go`.

Implementation design We have implemented for demonstration purposes (see Section 2 for several runtime evaluations and comparisons) the different primitives detailed in Section 1 inside the algorithmic descriptions of the two versions of the aggregation protocols. In particular, to deal with the polynomial rings considered in RLWE-based encryption, we use the programming language Go and the basic functionalities provided by the Lattigo library [9].

While it is true that Lattigo provides a wide set of high-level packages related to the execution of homomorphic encryption (e.g., he, schemes/bfv, schemes/bgv and schemes/ckks) together with the use of threshold variants (mainly mhe/mhefloat and mhe/mheint), many of those layers offer scheme-agnostic interfaces that are somewhat redundant and not really needed for our main purpose of implementing the multi-key aggregation protocols.

In contrast, here we are especially interested in adequately managing the polynomial arithmetic, sampling and rounding steps over the rings R_q, $R_{p'}$ and R_p. Taking this into account, we have made an extensive use of the low-level packages "ring" and "utils" from Lattigo v5.0.2.[3] We provide next a high-level description of the functionalities implemented, and discuss more in detail the operations used from the Lattigo packages in Section 1.3.

*Setup phase (*setupPhase *functionality):* In this phase we initialize and allocate in memory the protocol parameters and variables defined in Table 3. There are also several additional aspects that we must take into account for the implementation:

- $\{p, p', q\}$: The ciphertext modulo q considered in Lattigo must be composed of several prime factors q_i which fit within one machine limb for efficiency purposes (typically of size between 30 and 60 bits). This means that we define $q = \prod_{i=0}^{\mathsf{Level}-1} q_i$ and that q has a total of Level levels.
  If we now want to apply the rounding division from modulo q to a smaller modulo p', Lattigo allows us to do this rescaling by removing some of its levels. Consequently, p' must be composed of a subset of size Level′ of the prime factors of q; i.e., p' is defined as $p' = \prod_{i=0}^{\mathsf{Level}'-1} q_i$. Finally, we will typically choose p as q_0.
- Polynomial samplers: We must define and initialize both uniform and Gaussian random samplers which will be used in each phase of the protocol over the adequate modulo level of the rings R_q, $R_{p'}$ and R_p. For the use of the seeds during initialization, we must take into account that the generated "a" terms must be "correlated" among the different DOs in subsequent calls. It is also worth mentioning that Lattigo also offers cryptographically secure random samplers that we are currently exploring.
- Go routines: We define the number of threads to be used for parallelization during the execution of the protocol. We can specify a different number of threads for the different phases associated to the aggregator or data owners. We refer the reader to Section 1.3 for more details.

After initialization and memory allocation, the following steps are executed:

- genParties functionality: Generation and memory allocation for each Data Owner's individual secret key.
- genSetupShare functionality: Generate the correlated secret shares of zero associated to each Data Owner.
- genInputs functionality: Generate uniformly random plaintexts per Data Owner in R_p. The expected result for aggregation is computed in the clear to test the correctness of the result obtained as output of the private aggregation protocol.

*Encryption phase (*encPhase *functionality):* The current implementation follows the guidelines indicated in the version-2 of the protocol. Consequently, the functionality requires as:

[3] The github repository is available at `https://github.com/tuneinsight/lattigo`.

- Input: The corresponding model updates to be encrypted during one aggregation round.
- Secret key material and randomness: This includes some private information from the Data Owner, such as uniform and Gaussian samplers, individual secret key, and its corresponding share of zero.
- Output: An array of encryptions with their corresponding partial decryptions.

This implementation encrypts sequentially all input plaintexts, producing as output a total of $N_{\mathsf{Ctxts.PerRound}}$ ciphertexts in R_q, together with their corresponding partial decryptions in $R_{p'}$.

*Aggregation phase (*encEval *functionality):* The current implementation follows the guidelines indicated in the version-2 of the protocol. Consequently, the functionality requires as:

- Input: An array of $N_{\mathsf{Ctxts.PerRound}}$ ciphertexts and partial decryptions per Data Owner, with L Data Owners, resulting in a total of LN ciphertexts and LN partial decryptions.
- Output: The functionality returns the result of the aggregation as an array of $N_{\mathsf{Ctxts.PerRound}}$ masked polynomials in R_p.

This implementation computes sequentially the addition of the array of inputs ciphertexts and partial decryptions. As a result, an array of $N_{\mathsf{Ctxts.PerRound}}$ aggregated ciphertexts and partial decryptions is obtained. Next, the aggregated ciphertexts are divided and rounded from level $\mathsf{Level}-1$ to $\mathsf{Level}'-1$. Finally, they are combined with their corresponding partial decryptions, and subsequently divided and rounded from level $\mathsf{Level}'-1$ to level 0.

*Decryption phase (*decPhase *functionality):* The current implementation follows the guidelines indicated in the version-2 of the protocol. Consequently, the functionality requires as:

- Input: An array of $N_{\mathsf{Ctxts.PerRound}}$ masked polynomials in R_p.
- Secret randomness: This corresponds to the combined mask used to protect the result of the aggregation and which is known by all DOs
- Output: An array of $N_{\mathsf{Ctxts.PerRound}}$ polynomials in R_p corresponding to the aggregated update.

The implementation removes the mask from the polynomials in R_p and returns as a result the aggregated update.

Some clarifications regarding version-*1 and* version-*2:* While the implemented primitives are tailored to version-2, we want to remark that it is not hard to adapt them to the scenario described in the version-1 of the aggregation protocol. The changes required mainly imply to move the computation of the partial decryption to the decryption phase. As a consequence of this change, the aggregator reduces the computational cost, but a further communication round is now required among DOs to exchange their partial decryptions.

Dependencies on low-level packages of Lattigo As discussed, in our Go implementation we use the following packages from the Lattigo library:

- Package utils: First, we make use of several functions from this package for the correct initialization of different types of uniform and Gaussian samplers employed in the aggregation protocol (subpackage sampling). Second, we also use it to represent the big numbers associated to the encoding of q/p during encryption (subpackage bignum).
- Package ring: we make use of all the RNS (Residue Number System) and NTT (Number Theoretic Transform) functions facilitating modular arithmetic operations with polynomials in the RNS basis. In particular, polynomial elements are previously encoded in the RNS-NTT representation to accelerate multiplication. Also, we consider the Montgomery modular multiplication, which requires to keep track of whether the polynomials are in Montgomery form or not after several multiplication operations. Finally, we apply the division and rounding operations provided by Lattigo under the RNS representation.

Parallelization with Go routines Our implementation allows to control the number of lightweight threads used during different phases of the aggregation protocol. We can specify (1) the number of Go routines NumGoRoutinesParties used during encryption by the Data Owners, and a (2) different number of Go routines NumGoRoutinesCloud used by the aggregator.

This distinction will help to emulate in Section 2 the common scenario in which the aggregator has more computational resources than the Data Owners. Regarding how to distribute the work tasks among these Go routines, we choose different strategies depending on the executed phase. We elaborate more on this next.

*Encryption phase (*encPhaseParallel *functionality):* For this phase, we define an array of work tasks which has as many components as $N_{\mathsf{Ctxts.PerRound}}$. Then, the tasks assigned in each component are distributed among the number of available Go routines NumGoRoutinesParties.

The work assigned per task is the following:

- The Go routine is in charge of encrypting n local model parameters into one ciphertext and generating its corresponding partial decryption.

Once all the work tasks are finished, we can output the encrypted local model update.

*First part - Aggregation phase (*evalPhaseAggParallel *functionality):* For this first part, we define two different types of work tasks which are executed in sequential order:

- Tasks type (1). Each task corresponds to the aggregation of two arrays with length $N_{\mathsf{Ctxts.PerRound}}$, and each component holds one ciphertext. We define the following structure for a series of consecutive work tasks:
  1. We initialize an array layer with length $L' = L$ (number of DO). Each position of the array holds the encrypted input of a different DO.
  2. The first set of launched tasks with identifiers $k = 0, \ldots, \lfloor (L'-1)/2 \rfloor$ is defined as:
     - The Go routine with input identifier k must add component-wise the arrays of ciphertexts in positions $2k$ and $2k+1$. The result is stored in position $2k$.
  3. Next, we remove all the components of the array layer except those in positions:
     - $2k$ from $k = 0 \ldots \lfloor (L'-1)/2 \rfloor$ if L' was even.
     - $2k$ from $k = 0 \ldots \lfloor (L'-1)/2 \rfloor + 1$ if L' was odd.
  4. L' is updated with the new reduced size of layer. Now, steps 2, 3 and 4 are repeated till L' is equal to 1. Once this is achieved, layer stores the result of aggregating all the DOs' inputs.
- Tasks type (2). Starting from the result obtained ($N_{\mathsf{Ctxts.PerRound}}$ ciphertexts over R_q) from the execution of all Tasks type (1), we define an array of work tasks in which each task is associated to a different ciphertext. Then, the tasks are distributed among the number of available Go routines NumGoRoutinesCloud.

  The work assigned per task is the following:
  - The Go routine is in charge of dividing and rounding the input ciphertext from R_q into $R_{p'}$.

  As a result, we obtain an array of $N_{\mathsf{Ctxts.PerRound}}$ ciphertexts over $R_{p'}$.

*Second part - Aggregation phase (*evalPhaseDecParallel *functionality):* For this second part, we also define two different types of work tasks which are executed in sequential order:

- Tasks type (1). Each task corresponds to the aggregation of two arrays with length $N_{\mathsf{Ctxts.PerRound}}$, and each component holds one partial decryption. We define the following structure for a series of consecutive work tasks:
  1. We initialize an array layer with length $L' = L$ (number of DO). Each position of the array holds the partial decryptions of a different DO.
  2. The first set of launched tasks with identifiers $k = 0, \ldots, \lfloor (L'-1)/2 \rfloor$ is defined as:

- The Go routine with input identifier k must add component-wise the arrays of partial decryptions in positions $2k$ and $2k+1$. The result is stored in position $2k$.

3. Next, we remove all the components of the array layer except those in positions:
   - $2k$ from $k = 0 \dots \lfloor (L'-1)/2 \rfloor$ if L' was even.
   - $2k$ from $k = 0 \dots \lfloor (L'-1)/2 \rfloor + 1$ if L' was odd.
4. L' is updated with the new reduced size of layer. Now, steps 2, 3 and 4 are repeated till L' is equal to 1. Once this is achieved, layer stores the result of aggregating all the DOs' partial decryptions.

– Tasks type (2). Starting from the result obtained ($N_{\mathsf{Ctxts.PerRound}}$ partial decryptions over $R_{p'}$) from the execution of all Tasks type (1), we define an array of work tasks in which each task is associated to a different partial decryption. Then, the tasks are distributed among the number of available Go routines NumGoRoutinesCloud.

The work assigned per task is the following:

- The Go routine is in charge of combining the partial decryption with its corresponding encryption. Then, the result is divided and round from $R_{p'}$ into R_p.

As a result, we obtain an array of $N_{\mathsf{Ctxts.PerRound}}$ masked polynomials over R_p.

2 Performance Evaluation of Private Aggregation based on Multi-Key HE

In this section we present the runtime performance of the Multi-Key HE-based aggregation protocols outlined in Section 1. To ensure a fair comparison, we compare the performance of this scheme with a solution solely based on the use of threshold BFV. We begin by discussing how to select suitable and secure parameters for the BFV scheme. Subsequently, we provide several runtime comparisons for a broad range of practical parameter sets.

Comparison with Baseline threshold BFV To have a fair first comparison between our proposed protocols in Section 1 and a solution entirely based on threshold HE, we must also choose appropriate parameters for the latter. With this aim, we focus next on studying how to adequately select the cryptosystem parameters for a threshold variant of BFV.

Most of the RLWE parameters which were indicated in Table 1 still apply, but with some minor differences:

– FL training $\{L\}$: The parameters $N_{\mathsf{AggRounds}}$ and $N_{\mathsf{ModelParam}}$ can be omitted for our analysis, as the Lattigo library trims the Gaussian noise when a certain upper bound is exceeded. In contrast to the scheme from Section 1, this upper bound allows for the selection of parameters for threshold BFV which ensure perfect decryption.

– RLWE cryptographic parameters $\{p, q, n, B_{\mathsf{Init}}\}$: Similarly to Section 1, this set of parameters is associated with the use of homomorphic encryption for the computation of the encrypted aggregation. Note that now we do not need an extra modulo p', as BFV only considers a ciphertext modulo q and a plaintext modulo p.

– Security parameter $\{\lambda\}$: A parameter characterizing the minimum security (bit security $\geq \lambda$) of the private aggregation protocol. In contrast to Section 1, where λ only represents the bit security associated to the indistinguishability of RLWE instances, here λ will also characterize how big is the variance of the smudging noise applied during the distributed protocol for decryption.

Selection of BFV parameters: We briefly outline the process for selecting the protocol parameters with threshold BFV.

1. After defining the FL training task, we initially select adequate FL training parameters $\{N_{\mathsf{AggRounds}}, N_{\mathsf{ModelParam}}, L\}$. Remember that only the last one will have an effect in the choice of q.

2. If we have an upper bound $M > ||m_i||_\infty$ for all i, we choose $p/2 \geq L \cdot M$.
3. We select the security λ parameter. For example, we recommend choosing $\lambda = 128$.
4. We select the upper bound B_{Init} considered in the Lattigo library. By default the library uses $\sigma = 3.2$ for the error distribution χ, and an upper bound $B_{\mathsf{Init}} = 6\sigma = 19.2$.
5. To select values for q and n, we can begin by choosing an initial value for n (for example $n = 2048$) and
   (a) make use of the following lower bound for q derived from [11, Eqs. 8 and 9 from Appendix A]:
   $$q \geq 2 \cdot p \cdot B_{\mathsf{Init}} \cdot 2^{\frac{\lambda}{2}} \cdot (2nL^3 + L^2) + 2 \cdot p \cdot B_{\mathsf{Init}} \cdot (2nL^2 + L).$$
   (b) Given $\{n, q, B_{\mathsf{Init}}\}$, we can follow the recommendations of [3] to obtain an estimate of the bit security of the encryption scheme.
      - If the bit security is equal to λ, we finish the selection of parameters.
      - If the bit security is smaller than λ, we double n and go to step 5. (a).
      - If the bit security is bigger than λ, we can divide n by 2 and repeat again step 5.(a) as a means to check whether q and n can be further reduced. On the contrary, we finish the selection of parameters.

How to obtain the lower bound for q in BFV: According to [11, Eq. 8 and 9], we can upper bound the error of a fresh encryption with L participants as:

$$||e_{\mathsf{fresh}}||_\infty \leq B_{\mathsf{Init}}(2nL + 1).$$

The smudging noise applied during the distributed protocol for decryption is bounded as:

$$||e_{\mathsf{smg}}||_\infty \leq B_{\mathsf{smg}}L.$$

Then, from the previous two upper bounds, we can easily see that the expression

$$||e_{\mathsf{final}}||_\infty \leq L \cdot ||e_{\mathsf{fresh}}||_\infty + ||e_{\mathsf{smg}}||_\infty$$

corresponds to the upper bound for the final error present in the decrypted result.

To ensure security (see Section 1 for further details), we adhere to the guidelines provided in [6], requiring the smudging noise to satisfy $\sigma^2_{\mathsf{smg}} = 2^\lambda L \sigma^2_{\mathsf{fresh}}$, which results in $B_{\mathsf{smg}} = 2^{\frac{\lambda}{2}} L B_{\mathsf{Init}}$. Combining all the previous expressions, we obtain:

$$||e_{\mathsf{final}}||_\infty \leq B_{\mathsf{Init}} \cdot 2^{\frac{\lambda}{2}} \cdot (2nL^3 + L^2) + B_{\mathsf{Init}} \cdot (2nL^2 + L),$$

from which the lower bound of q in threshold BFV can be finally derived considering that, for correct decryption, $q/(2p) \geq ||e_{\mathsf{final}}||_\infty$.

Communication cost: We include in Table 4 the communication cost per party for the private aggregation protocol based on the use of threshold BFV. We assume that the number of model parameters $N_{\mathsf{ModelParam}}$ is high enough so that ciphertexts are fully packed. Note that the decryption share of each DO must be sent to the other $L - 1$ DOs. At worst-case, if each DO uses $L - 1$ party-to-party channels to transmit the decryption share, the communication cost per party during the phase of distributed decryption is $(L - 1)N_{\mathsf{ModelParam}} \cdot \log_2 q$.

Runtime comparisons with threshold BFV The implementation runtimes included in this section were conducted both single-threaded and multi-threaded on an Intel Core i7-10750H CPU @ 2.60GHz × 12 with 31.1 GB. For the performance comparison, we considered two aggregation protocols based on different HE tools: (1) threshold BFV, and (b) a tailored multi-key HE scheme.

Input per DO	Decryption share per DO	Aggregator output	Decrypted result
$2 \cdot N_{\mathsf{ModelParam}} \cdot \log_2 q$	$N_{\mathsf{ModelParam}} \cdot \log_2 q$	$2 \cdot N_{\mathsf{ModelParam}} \cdot \log_2 q$	$N_{\mathsf{ModelParam}} \cdot \log_2 p$

Table 4. Communication costs per party in each round of the private aggregation protocol based on the use of threshold BFV from Section 2.

Parameter	Paramater set 1	Paramater set 2	Parameter set 3
Secret key distribution	ternary	ternary	ternary
$N_{\mathsf{AggRounds}}$	16	16	16
$N_{\mathsf{ModelParam}}$	1048576	1048576	1048576
$N_{\mathsf{Ctxts.PerRound}}$	128	128	128
$\{n, L\}$	$\{8192, 16\}$	$\{8192, 16\}$	$\{8192, 16\}$
$\{p, q\}$	$\{22, 132\}$ bits	$\{30, 150\}$ bits	$\{60, 180\}$ bits
B_{Init}	19.2	19.2	19.2
$\{\mathsf{bit\ security} \geq \lambda, \lambda\}$	$\{192, 128\}$	$\{192, 128\}$	$\{128, 128\}$

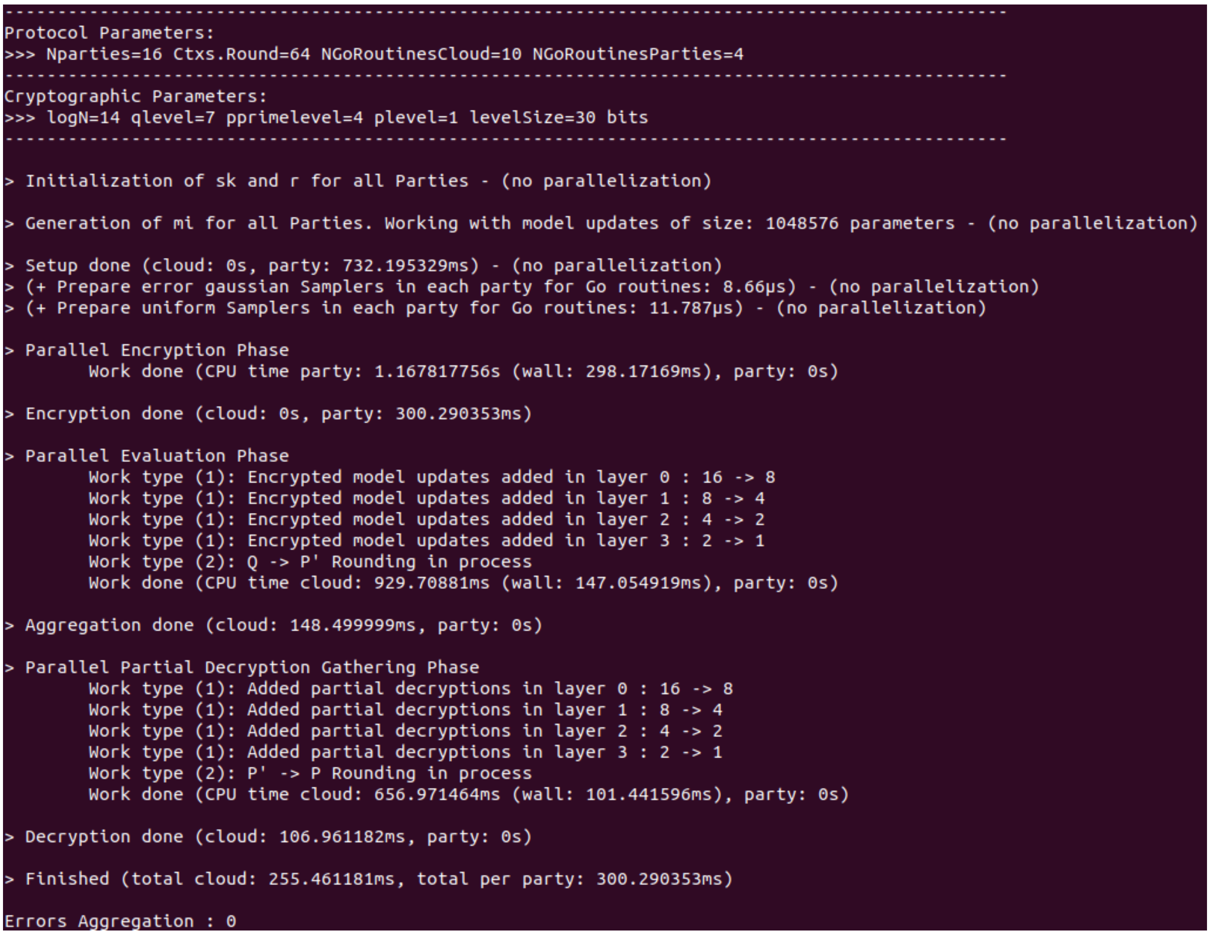
Table 5. Example parameter sets for the private aggregation protocol based on the use of threshold BFV from Section 2).

Parameter	Paramater set 1	Paramater set 2	Parameter set 3
Secret key distribution	Gaussian	Gaussian	Gaussian
$N_{\mathsf{AggRounds}}$	16	16	16
$N_{\mathsf{ModelParam}}$	1048576	1048576	1048576
$N_{\mathsf{Ctxts.PerRound}}$	128	128	64
$\{n, L\}$	$\{8192, 16\}$	$\{8192, 16\}$	$\{16384, 16\}$
$\{p, p', q\}$	$\{22, 44, 198\}$ bits	$\{30, 60, 210\}$ bits	$\{60, 90, 240\}$ bits
$\{B_{\mathsf{Init}}, B_{\mathsf{Agg}}\}$	$\{19.2, 307.2\}$	$\{19.2, 307.2\}$	$\{19.2, 307.2\}$
$\{\mathsf{bit\ security} = \lambda, \kappa\}$	$\{128, 120\}$	$\{128, 124\}$	$\{192, 123\}$

Table 6. Example parameter sets for the private aggregation protocol from Section 1.3 based on the use of multi-key HE).

We also include in Tables 5 and 6 three different sets of parameters for each of the two solutions. The selection process for the choice of parameters followed the indications discussed in Sections 2 and 1.

As an example, Figure 2 showcases the execution of our implementation for the protocol described in Section 1 with the "Parameter set 3" from Table 6, $\mathsf{NumGoRoutinesParties} = 4$ and $\mathsf{NumGoRoutinesCloud} = 10$.



```
-----------------------------------------------------------------------------------------------
Protocol Parameters:
>>> Nparties=16 Ctxs.Round=64 NGoRoutinesCloud=10 NGoRoutinesParties=4
-----------------------------------------------------------------------------------------------
Cryptographic Parameters:
>>> logN=14 qlevel=7 pprimelevel=4 plevel=1 levelSize=30 bits
-----------------------------------------------------------------------------------------------

> Initialization of sk and r for all Parties - (no parallelization)

> Generation of mi for all Parties. Working with model updates of size: 1048576 parameters - (no parallelization)

> Setup done (cloud: 0s, party: 732.195329ms) - (no parallelization)
> (+ Prepare error gaussian Samplers in each party for Go routines: 8.66µs) - (no parallelization)
> (+ Prepare uniform Samplers in each party for Go routines: 11.787µs) - (no parallelization)

> Parallel Encryption Phase
        Work done (CPU time party: 1.167817756s (wall: 298.17169ms), party: 0s)

> Encryption done (cloud: 0s, party: 300.290353ms)

> Parallel Evaluation Phase
        Work type (1): Encrypted model updates added in layer 0 : 16 -> 8
        Work type (1): Encrypted model updates added in layer 1 : 8 -> 4
        Work type (1): Encrypted model updates added in layer 2 : 4 -> 2
        Work type (1): Encrypted model updates added in layer 3 : 2 -> 1
        Work type (2): Q -> P' Rounding in process
        Work done (CPU time cloud: 929.70881ms (wall: 147.054919ms), party: 0s)

> Aggregation done (cloud: 148.499999ms, party: 0s)

> Parallel Partial Decryption Gathering Phase
        Work type (1): Added partial decryptions in layer 0 : 16 -> 8
        Work type (1): Added partial decryptions in layer 1 : 8 -> 4
        Work type (1): Added partial decryptions in layer 2 : 4 -> 2
        Work type (1): Added partial decryptions in layer 3 : 2 -> 1
        Work type (2): P' -> P Rounding in process
        Work done (CPU time cloud: 656.971464ms (wall: 101.441596ms), party: 0s)

> Decryption done (cloud: 106.961182ms, party: 0s)

> Finished (total cloud: 255.461181ms, total per party: 300.290353ms)

Errors Aggregation : 0
```

Fig. 2. Example execution of our implementation of the private aggregation protocol from Section 1.

Finally, Tables 7 and 8 report our runtime results for the two private aggregation protocols:

- Table 7 presents implementation runtimes for the baseline aggregation protocol based on threshold BFV, with $\mathsf{NumGoRoutinesCloud}$ set to $\{1, 4, 12\}$. We include the runtimes corresponding to the Collective Key Generation phase (CKG phase) described in [11, Protocol 1]. Note that for the collaborative decryption we make use of the public collective key-switching protocol (PCKS phase) described in [11, Protocol 4]. This procedure allows Data Owners save computation and communication costs by relying on the aggregator (i.e., cloud) to receive and combine all the partial decryptions. Instead of recovering the aggregation in the clear, the result of the PCKS phase is again an encrypted aggregation, but now only under an individual secret key known by all the Data Owners. It is also worth mentioning that, while the required q for the PCKS phase would be slightly bigger than the one discussed in Section 2, its effect in practice is negligible considering the smudging noise with $\lambda = 128$. We refer the reader to [11, Eq.(10)] for more details on the extra noise term introduced by the PCKS phase.

- Table 8 presents implementation runtimes for the protocol based on tailored Multi-Key HE, with NumGoRoutinesParties set to $\{1, 4, 12\}$ and NumGoRoutinesCloud equal to $\{1, 4, 12\}$. We do not include the runtime of the final unmasking done by the DOs as it is negligible comparing to the total runtime associated to the other phases (a total of $N_{\mathsf{ModelParam}}$ elemental subtractions modulo p in the clear domain).

Protocol phase	Parameter set 1	Parameter set 2	Parameter set 3
CKG phase (1 thread, Agg)	Agg: 673 μs	Agg: 698 μs	Agg: 885 μs
CKG phase (1 thread, DO)	DO: 1.17 ms	DO: 1.29 ms	DO: 1.45 ms
Encrypt phase (1 thread)	DO: 661 ms	DO: 767 ms	DO: 896 ms
Aggregation phase (1 thread)	Agg: 149 ms	Agg: 184 ms	Agg: 222 ms
Aggregation phase (4 threads)	Agg: 104 ms	Agg: 127 ms	Agg: 151 ms
Aggregation phase (12 threads)	Agg: 102 ms	Agg: 125 ms	Agg: 151 ms
PCKS phase (1 thread, Agg)	Agg: 128 ms	Agg: 159 ms	Agg: 188 ms
PCKS phase (1 thread, DO)	DO: 1365 ms	DO: 1527 ms	DO: 1708 ms
Decrypt phase (1 thread)	DO: 17.9 ms	DO: 24.9 ms	DO: 30 ms
Total runtime (1 thread)	2323 ms	2664 ms	3046 ms

Table 7. Implementation runtimes for the private aggregation protocol based on threshold BFV from Section 2 (NumGoRoutinesParties $= 1$ and NumGoRoutinesCloud $= \{1, 4, 12\}$).

Protocol phase	Parameter set 1	Parameter set 2	Parameter set 3
Encryption phase (1 thread)	DO: 1046 ms	DO: 787 ms	DO: 1017 ms
Encryption phase (4 threads)	DO: 306 ms	DO: 234 ms	DO: 295 ms
Encryption phase (12 threads)	DO: 181 ms	DO: 139 ms	DO: 184 ms
Eval phase - Part 1 Agg. (1 thread)	Agg: 412 ms	Agg: 313 ms	Agg: 404 ms
Eval phase - Part 1 Agg. (4 threads)	Agg: 199 ms	Agg: 145 ms	Agg: 162 ms
Eval phase - Part 1 Agg. (12 threads)	Agg: 153 ms	Agg: 125 ms	Agg: 144 ms
Eval phase - Part 2 Dec. (1 thread)	Agg: 145.22 ms	Agg: 144 ms	Agg: 268 ms
Eval phase - Part 2 Dec. (4 threads)	Agg: 65 ms	Agg: 62 ms	Agg: 114 ms
Eval phase - Part 2 Dec. (12 threads)	Agg: 53 ms	Agg: 55 ms	Agg: 97 ms
Total runtime (1 thread)	1604 ms	1244 ms	1689 ms

Table 8. Implementation runtimes for the private aggregation protocol based on tailored Multi-Key HE from Section 1 (NumGoRoutinesParties $= \{1, 4, 12\}$ and NumGoRoutinesCloud $= \{1, 4, 12\}$).

Precomputation of partial decryptions: The runtimes presented in Table 8 consider the cost associated with the computation of the "a" polynomial terms and their corresponding partial decryptions. It is noteworthy that, unlike the protocol relying on threshold BFV, in the multi-key HE-based protocol, the partial decryptions are independent of the encrypted result obtained by the Aggregator. This property allows improving efficiency by running an "offline precomputation phase" before initiating the online execution of the aggregation protocol. This is a line of work we would like to explore in the future.

Acknowledgements

A. Pedrouzo-Ulloa is partially supported by the European Union's Horizon Europe Framework Programme for Research and Innovation Action under project TRUMPET (proj. no. 101070038), by the European Regional Development Fund (FEDER) and Xunta de Galicia under project "Grupos de Referencia Competitiva" (ED431C 2021/47), and by FEDER and MCIN/AEI under project FELDSPAR (TED2021-130624B-C21). Part of the work has been completed as a visiting researcher at CEA-List, Université Paris-Saclay, funded by the European Union "NextGenerationEU/PRTR" under TRUFFLES and by means of a Margarita Salas grant of the Universidade de Vigo.

The author would like to thank Fernando Pérez-González and Miguel Morona-Mínguez for helpful feedback and thorough reading of a preliminary version of this technical report.

References

1. A. Aloufi, P. Hu, Y. Song, and K. E. Lauter. Computing blindfolded on data homomorphically encrypted under multiple keys: A survey. *ACM Comput. Surv.*, 54(9):195:1–195:37, 2022.
2. C. Baum, D. Escudero, A. Pedrouzo-Ulloa, P. Scholl, and J. R. Troncoso-Pastoriza. Efficient protocols for oblivious linear function evaluation from ring-lwe. *J. Comput. Secur.*, 30(1):39–78, 2022.
3. J. Bossuat, R. Cammarota, J. H. Cheon, I. Chillotti, B. R. Curtis, W. Dai, H. Gong, E. Hales, D. Kim, B. Kumara, C. Lee, X. Lu, C. Maple, A. Pedrouzo-Ulloa, R. Player, L. A. R. Lopez, Y. Song, D. Yhee, and B. Yildiz. Security guidelines for implementing homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 463, 2024.
4. Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology - CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.
5. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.
6. M. Checri, R. Sirdey, A. Boudguiga, J. Bultel, and A. Choffrut. On the practical CPAD security of "exact" and threshold FHE schemes and libraries. *IACR Cryptol. ePrint Arch.*, page 116, 2024.
7. J. H. Cheon, H. Choe, A. Passelègue, D. Stehlé, and E. Suvanto. Attacks against the INDCPA-D security of exact FHE schemes. *IACR Cryptol. ePrint Arch.*, page 127, 2024.
8. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology - ASIACRYPT*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, 2016.
9. C. M. et al. Lattigo: a multiparty homomorphic encryption library in go. In *WAHC 2020*, 2020.
10. J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.
11. C. Mouchet, J. R. Troncoso-Pastoriza, J. Bossuat, and J. Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proc. Priv. Enhancing Technol.*, 2021(4):291–311, 2021.
12. A. Pedrouzo-Ulloa, A. Boudguiga, O. Chakraborty, R. Sirdey, O. Stan, and M. Zuber. Practical multi-key homomorphic encryption for more flexible and efficient secure federated average aggregation. In *IEEE International Conference on Cyber Security and Resilience, CSR 2023, Venice, Italy, July 31 - Aug. 2, 2023*, pages 612–617. IEEE, 2023.
13. A. G. Sébert, M. Checri, O. Stan, R. Sirdey, and C. Gouy-Pailler. Combining homomorphic encryption and differential privacy in federated learning. In *20th Annual International Conference on Privacy, Security and Trust, PST 2023, Copenhagen, Denmark, August 21-23, 2023*, pages 1–7. IEEE, 2023.