

---

# MICROFEDML: EFFICIENT PRIVACY-PRESERVING FEDERATED LEARNING

---

June 27, 2023

## ABSTRACT

Secure aggregation on users' private data with the aid of a server provides strong privacy guarantees and has been well-studied in the context of privacy-preserving federated learning. An important problem in privacy-preserving federated learning with user-constrained computation and wireless network resources is the computation and communication overhead which wastes bandwidth, increases training time, and can even impact the model accuracy if many users drop out. The seminal work of Bonawitz et al. (CCS 2017) and the work of Bell et al. (CCS 2020) have constructed secure aggregation protocols for a very large number of users which handle dropout users in a federated learning setting. However, these works suffer from high round complexity (i.e., the number of times the users exchange messages with the server) and overhead in every training iteration. In this work, we propose MicroFedML, a new secure aggregation system with lower round complexity and computation overhead per training iteration. In particular, the round complexity is reduced to 3 rounds and the computation complexity of each iteration for each user increases linearly with the number of users in MicroFedML while that of Bonawitz et al. and Bell et al. both grow quadratically. Recently, Ma et al. (S&P 2023) proposed Flamingo, a 3-round protocol aided by a committee of decryptors, which achieves better server computation time than our protocol for models with a large number of weights/parameters. However, MicroFedML achieves better bandwidth and client computation time than Flamingo.

We report an implementation of our system and compare the performance against prior works, demonstrating that MicroFedML reduces the computational burden and the message size by multiple orders of magnitude.

## 1 Introduction

Federated learning allows a large number of users with limited resources, e.g., mobile phones, to collaboratively train a global learning model with the assistance of a central server without sharing the raw data with any other party. In each learning iteration of federated learning, a central server sends the global model to all users who then trains the model with their local data to obtain an updated model. The server aggregates the updates from all the users and updates the global model. The new global model is sent to all users, and the process repeats. In many cases, individual user privacy can still be compromised by using the trained model and the local updates to infer certain details of the training data set Nasr et al. (2019). To address this problem, the seminal work of Bonawitz et al. proposed the first secure aggregation protocol run among the server and the users without revealing any individual update. The protocol allows each user to modify its local updates by masking/encrypting them such that no information about the local updates is revealed to the server and the other users apart from the final aggregated update.

In the federated learning setting, we require secure aggregation protocols whose efficiency and communication requirements scale practically even when the number of parties is large. Bonawitz et al. (2017) proposed the first secure aggregation protocol which is efficient for deep-network-sized problems and real-world connectivity constraints. The protocol also allows users to drop offline and come back online later, as in many scenarios the users are not expected to be always online in the whole training process. The subsequent work of Bell et al. (2020) reduces the bandwidth cost at the price of higher round complexity and probabilistic correctness, i.e., when there are enough honest users online, the server learns the correct aggregated result with overwhelming probability. The work of

Bonawitz et al. (2017) provides a protocol with perfect correctness. Moreover, instead of requiring each user to exchange information with all other users in the network, Bell et al. (2020) works with sparse neighborhood graphs. The intuition is that each user has a small group of users as its neighbors and the communication only happens between the user and its neighbors.

The works of Bonawitz et al. (2017); Bell et al. (2020) suffer from some inefficiencies. More specifically, in every learning iteration the protocols in Bonawitz et al. (2017); Bell et al. (2020) reveal part of the masks used to protect the local updates from the server. That said, the masks cannot be reused in later training iterations, and fresh masks are generated at each iteration introducing more communication. Furthermore, every user needs to exchange information about the freshly generated masks with a number of other users (either all other users in Bonawitz et al. (2017) or a subset of users in Bell et al. (2020)) in every iteration, resulting in the communication cost growing significantly as the total number of users increases. Regarding the round complexity, the number of times messages are exchanged between the users and the server, it was reported to be 5 in Bonawitz et al. (2017) and 6 in Bell et al. (2020). A lower round complexity can prevent the multiple exchanges of messages between the server and the many users and can increase the accuracy given that users have fewer chances to drop out in every round. Therefore, a natural yet fundamental question to ask is the following: can we have a secure aggregation protocol which is more tailored to the multi-iteration nature of the federated learning setting with improvements in the round complexity, computational complexity, and bandwidth costs?

In this work, we answer the above question in the affirmative by designing secure aggregation protocols MicroFedML which considerably reduce the communication and computation overheads of Bonawitz et al. (2017); Bell et al. (2020); Ma et al. (2023). We are able to reduce the round complexity and reuse information used to mask the model updates. In a nutshell, our protocols generate the masks in a one-time setup phase and most importantly the masks can be reused in every training iteration while achieving the same provable security guarantees provided by Bonawitz et al. (2017); Bell et al. (2020). Notable improvements of MicroFedML over Bonawitz et al. (2017); Bell et al. (2020) include the following. MicroFedML reduces the round complexity from 5 in Bonawitz et al. (2017) and 6 in Bell et al. (2020) to just 3 rounds. The computation complexity of each iteration for each user increases linearly with the number of users in MicroFedML while that of Bonawitz et al. (2017); Bell et al. (2020) both grow quadratically. The recent work of Ma et al. (2023) achieves 3 rounds of communication too by leveraging not only neighborhoods but also a set of decryptors, who help reuse the masking information with a different approach than ours. However, our work does not require an additional set of decryptors to stay online, and it outperforms Flamingo in both user computation time and bandwidth utilization. On the other hand, our work has increased server computation time over Flamingo when we increase the size of the model weights, referred to as the vector length of the aggregation in the rest of the paper. Additionally, our server computation time is highly parallelizable.

## 1.1 Problem Statement and Threat Model

We consider a star topology network in which each user only communicates with the central server. We propose multi-iteration secure aggregation protocols in which  $n$  users  $P_i$  for  $i \in [n]$  hold a private value  $x_i$ , and they wish to learn the sum  $\sum_i x_i$  with the aid of a single untrusted server without leaking any information about the individual  $x_i$ . In the federated learning setting, the server and the users interact several times (multi-iteration) to compute the summation of model weights  $x_i$ . We model an adversary which can launch two kinds of attacks: (1) honest users that disconnect/drop out or are too slow to respond as a result of unstable network conditions, power loss, etc. Users can dynamically drop out and come back in a later iteration; and (2) arbitrary actions by an adversary that controls the server and a bounded fraction of the users.

Overall, we assume that the malicious adversary controls the server and at most  $\gamma$  fraction of users which it decides to corrupt before each protocol execution, and that at most  $\delta$  fraction of users are offline in every iteration. We assume that  $\gamma + 2\delta < 1$ . Like Flamingo, we assume a static adversary since the neighborhoods per federated learning iteration are fixed in the setup phase.

## 1.2 Our Results

We propose two new multi-iteration secure aggregation protocols, MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub>, both in the semi-honest and malicious adversary settings. Both protocols consist of two phases, the Setup phase and the Aggregation phase. The Setup phase which is independent of the user’s private inputs and runs only once at the beginning of the execution of federated learning consists of 3 rounds of interaction between the server and the users in MicroFedML<sub>1</sub> or 5 rounds in MicroFedML<sub>2</sub>. The aggregation phase runs repetitively for multiple learning iterations and consists of 2 rounds of interaction in the semi-honest scenario in both protocols, while an extra round is needed to protect privacy against a malicious adversary.

In Table 1 we list the communication complexity per user per round. As we can see our protocols offer a significant advantage per round. The one-time setup phase communication complexity is  $O(n)$  and  $O(\log n)$  for each user in MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub> respectively, which can be found in Table 2 in Section 7. We refer the reader to Section 7.1 and 7.2 for the detailed analysis of the asymptotic performance. The group version of our protocol is more suitable for larger number of parties and for use cases where weaker security guarantees are sufficient. Such as, the adversary cannot adaptively corrupt all parties in a single group neighborhood. If such an event happens privacy is lost. The group version in Bell et al. (2020) also cannot support adaptive corruptions and has a weaker security definition: for some parameter  $\alpha$  between  $[0, 1]$ , honest inputs are guaranteed to be aggregated at most once with at least  $\alpha$  fraction of other inputs from honest users. We summarize our results in the following two (informal) theorems:

**Theorem 1.1.** *The aggregation protocol MicroFedML<sub>1</sub> running with a server  $S$  and  $n$  users guarantees privacy in the presence of a semi-honest (malicious) adversary who can corrupt less than  $\frac{1}{2}$  ( $\frac{1}{3}$ ) fraction of the users and correctness for an adversary who can drop out less than  $\frac{1}{2}$  ( $\frac{1}{3}$ ) fraction of the users.*

**Theorem 1.2** (Group version). *Let  $\gamma, \delta$  be two parameters such that  $\gamma + 2\delta < 1$ . The aggregation protocol MicroFedML<sub>2</sub> running with a server  $S$  and  $n$  users guarantees privacy in the presence of a semi-honest (malicious) adversary who can corrupt less than  $\gamma = \frac{1}{2}$  ( $\gamma = \frac{1}{3}$ ) fraction of the users and correctness for an adversary who can drop out less than  $\delta$  fraction of the users.*

Correctness means that when parties follow the protocol the server gets a sum of online users at the end of each learning iteration even if dropouts happen during the computation of the summation. Privacy refers to the fact that an adversary (who may deviate from the protocol) cannot learn any individual user  $i$ 's input  $x_{i,k}$  for any training iteration  $k$ . Our protocols do not introduce any noise and thus do not affect the accuracy of the models. We test our protocol by running logarithmic regression algorithm on the Census adult dataset Dua and Graff (2017) and compare the learning results with plain federated learning in which the users send the plain text of the model update to the server. The two experiments provide models with the same accuracy (0.81).

We provide the asymptotic and concrete performance analysis in Section 7. We also offer two flavors of our constructions: (1) An algorithm based on general cyclic groups which work best with small input domains and is applicable in gradient sparsification Lin et al. (2020); Amiri and Gunduz (2020a,b); Sattler et al. (2018); Malekijoo et al. (2021); Aji and Heafield (2017); Alistarh et al. (2017); Chen et al. (2018), quantization and weight regularization areas Van Laarhoven (2017); Krogh and Hertz (1991); Yuen et al. (2021); Deng et al. (2021); Castagnos et al. (2019) in federated learning. This algorithm can also have applications to voting and any other application with small input domain. (2) An algorithm for any input domain based on class groups of unknown order Castagnos and Laguillaumie (2015) in Section 6. In recent years, class groups have been gaining renewed interest due to their usefulness in designing encryption schemes to multiparty computation protocols, as exemplified by Braun et al. (2022); Castagnos et al. (2018); Yuen et al. (2021); Deng et al. (2021); Castagnos et al. (2019); Boneh et al. (2019); Abram et al. (2022); Kate et al. (2023). For a more comprehensive understanding of the class group assumptions and more constructions, refer to the BICYCL work Bouvier et al. (2022).

**Implementation and evaluation.** We implemented the proposed system and report performance in Section 7. Our protocol MicroFedML<sub>1</sub> outperforms BIK+17 by 100 times in computation time with 500 total participants, while MicroFedML<sub>2</sub> runs more than 5 times faster than BBG+20 when the connectivity of the user communication graph is 100 and the total number of users is 500. For 1000 participants, MicroFedML<sub>2</sub> is 5 times faster than MicroFedML<sub>1</sub>. With class group, the computation time of our protocol can be further reduced. With 1024 participants, neighborhood size 64, and input of 32 bit length, MicroFedML<sub>2</sub> with class group is 10 times faster than Flamingo on both user and server side.

### 1.3 Related Works

Our work is inspired by the line of works on secure aggregation protocols Bonawitz et al. (2017); Bell et al. (2020). There are several other works Liu et al. (2020); Yang et al. (2021b); Fereidooni et al. (2021); Bell et al. (2022); Liu et al. (2022b) exploring the secure aggregation problem. Another line of works Truex et al. (2019); Xu et al. (2019) adopt differential privacy which is a generic privacy protection technique in database and machine learning areas. However, all of them either only consider semi-honest adversaries or do not allow offline users to come back online again.

Our protocol can also be combined with other secure aggregation solutions. A recent work by Liu et al. (2022a) uses a Learning With Rounding (LWR)-based homomorphic PRG to improve efficiency by removing the need for one layer of symmetric masks while introducing noise growing in proportion to the number of users. Their solution uses existing secure aggregation protocols as a black box to aggregate the seeds of PRG from users, which is an application suitable for MicroFedML as MicroFedML is more efficient than the protocol Bell et al. (2020); Bonawitz et al. (2017).

Round	Communication cost per user				
	BIK+17	BBG+20	Flamingo	MicroFedML <sub>1</sub>	MicroFedML <sub>2</sub>
1	$O(n)$ elements	<u><math>O(1)</math> elements</u>	1 element	1 element	1 element
2	$O(n)$ elements	$O(\log n)$ elements	<u><math>O(\log n)</math> elements</u>	1 element + $O(n)$ bits	1 element + $O(\log n)$ bits
3	1 elements	$O(\log n)$ elements	$O(\log n)$ elements	<u><math>O(n)</math> elements</u>	<u><math>O(\log n)</math> elements</u>
4	<u><math>O(n)</math> elements</u>	1 element			
5	$O(n)$ elements	<u><math>O(\log n)</math> elements</u>			
6		<u><math>O(\log n)</math> elements</u>			

Table 1: Communication overhead per user of each training iteration of our protocols guaranteeing privacy against semi-honest/malicious adversaries (the extra round required for privacy in the malicious setting is marked as blue and underlined in the table).  $n$  denotes the total number of users and  $R$  denotes the size of the range of the aggregation output. An element in BIK+17 and BBG+20 is of size  $O(\log R)$  while an element in MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub> is of size  $O(R)$ . The overhead includes both received and sent messages.

used in the paper. Combining with a homomorphic PRG also reduces the computation cost of our protocol for use cases with long vector inputs. The differential privacy in hybrid approaches can also be applied to our protocol.

## 2 Our Approach

In this section we explain the high level idea of our constructions. We first revisit the idea of BIK+17 Bonawitz et al. (2017) which sprouts from the following simple idea: to let the server learn the sum of the inputs  $x_1, \dots, x_n$  while hiding each individual input  $x_i$ , each individual user  $i$  adds a mask  $h_i$  to its secret input  $x_i$  which is hidden from the server and all other users and can be cancelled out when all the masks are added up, i.e.,  $\sum_{i \in [n]} h_i = 0$ , and sends  $X_i = h_i + x_i$  to the server. By adding all  $X_i$  up, the server obtains the sum of all  $x_i$ . More concretely, assuming  $i > j$  without loss of generality, each pair of users  $i, j$  first agree on a random symmetric secret mask  $mk_{i,j}$ , then they mask their inputs by user  $i$  adding  $mk_{i,j}$  to  $x_i$  while user  $j$  subtracting  $mk_{i,j}$  from  $x_j$ . In other words, each user  $i$  computes a mask  $h_i = \sum_{j < i} mk_{i,j} - \sum_{j > i} mk_{i,j}$  and sends the masked input  $X_i = x_i + h_i$  to the server. The server can get the sum of all  $x_i$  by adding the masked inputs up as  $mk_{i,j}$  and  $-mk_{i,j}$  for each pair of  $i, j$  add up to zero. As long as there are at least two honest users not colluding with other users or the server, the honest users' inputs are hidden from the corrupt parties.

However, this solution only works when all user are always online. If masked input  $X_i$  of some user  $i$  is missing, the sum of  $h_j$  of online users  $j$  will not cancel out in the final sum. To tolerate the fail-stop failure, the protocol adopts  $t$ -out-of- $n$  Shamir's secret sharing scheme which allows a secret value to be divided into  $n$  shares and to be reconstructed with any  $t$  shares of them while guaranteeing that anyone with less than  $t$  shares cannot obtain any information about the secret. More specifically, each user  $i$  shares its masks  $mk_{i,j}$  with all  $n$  users using Shamir's secret sharing before sending the masked input to the server. If any users then fail to send their masked inputs later, the online users can help the server reconstruct their masks as long as there are at least  $t$  users are still online. Also, to prevent the server from directly reconstruct the secret when it forwards the shares for the users, each pair of users  $i, j$  first agree on a symmetric encryption key  $ek_{i,j}$  (with a key exchange algorithm which is introduced in Section 3) and encrypts the shares before they send the shares to each other.

This fix brings another problem, when the server is controlled by a malicious adversary it can lie about the online set and ask online users to help reconstruct  $mk_{i,j}$  of an online user  $i$ . With both the  $X_i$  and  $h_i$ , the server can obtain the secret input  $x_i$ . To tolerate a malicious adversary, each honest user adds another layer of mask  $r_i$  which is uniformly randomly chosen by itself and also secret-shared, shares are denoted by  $r_{i,j}$ , among all users and adds it to the masked input, i.e.,  $X_i = x_i + h_i + r_i$ . To obtain the sum of all  $x_i$  of the online user set  $\mathcal{O}$ , the server needs to remove  $\sum_{i \in \mathcal{O}} r_i$  of the online users from  $\sum_{i \in \mathcal{O}} X_i$  and cancel  $\sum_{i \in \mathcal{O}} h_i$  with  $\sum_{i \notin \mathcal{O}} h_i$ . Thus, if user  $i$  is online in the view of at least  $t$  honest users, then  $r_i$  is reconstructed and can be removed from its masked input, and  $h_i$  is kept hidden and can be cancelled with other users  $j$ 's mask  $h_j$ ; otherwise, if user  $i$  is offline in at least  $t$  honest users' view, these honest users help the server reconstruct  $mk_{i,j}$ . Moreover, all honest users  $i$  use an extra round to agree on the online set in their view by signing the online set and sending to other users their signatures which can be verified with their public keys and cannot be forged by other parties, as otherwise the server can ask different set of users to help it reconstruct the masks of different subset of users. By appropriately setting the threshold  $t$ , for each user the server can recover at most one mask while the other mask is kept hidden so that the input is covered.

**Our first solution: MicroFedML<sub>1</sub>** In the construction above, in each iteration, for every user  $i$  either  $r_i$  or  $h_i$  is revealed, thus two layers of masks are needed and none of them can be reused. The most intuitive change is to let the server only learn the sum of masks rather than each individual mask — this is achievable with the additively homomorphic property of the Shamir secret sharing scheme. In other words, the server can reconstruct the sum of secrets with the sum of shares of different secrets. In this way, only one layer of mask is needed. Each honest user  $i$  first uniformly randomly chooses a mask  $r_i$ , secret shares it to  $\{r_{i,j}\}_{j \in [n]}$  with all users  $j$ , and sends the masked input  $X_i = x_i + r_i$  to the server. Let  $\mathcal{O}$  denote the set of online users  $i$  who successfully send the server  $X_i$ . Then the server requires  $\sum_{i \in \mathcal{O}} r_{j,i}$  from online users  $j$ . As long as at least  $t$  users  $j$  reply, the server can reconstruct  $\sum_{i \in \mathcal{O}} r_i$  and removes it from the sum of  $X_i$  to get the sum of  $x_i$ .

Although the server cannot directly reconstruct any individual mask now, this modification does not allow reusing  $r_i$ . As the server reconstructs the sum of all  $r_i$  of a set of users in every iteration, it can still learn a single user's random mask by accumulating the information of the sum of masks of different user sets in multiple training iterations. For example, if each user  $i$  uses the same  $r_i$  in every iteration, then when user 1 drops offline in some iteration  $k > 1$  while all other users are always online, the server can learn user 1's random mask  $r_1$  from the difference of the sum of  $r$ 's and immediately learn all its historical inputs (and future inputs). To avoid this problem, we further hide the sum of  $r_i$  from the server. Let  $H(\cdot)$  be a hash function mapping a fresh input value to a random generator of a cryptographically secure cyclic group which is easy to compute but very hard to invert. In every iteration  $k$ , each user and the server calculate the hash value  $H(k)$  of the iteration number  $k$ . We depict an overview of our first protocol in Figure 1. Instead of sending  $X_i = r_i + x_i$  and the sum of shares  $\sum_{j \in \mathcal{O}} r_{j,i}$ , now each user sends  $H(k)^{X_i}$  and  $H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}}$  to the server. The server can then reconstruct  $H(k)^{\sum_{i \in \mathcal{O}} r_i}$  in the exponent as described in Section 3. Intuitively, the sum of  $r_i$  is hidden in this way because the finite field is very large so that it is impractical to find the discrete log of  $H(k)^{\sum_{i \in \mathcal{O}} r_i}$  which is uniformly random. At the same time, as  $\sum_{i \in \mathcal{O}} x_i$  is much smaller, the server can obtain it by calculating the discrete log of  $H(k)^{\sum_{i \in \mathcal{O}} X_i} - H(k)^{\sum_{i \in \mathcal{O}} r_i}$ . The protocol description is included in Section 4. We defer the security proof to Appendix B.

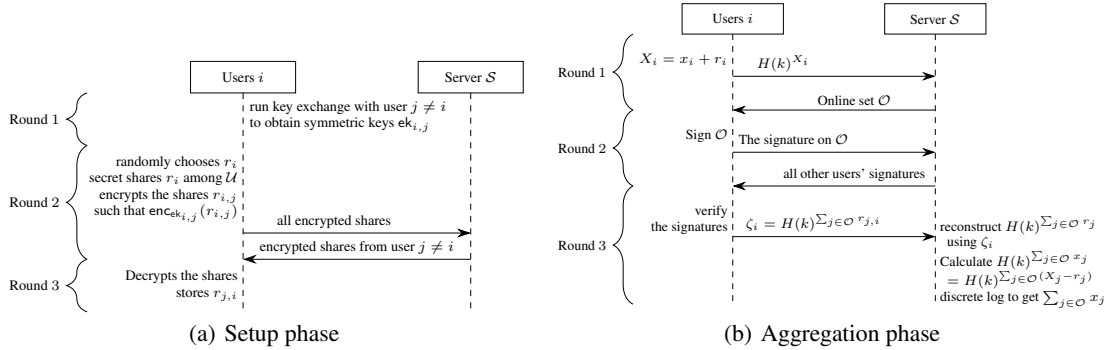


Figure 1: An overview of MicroFedML<sub>1</sub> protocol.

**User grouping: MicroFedML<sub>2</sub>** To further reduce the communication cost, we divide all users into small groups so that each user only needs to know the status of a small number of neighbors in the same group in every iteration. The simplest construction is that each group of users run the previous protocol with the same central server in parallel. The server obtains the sum of all users' inputs by summing up the results of all protocol instances. Obviously, this strategy violates the security requirement that for each iteration, the server can only learn the sum of inputs of a single large subset of users. Thus, we add the mask  $h_i$  generated in a similar way as introduced above so that  $\sum_{i \in [n]} h_i = 0$  to protect the sum of inputs of each small group. As the sum of  $h_i$  for users  $i$  in any single group is random and not known to the server, the server can only learn the global sum in which all  $h_i$  cancel out. This mask should also be secret shared in the group in the same way as sharing  $r_i$  and can also be reused when it is protected in the same way as  $r_i$ . We depict the high-level idea of the Aggregation phase of the protocol in Figure 2 and formally describe the whole protocol and the security properties in Appendix 5.

**Class group for any (larger) input domain** The solutions introduced above require calculating discrete logarithm in a large group. As there is no known efficient algorithm for large range of discrete logarithm, it limits the domain of user inputs. To solve problems with larger input domain, we use the primitive of a group  $\mathbb{G}$  with generator  $g$  in which the DDH assumption holds whereas it contains a subgroup  $\mathbb{F}$  with generator  $f$  in which discrete logarithm is easy to

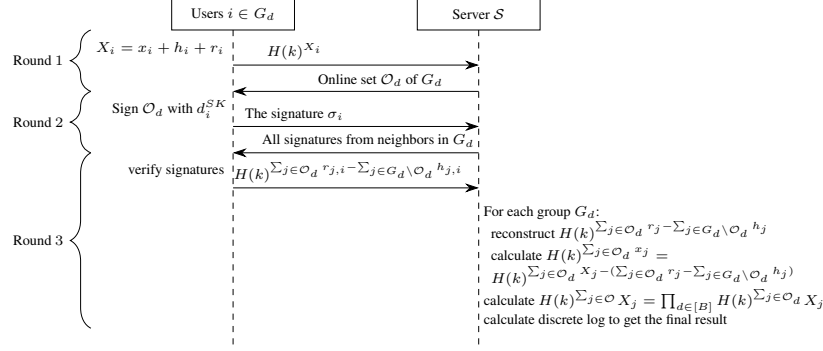


Figure 2: An overview of the Aggregation phase protocol with user grouping. All users are uniformly randomly assigned to  $B$  groups so that each group  $G_d$  for  $d \in [B]$  contains  $n/B$  users. Compared to MicroFedML<sub>1</sub>, in the Setup phase (see Appendix 5.1), each user  $i$  additionally obtains a mask  $h_i$  such that  $\sum_{i \in [n]} h_i = 0$  and secret shares both  $h_i$  and  $r_i$  among its neighbors in the same group. It also obtains the shares  $r_{j,i}$  and  $h_{j,i}$  from its neighbors  $j$ .

compute. This primitive is formalized in Castagnos and Laguillaumie (2015) can be instantiated with class groups as introduced in the same work.

The intuition is to store the input in the exponent of  $f$  so that the discrete logarithm calculation used to recover the sum of input is easy while the random mask is still in the exponent of  $g$  to cover the secret input. The same technique can be also applied to both our protocols MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub>. We discuss this in more detail in 6. In Appendix D, we give a modified version of our first protocol MicroFedML<sub>1</sub> using class groups instead of general cyclic groups.

### 3 Preliminaries

We use  $[n_1, n_2]$  for two integers  $n_1, n_2$  to denote the set of integers  $\{n_1, \dots, n_2\}$ , and we omit the left bound if it equals to 1, i.e.,  $[n]$  denotes the set  $\{1, \dots, n\}$ .

Let  $p, q$  be two primes such that  $p = 2q + 1$ . A finite field  $\mathbb{Z}_p$  is a set of elements  $\{0, 1, \dots, p-1\}$  with multiplication and addition (as well as division and subtraction) operations. Basically, multiplication and addition between elements are conducted as normal arithmetic operations with modulus  $p$ .

**Diffie-Hellman Key Exchange** The Diffie-Hellman key exchange algorithm allows two parties to securely agree on a symmetric secret over a public channel, assuming the discrete log problem is computationally hard. It consists of three algorithms,

- $\text{KA.setup}(\kappa) \rightarrow (\mathbb{G}', g, q, H)$ , in which  $\mathbb{G}'$  is a group of order  $q$  with a generator  $g$ ,  $H$  is a cryptographically secure hash function;
- $\text{KA.gen}(\mathbb{G}', g, q, H) \rightarrow (x, g^x)$  in which  $x$  is uniformly sampled from  $\mathbb{Z}_q$ . This algorithm generates a pair of keys used later in key exchange. The secret key  $x$  should be kept secret, while the public key  $g^x$  will be disclosed to other parties for key exchange.
- $\text{KA.agree}(x_u, g^{x_v}) \rightarrow s_{u,v} = H((g^{x_v})^{x_u})$ . This algorithm allows party  $u$  to obtain the symmetric secret  $s_{u,v} = s_{v,u}$  between party  $u$  and party  $v$  with its own secret key  $x_u$  and the public key  $g^{x_v}$  of party  $v$ .

**Shamir's Secret Sharing** We use Shamir's  $t$ -out-of- $n$  secret sharing in Shamir (1979) to tolerate offline users. Informally speaking, it allows the secret holder to divide the secret into  $n$  shares such that anyone who knows any  $t$  of them can reconstruct the secret, while anyone who knows less than  $t$  shares cannot learn anything about the secret. More specifically, let  $s, x_1, \dots, x_n \in \mathbb{Z}_q$  for some prime  $q$ . The Shamir's Secret Sharing scheme consists of two algorithms:

- $\text{SS.share}(s, \{x_1, x_2, \dots, x_n\}, t) \rightarrow \{(s_1, x_1), \dots, (s_n, x_n)\}$ , in which  $s$  denotes the secret,  $x_1, \dots, x_n$  denotes the  $n$  indices, and  $t$  denotes the threshold of the secret sharing. This function returns a list of shares  $s_i$  of the secret  $s$  with their corresponding indices  $x_i$ .
- $\text{SS.recon}(\{(s_1, x_1), \dots, (s_n, x_n)\}, t) = s$ , in which each pair  $(s_i, x_i)$  denotes the share  $s_i$  on index  $x_i$ . This function returns the original secret  $s$ .

Furthermore, we define an extension of the standard Shamir's secret sharing above,  $\text{SS.expoRecon}((g^{s_1}, x_1), \dots, (g^{s_n}, x_n), t) = g^s$ , which allows the reconstruction of the secret  $s$  with shares  $s_1, \dots, s_n$  being performed in the exponents of finite field elements, and a modified version of Shamir Secret Sharing which works on a bounded interval of integers rather than a finite field. We delay the detail of the definition and the implementation of these functions to Appendix A.1.

**Authenticated Encryption** We use symmetric authenticated encryption to guarantee that the messages between honest parties cannot be either extracted by the adversary or be tampered without being detected. An authenticated encryption scheme consists of two algorithms:  $\text{AE.enc}(m, k) \rightarrow c$ , which encrypts message  $m$  with a key  $k$  and generates a ciphertext  $c$ ; and  $\text{AE.dec}(c, k) \rightarrow m$ , which decrypts the ciphertext  $c$  with the key  $k$  and outputs the original message  $m$ . We assume that the scheme we use satisfies IND-CCA2 security.

**Public Key Infrastructure** A public key infrastructure (PKI) is an arrangement that binds public keys with the respective identities of participants and provides sender authentication for messages. It allows parties to create signatures on messages which can be verified with their public keys and cannot be forged or tampered.

**Negligibility and Indistinguishability** A function  $f : \mathbb{N} \rightarrow \mathbb{R}$  is a negligible function if for every positive integer  $c$  there exists an integer  $n_c$  such that for all  $n > n_c$ ,  $f(n) < \frac{1}{n^c}$ .

We say that an event happens with negligible probability if its probability is a function negligible in the security parameter. Symmetrically, we say that an event happens with overwhelming probability if it happens with 1 but negligible probability.

We say that two ensembles of probability distributions  $\{X_n\}_{n \in \mathbb{N}}$  and  $\{Y_n\}_{n \in \mathbb{N}}$  are computationally indistinguishable (denoted with  $\approx_c$ ) if for all non-uniform PPT distinguisher  $\mathcal{D}$ , there exists a negligible function  $f$  such that for all  $n \in \mathbb{N}$ ,

$$\left| \Pr_{t \leftarrow X_n} [\mathcal{D}(1^n, t) = 1] - \Pr_{t \leftarrow Y_n} [\mathcal{D}(1^n, t) = 1] \right| < f(n).$$

**Finite Field and Cyclic Group** Let  $p, q$  be two primes such that  $p = 2q + 1$ .  $\mathbb{Z}_p$  denotes a finite field with elements  $\{0, 1, \dots, p-1\}$  and  $\mathbb{Z}_p^*$  denotes a group  $\{1, \dots, p-1\}$ .  $\mathbb{G}$  refers to a subgroup of  $\mathbb{Z}_p^*$  of order  $q$ , which is also a cyclic group and every element in it is a generator of the group. In other word, for any element  $g \in \mathbb{G}$ ,  $\mathbb{G} = \{g^0, g^1, \dots, g^{q-1}\}$ . In the protocol description in this paper, by uniformly randomly choosing some value, we mean uniformly randomly choosing an element from  $\mathbb{Z}_q$  if not noted explicitly; by computing  $g^r$  or  $\log_g R$  for some element  $g \in \mathbb{G}$ , we mean the power and discrete log computation happening in group  $\mathbb{G}$ .

**Decisional Diffie-Hellman (DDH) Assumption** In our protocol, we assume that the following assumption holds: Let  $p, q$  be two primes,  $p = 2q + 1$ . Let  $g$  be a generator of  $\mathbb{Z}_p^*$ . Then the following two distributions are computationally indistinguishable, given that  $a, b, c$  are independently and uniformly randomly chosen from  $\mathbb{Z}_q$ :

$$(g^a, g^b, g^{ab}) \text{ and } (g^a, g^b, g^c).$$

**Hypergeometric Distribution** The hypergeometric distribution  $X \sim \text{HyperGeom}(N, m, n)$  is a discrete probability distribution that describes the probability of picking  $X$  objects with some specific feature in  $n$  draws, without replacement, from a finite population of size  $N$  that contains exactly  $m$  objects with that feature.

We use the following tail bounds for  $X \sim \text{HyperGeom}(N, m, n)$ :

- $\forall d > 0 : \Pr[X \leq (m/N - d)n] \leq e^{-2d^2 n},$
- $\forall d > 0 : \Pr[X \geq (m/N + d)n] \leq e^{-2d^2 n}.$

**DDH Group with an Easy DL Subgroup** To accommodate larger input, we use an assumption from Castagnos and Laguillaumie (2015). Intuitively, it assumes the existence of a group  $\mathbb{G} = \langle g \rangle$  with a subgroup  $\mathbb{F} = \langle f \rangle$  such that the DDH assumption holds in  $\mathbb{G}$  and discrete logarithm is easy to calculate in the group  $\mathbb{F}$ . More formally, we use the following definition from Castagnos and Laguillaumie (2015):

**Definition 3.1** (DDH Group with an Easy DL Subgroup Castagnos and Laguillaumie (2015)). *A DDH group with an easy DL subgroup is a pair of algorithms (cl.gen, cl.solve). The cl.gen algorithm is a group generator which takes as input two parameters  $\lambda$  and  $\mu$  and outputs a tuple  $(B, N, p, s, g, f, \mathbb{G}, \mathbb{F})$ . The integers  $B, N, p$  and  $s$  are such that  $s$  is a  $\lambda$ -bit integer,  $p$  is a  $\mu$ -bit integer,  $\gcd(p, s) = 1$ ,  $N = p \cdot s$ , and  $B$  is an upper bound for  $s$ . The set  $(\mathbb{G}, \cdot)$  is a cyclic*

group of order  $n$  generated by  $g$ , and  $\mathbb{F} \subset \mathbb{G}$  is the subgroup of  $G$  of order  $p$  and  $f$  is a generator of  $F$ . The upper bound  $B$  is chosen such that the distribution induced by  $\{g^r, r \xleftarrow{\$} \{0, \dots, Bp - 1\}\}$  is statistically indistinguishable from the uniform distribution on  $\mathbb{G}$ . We assume that the canonical surjection  $\pi : \mathbb{G} \rightarrow \mathbb{G}/\mathbb{F}$  is efficiently computable from the description of  $\mathbb{G}, \mathbb{F}$  and  $p$  and that given an element  $h \in \mathbb{G}/\mathbb{F}$  one can efficiently lift  $h$  in  $\mathbb{G}$ , i.e., compute an element  $h_\ell \in \pi^{-1}(h)$ . We suppose moreover that:

1. The DL problem is easy in  $F$ . The  $\text{cl.solve}$  algorithm is a deterministic polynomial time algorithm that solves the discrete logarithm problem in  $F$ :

$$\begin{aligned} \Pr[x = x^* : (B, N, p, s, g, f, \mathbb{G}, \mathbb{F}) \xleftarrow{\$} \text{cl.gen}(1^\lambda, 1^\mu), \\ x \xleftarrow{\$} \mathbb{Z}/p\mathbb{Z}, X = f^x, \\ x^* \leftarrow \text{cl.solve}(B, p, g, f, \mathbb{G}, \mathbb{F}, X)] = 1 \end{aligned}$$

2. The DDH problem is hard in  $G$  even with access to the  $\text{cl.solve}$  algorithm:

$$\begin{aligned} |\Pr[b = b^* : (B, N, p, s, g, f, \mathbb{G}, \mathbb{F}) \xleftarrow{\$} \text{cl.gen}(1^\lambda, 1^\mu), \\ x, y, z \xleftarrow{\$} \mathbb{Z}/N\mathbb{Z}, X = g^x, Y = g^y, \\ b \xleftarrow{\$} \{0, 1\}, Z_0 = g^z, Z_1 = g^{xy}, \\ b^* \xleftarrow{\$} \mathcal{A}(B, p, g, f, \mathbb{G}, \mathbb{F}, X, Y, Z_b, \text{cl.solve}(\cdot))] \\ - \frac{1}{2}| \leq \text{negl}(\lambda) \end{aligned}$$

for any probabilistic polynomial time adversary  $\mathcal{A}$ .

### 3.1 Security Definition

In this section, we formally define the secure aggregation protocol and the security property for a multi-iteration secure aggregation protocol.

**Definition 3.2** (Aggregation Protocol). An aggregation protocol  $\Pi(\mathcal{U}, \mathcal{S}, K)$  with a set of users  $\mathcal{U}$ , a server  $\mathcal{S}$ , integers  $K$  as parameters consists of two phases: the Setup phase and the Aggregation phase. The Setup phase runs once at the beginning of the execution, then the Aggregation phase runs for  $K$  iterations. At the beginning of each iteration  $k \in [K]$  of the Aggregation phase, each user  $i \in \mathcal{U}$  holds a input  $x_i^k$ , and at the end of each iteration  $k$ , the server  $\mathcal{S}$  outputs a value  $w^k = \sum_{i \in \mathcal{U}} x_i^k$ .

We define the correctness and privacy property of the protocol below.

**Definition 3.3** (Correctness with Dropouts). Let  $n = |\mathcal{U}|$ . An aggregation protocol  $\Pi$  guarantees correctness with  $\delta$  offline rate if for every iteration  $1 \leq k \leq K$  and for all sets of offline users  $\text{offline}_k \subset \mathcal{U}$  with  $|\text{offline}_k| < \delta n$ , the server outputs  $w^k = \sum_{i \in \mathcal{U} \setminus \text{offline}_k} x_i^k$  at the end of iteration  $k$  if every user and the server follows the protocol except that the users in  $\text{offline}_k$  drops offline at some point in iteration  $k$ .

**Ideal Functionality** To define privacy property, we first describe an ideal functionality which allows the adversary to learn the sum of secrets of all honest and online users chosen by the adversarial server in every iteration. More formally,  $\text{Ideal}_{\{x_i^k\}_{i \in \mathcal{U} \setminus \mathcal{C}, k}}^\delta$  is an ideal oracle, which can be queried once for each iteration  $k \in [K]$ . When queried with a large enough set of honest users  $U$  and the iteration  $k$ , it provides  $\sum_{i \in U} x_i^k$ . More specifically, given a set of users  $U$  and an iteration number  $k$ , it operates as follows:

$$\text{Ideal}_{\{x_i^k\}_{i \in \mathcal{U} \setminus \mathcal{C}, k}}^\delta(U, k) = \begin{cases} \sum_{i \in U} x_i^k & \text{if } U \subseteq (\mathcal{U} \setminus \mathcal{C}) \\ & \text{and } |U| > (1 - \delta)|\mathcal{U}| - |\mathcal{C}|, \\ \perp & \text{otherwise.} \end{cases}$$

**Definition 3.4** (Privacy against Semi-honest/Malicious Adversary). Let  $K, n$  be integer parameters. Let  $\Pi$  be a multi-iteration secure aggregation protocol running with one central server  $\mathcal{S}$  and a set of  $n$  users  $\mathcal{U} = \{1, \dots, n\}$ . An aggregation protocol  $\Pi$  guarantees privacy against  $\gamma$  fraction of semi-honest/malicious adversary with  $\delta$  offline rate if there exists a PPT simulator  $\text{SIM}$  such that for all  $k = 1, \dots, K$ , all inputs vectors  $X^k = \{x_1^k, \dots, x_n^k\}$  for each iteration  $1 \leq k \leq K$ , and all sets of corrupted users  $\mathcal{C} \subset \mathcal{U}$  with  $|\mathcal{C}| < \gamma n$  controlled by an honest-but-curious/malicious



adversary  $M_C$  which also controls the server  $\mathcal{S}$ , the output of SIM is computationally indistinguishable from the joint view of the server and the corrupted users in that execution, i.e.,

$$\begin{aligned} & \text{REAL}_{\mathcal{C}}^{\mathcal{U}, K}(M_C, \{x_i^k\}_{i \in \mathcal{U} \setminus \mathcal{C}, k \in [K]}) \\ & \approx_c \text{SIM}_{\mathcal{C}}^{\mathcal{U}, K, \text{Ideal}^\delta_{\{x_i^k\}_{i \in \mathcal{U} \setminus \mathcal{C}, k \in [K]}}}(M_C) \end{aligned}$$

## 4 MicroFedML<sub>1</sub>: Basic Solution

The protocol runs with one server and  $n$  users  $1, 2, \dots, n$ , which can only communicate with the server through secure channels. The protocol consists of two phases: the Setup phase and the Aggregation phase. The Setup phase runs only once at the beginning of the protocol, and the Aggregation phase runs for  $K$  iterations after the Setup phase completes. We assume that each user holds a secret input at the beginning of each iteration of the Aggregation phase. Users can drop offline at any time point during the execution. We describe the Setup phase in Algorithm 1 and the Aggregation phase in Algorithm 2. We mark the part of execution that only needed in malicious settings with blue color and underlines.

### 4.1 Protocol

---

#### Algorithm 1 Setup (MicroFedML<sub>1</sub>)

---

This protocol uses the following algorithms defined in Section 3: a Public key infrastructure, a Diffie-Hellman key exchange scheme (KA.setup, KA.gen, KA.agree); a CCA2-secure authenticated encryption scheme (AE.enc, AE.dec); a Shamir's secret sharing scheme (SS.share, SS.recon, SS.expoRecon). It proceeds as follows:

**Input:** A central server  $\mathcal{S}$  and a user set  $\mathcal{U}$  of  $n$  users. Each user can communicate with the server through a private authenticated channel. All parties are given the public parameters: the security parameter  $\kappa$ , the number of users  $n$ , a threshold value  $t$ , honestly generated  $pp \leftarrow \text{KA.setup}(\kappa)$  for key agreement, the input space, and a field  $\mathbb{Z}_q$  for secret sharing.

Moreover, every party  $i$  holds its own signing key  $d_i^{SK}$  and a list of verification keys  $d_j^{PK}$  for all other parties  $j$ . The server  $\mathcal{S}$  also has all users' verification keys.

**Output:** Every user  $i \in \mathcal{U}$  either obtains a set of users  $\mathcal{U}_i$  such that  $|\mathcal{U}_i| \geq t$  and a share  $r_{j,i}$  of a secret value  $r_j$  for each  $j \in \mathcal{U}_i$  or aborts. The server either outputs a set of users  $\mathcal{U}_S$  such that  $|\mathcal{U}_S| \geq t$  or aborts.

##### Round 1: Encryption Key Exchange

- 1: Each user  $i \in \mathcal{U}$ : generates a pair of encryption keys  $(sk_i, pk_i) \leftarrow \text{KA.gen}(pp)$ , then signs  $pk_i$  with  $d_i^{SK}$  and sends  $(pk_i, \sigma_i)$  to the server, in which  $\sigma_i$  denotes the signature.
- 2: Server  $\mathcal{S}$ : On receiving  $(pk_i, \sigma_i)$  from user  $j$ , the server verifies the signature  $\sigma_j$  with  $d_j^{PK}$ . If the signature verification fails, ignore the message from user  $j$ . Otherwise, add  $j$  to a user list  $\mathcal{U}_S^1$ . If  $|\mathcal{U}_S^1| < t$  after processing all messages from users,  $\mathcal{S}$  aborts. Otherwise, the server sends all public keys and signatures it receives from users  $j \in \mathcal{U}_S^1$  to each user in  $\mathcal{U}_S^1$ .

##### Round 2: Mask Sharing

- 3: Each user  $i$ : On receiving  $(pk_i, \sigma_i)$  for a user  $j \in \mathcal{U}$  from the server, each user  $i$  verifies the signatures  $\sigma_j$  with  $d_j^{PK}$ . It aborts if any signature verification fails as that indicates the server is corrupt. Otherwise, it puts  $j$  into a user list  $\mathcal{U}_i^1$  and stores  $ek_{i,j} = \text{KA.agree}(pk_j, sk_i)$ . It aborts if  $|\mathcal{U}_i^1| < t$  after processing all received messages. Otherwise, user  $i$  uniformly randomly chooses  $r_i$ , and calculates the secret shares of  $r_i$  by  $\{r_{i,j}\}_{j \in \mathcal{U}} \leftarrow \text{SS.share}(r_i, \mathcal{U}_i^1, t)$ . Then it encrypts each share  $r_{i,j}$  by  $c_{i,j} \leftarrow \text{AE.enc}(r_{i,j}, ek_{i,j})$  and sends all encrypted shares  $\{c_{i,j}\}_{j \in \mathcal{U}_i^1}$  to the server.
- 4: Server  $\mathcal{S}$ : If it receives messages from less than  $t$  users, abort. Otherwise, it denotes this set of users with  $\mathcal{U}_S$ . It sends each  $c_{i,j}$  to the corresponding receiver  $j$  for each  $i \in \mathcal{U}_S$ . Then it outputs the client set  $\mathcal{U}_S$ .

##### Round 3: User Receiving Shares

- 5: Each user  $i$  If it receives  $c_{j,i}$  for less than  $t$  users  $j$  from the server, abort. Otherwise, decrypt each encrypted share by  $r_{j,i} = \text{AE.dec}(c_{j,i}, ek_{i,j})$ . If the decryption of the share from user  $j$  fails, it ignores the encrypted share. Otherwise, it puts  $j$  into a user set  $\mathcal{U}_i^2$  and stores  $r_{j,i}$ . If  $|\mathcal{U}_i^2| < t$  after processing all shares, it aborts. Otherwise, it stores  $r_i$ , the set  $\mathcal{U}_i = \mathcal{U}_i^2$ , and all  $r_{j,i}$  for  $j \in \mathcal{U}_i$ .
- 

---

#### Algorithm 2 Aggregation (MicroFedML<sub>1</sub>)

---

This protocol uses the following algorithms defined in Section 3: a Public key infrastructure, a Shamir's secret sharing scheme (SS.share, SS.recon, SS.expoShare, SS.expoRecon), a hash function  $H(\cdot)$ . It proceeds as follows:

**Input:** Every user  $i$  holds its own signing key  $d_i^{SK}$  and all users' verification key  $d_j^{PK}$  for  $j \in [n]$ ,  $r_i$ , a list of users  $\mathcal{U}_i$ , and  $r_{j,i}$  for every  $j \in \mathcal{U}_i$  it obtains in the Setup phase. Moreover, it also holds a secret input  $x_i^k$  for every iteration  $k$ . The server  $\mathcal{S}$  holds all users' verification keys, all public parameters it receives in the Setup phase, and a list of users  $\mathcal{U}_{\mathcal{S}}$  which is its output of the Setup phase.

**Output:** For each iteration  $k$ , if there are at least  $t$  users being always online during iteration  $k$ , then at the end of iteration  $k$ , the server  $\mathcal{S}$  outputs  $\sum_{i \in \mathcal{O}^k} x_i^k$ , in which  $\mathcal{O}^k$  denotes a set of users of size at least  $t$ .

**Note:** For simplicity of exposition, we omit the superscript  $k$  of all variables when it can be easily inferred from the context.

1: **for** Iteration  $k = 1, 2, \dots$  **do**

**Round 1: Secret Sharing:**

2:     **User  $i$ :** It calculates  $X_i = x_i + r_i$  and sends  $H(k)^{X_i}$  to the server.

3:     **Server  $\mathcal{S}$ :** Denote the set of users it receives messages from with  $\mathcal{O}$ . If  $|\mathcal{O}| < t$ , abort. Otherwise, it sends  $\mathcal{O}$  to all users  $i \in \mathcal{O}$ .

**Round 2: Online Set Checking (Only needed in Malicious setting):**

4:     **User  $i$ :** On receiving  $\mathcal{O}$  from the server, it first checks that  $\mathcal{O} \subseteq \mathcal{U}_i$  and  $|\mathcal{O}| \geq t$ , then signs the set  $\mathcal{O}$  and sends the signature  $\sigma_i$  to the server.

5:     **Server  $\mathcal{S}$ :** If it receives less than  $t$  valid signatures on  $\mathcal{O}$ , abort. Otherwise, it forwards all valid signatures to all users in  $\mathcal{O}$ .

**Round 3: Mask Reconstruction on the Exponent:**

6:     **User  $i$ :** On receiving signatures from the server, it first verifies the signatures with  $\mathcal{O}$  and the verification keys of the other users. If there are less than  $t$  valid signatures, abort. Otherwise, it calculates  $\zeta_i = H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}}$ . It sends  $\zeta_i$  to the server.

7:     **Server  $\mathcal{S}$ :** If it receives  $\zeta_i$  from less than  $t$  users, abort. Otherwise, let  $\mathcal{O}'$  denote the set of users  $i$  successfully sends  $\zeta_i$  to the server. The server reconstructs  $R_{\mathcal{O}} = \text{SS.expoRecon}(\{\zeta_j, j\}_{j \in \mathcal{O}'}, t)$  and calculates the discrete log of  $H(k)^{\sum_{i \in \mathcal{O}} X_i} / R_{\mathcal{O}}$  to get  $\sum_{i \in \mathcal{O}} x_i$ .

8: **end for**

## 4.2 Correctness with Dropouts

We first discuss the correctness guarantee of the protocol when all users and the server follow the protocol except that less than  $\delta$  fraction of users are offline in each iteration. The correctness property is easy to see when the server gets enough shares (i.e.,  $|\mathcal{O}'| > (1 - \delta)n$ ) in the last round of each iteration of the aggregation phase to reconstruct  $R_{\mathcal{O}} = H(k)^{\sum_{i \in \mathcal{O}} r_i}$ . The condition is satisfied when there are less than  $\delta$  fraction of users are ever offline in the iteration and the threshold  $t$  of secret sharing is set as  $\lfloor (1 - \delta)n \rfloor + 1$ . Thus, we have the following theorem.

**Theorem 4.1.** *The protocol  $\Pi$  instantiated with Algorithm 1 and Algorithm 2 with parameter  $t = \lfloor (1 - \delta)n \rfloor + 1$  guarantees correctness with  $\delta$  offline rate.*

## 4.3 Privacy against Semi-Honest and Malicious Adversaries

we discuss the privacy guarantee of the protocol when the semi-honest/malicious adversary controls both a subset of users and the server. Due to the limited space, we only provide the theorems here and defer the proofs to Appendix B.

**Theorem 4.2** (Privacy against Semi-honest Adversary). *The protocol  $\Pi$  instantiated with Algorithm 1 and Algorithm 2 with parameter  $t = \lfloor n/2 \rfloor + 1$  guarantees privacy against  $\frac{1}{2}$ -fraction of semi-honest adversary with  $\frac{1}{2}$  offline rate.*

**Theorem 4.3** (Privacy against Malicious Adversary). *The protocol  $\Pi$  instantiated with Algorithm 1 and Algorithm 2 (including the underlined parts) with parameter  $t = \lfloor 2n/3 \rfloor + 1$  guarantees privacy against  $\frac{1}{3}$ -fraction of malicious adversary with  $\frac{1}{3}$  offline rate.*

## 5 MicroFedML<sub>2</sub>: Improvement with User Grouping

In the protocol introduced in Section 4 we eliminate the communication cost of secret sharing in each iteration by reusing the random mask. However, each user still needs to know the online status of all participants to calculate the sum of the shares of the mask, which needs to be represented in at least  $O(n)$  bits. In the malicious setting, this also means each user needs to receive  $O(n)$  signatures of other online users from the server to guarantee the agreement on the online set. In this section, we further reduce the communication cost by grouping the users, so that each user only needs to know the status of a small number of neighbors in the same group. We show the Setup phase and the Aggregation phase of MicroFedML<sub>2</sub> with privacy guarantee against malicious adversary in Figure 3 and Figure 4

respectively, then we give the formal description of the protocol in Section 5.1. Due to the length limit, we delay the correctness proof to Appendix C.2 and the privacy proof to Appendix C.3.

For simplicity, we assume that the the group assignment is provided by the trusted third party as part of the inputs, but the assignment can also be implemented with a distributed randomness generation protocol to allow all users to decide the assignment together. We chooses the group size in exactly the same way and under the same assumptions as neighborhood size is chosen in BBG+20 Bell et al. (2020). We discuss the group properties we need in Appendix C.1.

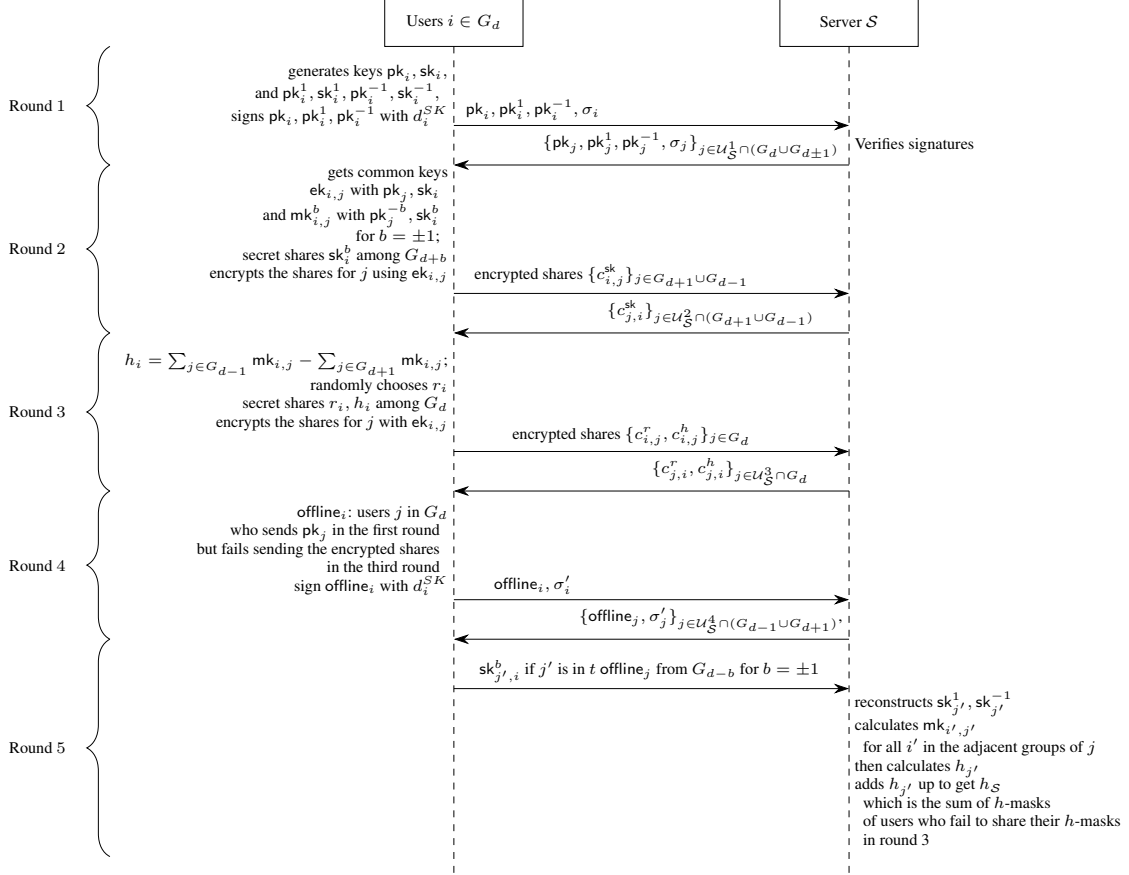


Figure 3: An overview of the Setup phase of MicroFedML<sub>2</sub>

## 5.1 Protocol

The protocol runs with one server and  $n$  users  $1, 2, \dots, n$ , which can only communicate with the server through secure channels. Same as the previous protocol, the server and the users perform the Setup phase first, then execute the Aggregation phase for  $K$  iterations.

We describe the Setup phase in Algorithm 3 and the Aggregation phase in Algorithm 4. The parts that only needed to protect privacy against malicious adversary are marked with blue color and underlines.

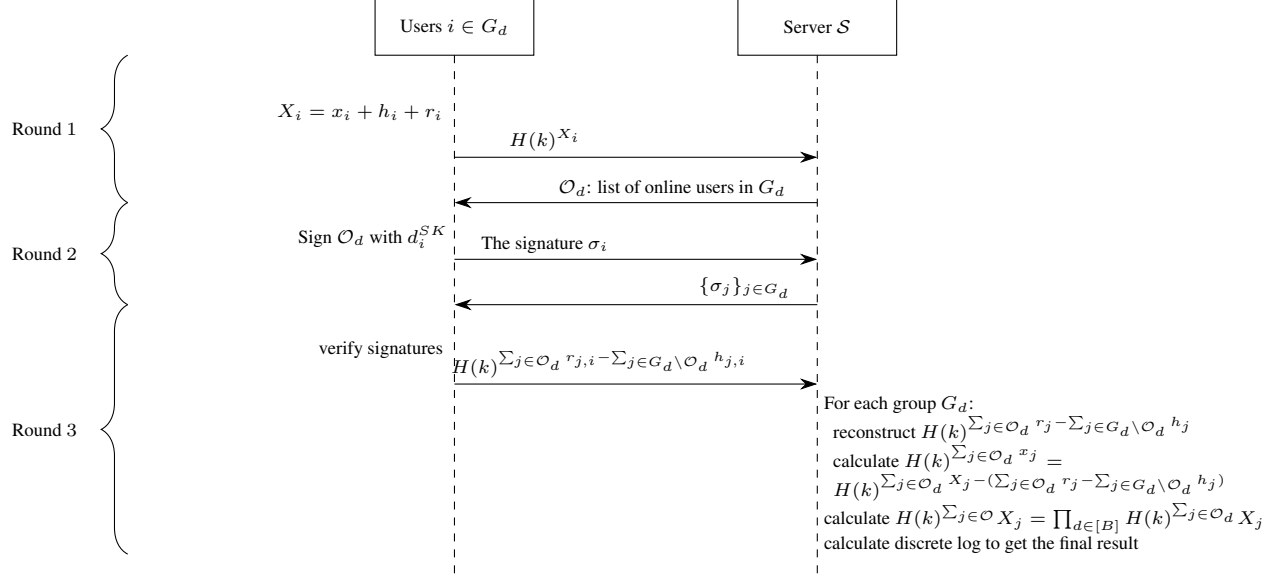
---

### Algorithm 3 Setup (MicroFedML<sub>2</sub>)

---

This protocol uses the following algorithms defined in Section 3: a public key infrastructure, a Diffie-Hellman key exchange scheme (KA.setup, KA.gen, KA.agree); a CCA2-secure authenticated encryption scheme (AE.enc, AE.dec); a Shamir's secret sharing scheme (SS.share, SS.recon, SS.expoShare, SS.expoRecon); It also accesses a random oracle  $H(\cdot)$ , which has range in  $\mathbb{Z}_p^*$ . It proceeds as follows:

**Input:** A central server  $S$  and a user set  $\mathcal{U}$  of  $n$  users. Each user can communicate with the server through a private authenticated channel. All parties are given public parameters: the security parameter  $\kappa$ , the number of users  $n$ , a threshold value  $t$ , honestly generated  $pp \leftarrow \text{KA.setup}(\kappa)$  for key agreement, the input space  $\mathcal{X}$ , a field  $\mathbb{F}$  for secret sharing.

Figure 4: An overview of the Aggregation phase of MicroFedML<sub>2</sub>

Moreover, all clients are uniformly randomly divided into  $B$  groups  $G_1, \dots, G_B$ , each of which contains  $n/B$  clients. For convenience, in the description of the protocol, let  $G_{-1} = G_B$ , and  $G_{B+1} = G_1$ . The user  $i$  in group  $d$  holds the group index  $d$ , its own signing key  $d_i^{SK}$  and a list of verification keys  $d_j^{PK}$  for  $j \in [n]$ . The server  $S$  also has all users' verification keys.

**Output:** Every user  $i \in \mathcal{U}$  who is online through the Setup phase either obtains a set of users  $\mathcal{U}_i$  of size at least  $t$  and two shares  $r_{j,i}, h_{j,i}$  for each  $j \in \mathcal{U}_i$  or abort. The server either obtains a set of users  $\mathcal{U}_S$  such that  $|\mathcal{U}_S| \geq Bt$  and a global mask  $h_S$  or abort.

#### Round 1: Key Exchange:

- 1: Each user  $i \in G_d$ : It generates a pair of encryption keys  $(pk_i, sk_i)$  and two pairs of masking keys  $(pk_i^1, sk_i^1)$  and  $(pk_i^{-1}, sk_i^{-1})$ . It sends the three public keys with signatures on them to the server.
- 2: Server  $S$ : Let  $\mathcal{U}_S^1$  denotes the set of users send the server public keys with valid signatures. For  $i \in \mathcal{U}_S^1 \cap G_d$ , the server distributes the public encryption key  $pk_i$  and the signature received from user  $i \in G_d$  to all users in  $\mathcal{U}_S^1 \cap (G_d \cup G_{d-1} \cup G_{d+1})$ , and distributes the public masking key  $pk_i^b$  with the signatures to all users in  $\mathcal{U}_S^1 \cap G_{d+b}$  for  $b \in \{-1, 1\}$ .

#### Round 2: Secret Mask Key Sharing:

- 3: Each user  $i \in G_d$ : Let  $\mathcal{U}_i^1$  denote the set of users from whom user  $i$  receives the public keys with valid signatures. Note that now  $\mathcal{U}_i^1 \subseteq G_d \cup G_{d-1} \cup G_{d+1}$ . It runs the key exchange algorithm to generate  $ek_{i,j}$  for  $j \in \mathcal{U}_i^1$  as described in line 3 in Algorithm 1. Moreover, it runs the key exchange algorithm to generate  $mk_{i,j}$  for  $j \in G_{d+b}$  with  $sk_i^b$  and  $pk_j^{-b}$  for  $b = \{1, -1\}$ .

It then calculates  $t$ -out-of- $\frac{n}{B}$  secret shares of  $sk_i^b$  among users in  $G_{d+b}$  to generate  $\{sk_{i,j}^b\}_{j \in G_{d+b}}$ , and encrypts each share with  $ek_{i,j}$  to generate cipher text  $c_{i,j}^{sk^b}$  for  $b = \{-1, 1\}$ . It then sends all encrypted shares to the server.

- 4: Server  $S$ : Let  $\mathcal{U}_S^2$  denote the set of users who successfully send the server messages. If for any group  $G_d$ ,  $|\mathcal{U}_S^2 \cap G_d| < t$ , abort. Otherwise, For each group  $d \in [B]$  and each  $i \in \mathcal{U}_S^2 \cap G_d$ , The server sends each encrypted share  $c_{i,j}^{sk^b}$  to the corresponding receiver  $j \in G_{d+b}$ .

#### Round 3: Mask Sharing:

- 5: Each user  $i \in G_d$ : Denote the set of users  $j \in G_{d-b}$  from who user  $i$  receives  $c_{j,i}^{sk^b}$  for  $b = \{1, -1\}$  with  $\mathcal{U}_i^2$ . It decrypts the encrypted share by  $sk_{j,i}^b = \text{AE.dec}(c_{j,i}^{sk^b}, ek_{i,j})$ . If any  $c_{j,i}^{sk^b}$  for  $j \in \mathcal{U}_i^2 \cap G_{d-b}$  cannot be correctly decrypted, remove  $j$  from  $\mathcal{U}_i^2$ . It then checks if for  $b \in \{1, -1\}$ ,  $|\mathcal{U}_i^2 \cap G_{d+b}| < t$ . If yes, abort.

Otherwise, user  $i$  uniformly randomly chooses a self mask ( $r$ -mask) and calculates the  $h$ -mask  $h_i = \sum_{j \in \mathcal{U}_i^2 \cap G_{d-1}} mk_{i,j} - \sum_{j \in \mathcal{U}_i^2 \cap G_{d+1}} mk_{i,j}$ . Then it calculates the shares of  $r_i$  and  $h_i$  among  $j \in \mathcal{U}_i^1 \cap G_d$

and encrypts the shares with  $ek_{i,j}$  as described in line 3 of Algorithm 1. It then sends the encrypted shares  $\{c_{i,j}^r, c_{i,j}^h\}_{j \in \mathcal{U}_i^1 \cap G_d}$  to the server.

- 6: **Server  $\mathcal{S}$ :** Denote the set of all users  $i$  who successfully sends the server encrypted shares with  $\mathcal{U}_S^3$ . If for any group  $G_d$ ,  $|\mathcal{U}_S^3 \cap G_d| < t$ , abort. Otherwise, It sends the shares to the corresponding receiver  $j$  for each  $i \in \mathcal{U}_S^3$ , and an offline set of group  $d$   $\text{offline}_d = G_d \cap (\mathcal{U}_S^2 \setminus \mathcal{U}_S^3)$  to users in  $\mathcal{U}_S^3 \cap (\mathcal{U}_{d-1} \cup \mathcal{U}_{d+1})$ . (The server doesn't need to send this offline set in the malicious setting. Instead, it waits for the users to send the offline sets in their views with their signatures as described in the red underlined part of Round 4.)

**Round 4: Agreeing on the Offline User Set:**

- 7: Each user  $i \in G_d$ : It decrypts each received encrypted share by  $r_{j,i} = \text{AE.dec}(c_{j,i}^r, ek_{i,j})$ , and  $h_{j,i} = \text{AE.dec}(c_{j,i}^h, ek_{i,j})$ . If the decryption of the share from user  $j$  fails, it ignores the message from user  $j$ . Otherwise, it puts  $j$  into a user set  $\mathcal{U}_i^3$  and stores  $r_{j,i}$  and  $h_{j,i}$ . If  $|\mathcal{U}_i^3| < t$  after processing all shares, it aborts. Then it signs and sends a user list  $\text{offline}_i = (\mathcal{U}_i^1 \cap G_d) \setminus \mathcal{U}_i^3$  with the signature  $\sigma_i$  on the list to the server.
- 8: **Server  $\mathcal{S}$ :** If the server receives  $\text{offline}_i$  with valid signatures  $\sigma_i$  from less than  $t$  users  $i$  from any group  $G_d$ , abort. Otherwise, denote the set of users  $i$  who send the offline lists with valid signatures to the server with  $\mathcal{U}_S^4$ . The server sends the list and the signature  $(\text{offline}_i, \sigma_i)$  for all  $i \in G_d \cap \mathcal{U}_S^4$  to all users in  $(G_{d-1} \cup G_{d+1}) \cap \mathcal{U}_S^4$ .

**Round 5: Reconstructing Offline Users' Masks:**

- 9: Each user  $i \in G_d$ : After receiving all user lists with the signature  $(\text{offline}_j, \sigma_j)$  from the server, it verifies the signatures and aborts if any signature verification fails. It also aborts if it receives less than  $t$  offline lists with valid signatures from group  $G_{d-1}$  or  $G_{d+1}$ . Otherwise, for group  $G_{d-1}$ , it checks if there is any user  $j' \in G_{d-1} \cap \mathcal{U}_i^2$  being included in at least  $t$  offline lists it receives from users in  $G_{d-1}$ . If yes, put them in a list  $\text{offline}_{d-1}$ . It repeats the process on group  $G_{d+1}$ . It sends  $sk_{j',i}^1$  for  $j' \in \text{offline}_{d-1}$  and  $sk_{j',i}^{-1}$  for  $j' \in \text{offline}_{d+1}$  to the server. It also stores  $\mathcal{U}_i = \mathcal{U}_i^3$  and  $r_{j,i}, h_{j,i}$  for  $j \in \mathcal{U}_i$ .
- 10: **Server  $\mathcal{S}$ :** For each group  $d$ , if for a user  $i \in G_d$  the server receives at least  $t$  shares  $sk_{i,j}^b$  from both groups  $G_{d+b}$  for  $b \in \{-1, 1\}$ , the server puts  $i$  into a user list  $\text{offline}_S$ . Each user in this list fails to share their  $r$ - and  $h$ -masks with their group members in Round 3, while the symmetric masking keys  $mk_{i,j}$  between itself and the member  $j$  of  $i$ 's neighbor group have been included in the  $h_j$ . Thus, the server needs to calculate  $h_i$  by reconstructing  $sk_i^b$  for  $b = \pm 1$ , running the key exchange algorithm for  $i$  and user  $j \in \mathcal{U}_S^2 \cap G_{d-1}$  by  $mk_{i,j} = \text{KA.agree}(pk_j^1, sk_i^{-1})$  and for  $i$  and user  $j \in \mathcal{U}_S^2 \cap G_{d+1}$  by  $mk_{i,j} = \text{KA.agree}(pk_j^{-1}, sk_i^1)$ , and calculating  $h_i = \sum_{j \in \mathcal{U}_S^2 \cap G_{d-1}} mk_{i,j} - \sum_{j \in \mathcal{U}_S^2 \cap G_{d+1}} mk_{i,j}$ . Then it obtains  $\mathcal{U}_S = \mathcal{U}_S^2 \setminus \text{offline}_S$  and  $h_S = \sum_{i \in \text{offline}_S} h_i$ .

As the participants may drop offline in any round in the execution, some users  $i$  might fail to complete the key exchange process, to send out the encrypted shares of  $sk_i^1$  and  $sk_i^{-1}$ , or to share the two masks  $c_{i,j}^h$  and  $c_{i,j}^r$  in the Setup phase. If a user fails to finish key exchange for  $ek$  or  $mk$  with the other users, it will not be considered in the calculation of other users'  $h$ -masks or participate the Aggregation phase. However, if a user  $i$  drops offline at the end of the second round of the Setup phase, i.e., after exchanging  $mk_{i,j}$  with other users  $j$  and sending the encryption of shares  $sk_{i,j}^1$  and  $sk_{i,j}^{-1}$ , user  $j$  will include  $mk_{i,j}$  in its  $h$ -mask, while user  $i$  fails to share its own  $h_i$  with the other group members in  $G_d$  in the third round. In this case, if user  $i$  drops offline in the Aggregation phase, the online users in  $G_d$  are not able to help the server reconstruct  $h_i$ , and the  $h$ -masks in the final sum will not cancel out. Thus, the server needs to reconstruct  $sk_i^1$  and  $sk_i^{-1}$  and calculates the  $h$ -masks of these users  $i$  so that it can cancel the  $h$ -masks in the final sum by itself. Also, these users should not participate in the Aggregation phase as their  $h$ -masks are revealed. More specifically, the honest users who finish the Setup phase update their participants list, removing the users in their groups who fail to send them the shares of their  $r$ - and  $h$ -masks. This set can be different in different users' view if the adversary is malicious.

**Algorithm 4 SecAgg (MicroFedML<sub>2</sub>)**

This protocol uses the following algorithms defined in Section 3: a public key infrastructure, a Shamir's secret sharing scheme (SS.share, SS.recon, SS.expoShare, SS.expoRecon); a random oracle  $H(\cdot)$  which returns a random generator of  $\mathbb{Z}_p^*$  on a fresh input. It proceeds as follows:

**Input:** Every user  $i \in G_d$  holds its own signing key  $d_i^{SK}$  and every user  $j$ 's verification keys  $d_j^{PK}$ ,  $r_i, h_i$ , a list of users  $\mathcal{U}_i \subseteq G_d$  with  $r_{j,i}, h_{j,i}$  for every  $j \in \mathcal{U}_i$ . Moreover, for every iteration  $k$ , it also holds a secret input  $x_i^k$ . The server  $\mathcal{S}$  holds all inputs it receives in the Setup phase, and the list of users  $\mathcal{U}_S$  it outputs in the Setup phase.

**Output:** For each iteration  $k$ , if all users are honest and there are at least  $t$  users being always online in each group during iteration  $k$ , then at the end of iteration  $k$ , the server  $\mathcal{S}$  outputs  $\sum_{i \in \mathcal{O}^k} x_i^k$ , in which  $\mathcal{O}^k$  denotes a set of at least  $Bt$  users.

**Note:** For simplicity of exposition, we omit the superscript  $k$  of all variables when it can be easily inferred from the context.

1: **for** Iteration  $k = 1, 2, \dots$  **do**

**Round 1: Masked Input:**

2: User  $i$ : It masks the input by  $X_i = x_i + r_i + h_i$  and sends  $H(k)^{X_i}$  to the server.

3: Server  $\mathcal{S}$ : If it receives messages from less than  $t$  users from any group, abort. If it receives messages from a user not in  $\mathcal{U}_S$ , ignore the message. Otherwise, let  $\mathcal{O}$  denote the set of users  $i$  who successfully send the masked input to the server. For each group  $G_d$ , the server sends  $\mathcal{O}_d = \mathcal{O} \cap G_d$  to all users  $i \in \mathcal{O}_d$ .

**Round 2: Online Set Checking:**

4: User  $i$ : It checks  $|\mathcal{O}_d| \geq t$ , signs  $\mathcal{O}_d$  and sends back to the server as described in line 4 of Algorithm 2.

5: Server  $\mathcal{S}$ : It distributes the signatures from users  $G_d$  to  $\mathcal{O}_d$  as described in line 5 of Algorithm 2.

**Round 3: Mask Reconstruction:**

6: User  $i$ : it checks that it receives at least  $t$  valid signatures from the members of  $G_d$  on  $\mathcal{O}_d$ . If any signature is invalid, abort. Then it calculates  $\zeta_i = H(k)^{\sum_{j \in \mathcal{O}_d} r_{j,i} - \sum_{j \in \mathcal{U}_i \setminus \mathcal{O}_d} h_{j,i}}$  and sends  $\zeta_i$  to the server.

7: Server  $\mathcal{S}$ : If it receives  $\zeta_i$  from less than  $t$  users in any group  $G_d$ , abort. Otherwise, the server calculates

$$z = \log_{H(k)} \left( H(k)^{\sum_{i \in \mathcal{O}} X_i} / \prod_{d \in [B]} \text{SS.expoRecon}(\{\zeta_i\}_{i \in \mathcal{O}_d}, t) \right)$$

by brute force. Then it calculates  $\sum_{i \in \mathcal{O}} x_i = z + h_S$ , in which  $h_S$  is from the server's output in the Setup phase.

8: **end for**

## 5.2 Group Properties

To achieve security and correctness at the same time, there should not be too many corrupt or offline line nodes in each group. We show that if we choose the size  $N = n/B$  of each group and the threshold  $t$  appropriately, this requirement can be satisfied with overwhelming probability. We follow the same reasoning and calculation with the same assumptions as in Bell et al. (2020).

We first define the requirements as the following two good events.

**Definition 5.1** (Not too many corrupt members). *Let  $N, t$  be integers such that  $N < n$  and  $t \in (N/2, N)$ , and let  $\mathcal{C} \subset [n]$ . Let  $\mathbf{G} = (G_1, \dots, G_B)$  is a partition of  $[n]$  so that  $|G_d| = N$  for each  $d \in [B]$ . We define event  $E_1$  as*

$$E_1(\mathcal{C}, \mathbf{G}, N, t) = 1 \text{ iff } \forall d \in [B] : |G_d \cap \mathcal{C}| < 2t - N.$$

**Definition 5.2** (Enough shares are available). *Let  $N, t$  be integers such that  $N < n$  and  $t \in (N/2, N)$ , and let  $D \subset [n]$ . Let  $\mathbf{G} = (G_1, \dots, G_B)$  is a partition of  $[n]$  so that  $|G_d| = N$  for each  $d \in [B]$ . We define event  $E_2$  as*

$$E_2(D, \mathbf{G}, N, t) = 1 \text{ iff } \forall d \in [B] : |G_d \cap ([n] \setminus D)| \geq t.$$

We say a distribution of grouping is *nice grouping* if the above two events happen with overwhelming probability. In other words, with a nice grouping algorithm, each group has neither too many corrupt members nor too many offline members with 1 but negligible probability.

**Definition 5.3** (Nice Grouping). *Let  $N, \sigma, \eta$  be integers and let  $\gamma, \delta \in [0, 1]$ . Let  $\mathcal{C} \subset [n]$  and  $|\mathcal{C}| \leq \gamma n$ . Let  $\mathcal{D}$  be a distribution over pairs  $(\mathbf{G}, t)$ . We say that  $\mathcal{D}$  is  $(\sigma, \eta, \mathcal{C})$ -nice if, for all set  $D \subset [n]$  such that  $|D| \leq \delta n$ , we have that*

$$1. \Pr[E_1(\mathcal{C}, \mathbf{G}', N, t') = 1 \mid (\mathbf{G}', t') \leftarrow \mathcal{D}] > 1 - 2^{-\sigma},$$

$$2. \Pr[E_2(D, \mathbf{G}', N, t') = 1 \mid (\mathbf{G}', t') \leftarrow \mathcal{D}] > 1 - 2^{-\eta}.$$

**Lemma 5.4.** *Let  $\gamma, \delta \geq 0$  such that  $\gamma + 2\delta < 1$ . Then there exists a constant  $c$  making the following statement true for all sufficiently large  $n$ . Let  $N$  and  $t$  be such that*

$$N \geq c(1 + \log n + \eta + \sigma), \quad t = \lceil (3 + \gamma - 2\delta)N/4 \rceil.$$

*Let  $\mathcal{C} \subset [n]$ , such that  $|\mathcal{C}| \leq \gamma n$ , be the set of corrupt clients. Then for sufficiently large  $n$ , the distribution  $\mathcal{D}$  over pairs  $(G, t)$  implemented by uniformly randomly assigning all  $n$  users into  $n/N$  groups each of size  $N$  is  $(\sigma, \eta, \mathcal{C})$ -nice.*

Due to page limit, we defer the proof to Appendix C.1.



### 5.3 Correctness with Dropouts

Similar as MicroFedML<sub>1</sub>, the protocol guarantees correctness when there are enough number of users online in each group so that the server can reconstruct the sum of  $r_i$  for online users  $i$  and the sum of  $h_j$  for offline users  $j$ .

**Theorem 5.5** (Correctness with dropouts). *Let  $\gamma, \delta$  be two parameters such that  $\gamma < 1/3$ ,  $\gamma + 2\delta < 1$ , and  $\sigma$  and  $\eta$  be two security parameters. The protocol  $\Pi$  be an instantiation of Algorithm 3 and Algorithm 4 running with a server  $S$  and  $n$  users guarantees correctness with  $\delta$  offline rate with probability  $1 - K \cdot 2^{-\eta}$ , when the grouping algorithm is  $(\sigma, \eta, \mathcal{C})$ -nice for  $\mathcal{C} \subset \mathcal{U}$  with  $|\mathcal{C}| < \gamma|\mathcal{U}|$ .*

### 5.4 Privacy against Semi-Honest and Malicious Adversaries

It is easy to see that same as MicroFedML<sub>1</sub>, this protocol provides perfect privacy when the server is honest, as the joint view of any set of users does not depend on the input value of other users. We also omit the privacy proof against semi-honest adversary as it is a simplified version of the more complex malicious proof without any part related to public-key infrastructure. With malicious adversaries who control both the server and corrupt clients, we provide the privacy described in Theorem 5.6. We defer the proof to Appendix C due to the page limit.

**Theorem 5.6** (Privacy against Malicious Adversary). *Let  $\gamma$  and  $\delta$  be two parameters such that  $\gamma < 1/3$ ,  $\gamma + 2\delta < 1$ , and  $\sigma$  and  $\eta$  be two security parameters. The protocol  $\Pi$  be an instance of Algorithm 3 and Algorithm 4 guarantees privacy against  $\gamma$ -fraction of malicious adversary with  $\delta$  offline rate with probability  $1 - 2^{-\sigma}$  when the grouping is  $(\sigma, \eta, \mathcal{C})$ -nice. The Setup phase of the protocol runs in 5 rounds with  $O(n)$  communication complexity per user and the Aggregation phase runs in 3 rounds with  $O(n)$  communication complexity per user.*

## 6 Improvement with Class Group

To overcome the limitation that the discrete logarithm, we propose an alternative version of our protocol based on class group primitive as defined in Definition A.1. Let  $\mathbb{G}$  be the group with the DDH assumption and  $\mathbb{F}$  be its subgroup in which discrete logarithm is easy, the core idea is to leverage the subgroup  $\mathbb{F}$  where the discrete logarithm is easy to hide secret inputs while still using the group  $\mathbb{G}$  to hide the masks. Once the sum of masks in the exponent of the generator of the large group is reconstructed and removed, the server only needs to calculate the easy discrete logarithm in the subgroup to recover the sum of secret inputs. One important thing is that the adversary should not know the order of  $\mathbb{G}$ . Otherwise, the DDH assumption no longer holds and the corrupt server can recover information about shares of masks in the exponent. Thus, the masks and the shares cannot be sampled from a finite field but from integers within a bounded interval, as the server who performs reconstruction in the exponent of the generator of  $\mathbb{G}$  does not know the order of group  $\mathbb{G}$ . Therefore, we use the modified version of Shamir secret sharing which works on a bounded integer interval as introduced in Appendix A.1. As the modified version of Shamir secret sharing on integer interval scales the shares of masks up by  $n!$  in which  $n$  denotes the number of parties the secret is shared with, this method is more suitable for a relatively smaller set of users or the group version in which each secret is only shared within a small group of users.

To be more specific, we need to make changes to the following part of the protocol (we use MicroFedML<sub>2</sub> here but it also applies to MicroFedML<sub>1</sub>): 1) the Shamir secret sharing scheme now works on a bounded integer interval instead of a finite field; 2) in the first round of the Aggregation phase, each user  $i$  sends  $g_k^{r_i+h_i} \cdot f_i^x$  instead of  $H(k)^{r_i+h_i+x_i}$  to the server, where  $g_k$  and  $f_k$  are the generators of a group  $\mathbb{G}_k$  with subgroup  $\mathbb{F}_k$  satisfying Definition A.1 which are freshly and honestly chosen in advance for each iteration; 3) in the last round of the Aggregation phase, each user sends  $\zeta_i = g_k^{\sum_{j \in \mathcal{O}_d} r_{j,i} - \sum_{j \in \mathcal{U}_i \setminus \mathcal{O}_d} h_{j,i}}$  to the server. The server reconstructs  $R_{\mathcal{O}_d} = \text{SS.expoRecon}(\{\zeta_j, j\}_{j \in \mathcal{O}_d}, t)$ , calculates  $R_{\mathcal{O}} = \prod_d R_{\mathcal{O}_d} \cdot g_k^{h_s}$ , and calculates

$$\sum_{i \in \mathcal{O}} x_i \leftarrow \text{cl.solve}(B, p, g, f, G, F, g_k^{\sum_{i \in \mathcal{O}} r_i} f^{\sum_{i \in \mathcal{O}} x_i} / R_{\mathcal{O}}).$$

A full version will be included in the full version of the paper.

	Communication cost		#Round
	User	Server	
MicroFedML <sub>1</sub> (Setup)	$O(n)$	$O(n^2)$	3
MicroFedML <sub>2</sub> (Setup)	$O(\log n)$	$O(n \log n)$	5

Table 2: Communication overhead of the Setup phase of aggregation of MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub> guaranteeing privacy against malicious adversaries in which  $n$  denotes the total number of users. Note that BIK+17 and BBG+20 have  $O(n^2)$  and  $O(n \log n)$  communication cost, respectively, on the server side but they incur this cost in every iteration.

## 7 Performance

### 7.1 Asymptotic Performance of MicroFedML<sub>1</sub>

#### 7.1.1 Semi-Honest Protocol

**Communication** In the Setup phase, each user sends one public encryption key ( $O(1)$ ) to the server and receives public encryption keys of all other users ( $O(n)$ ), then it sends encrypted shares for all other users of its random mask chosen from  $\mathbb{Z}_q$  to the server and receives one encrypted share of mask of each other user ( $O(nR)$ ). This results in  $O(nR)$  communication cost for each user. As the message the server sends to each user is of the same size, the communication cost for the server is  $O(n^2R)$ .

In the first round of the Aggregation phase, each user sends an element  $H(k)^{x_i+r_i} \in \mathbb{Z}_p^*$  to the server ( $O(R)$ ) and receives the indicator of the online set  $\mathcal{O}$  ( $n$  bits), which results in  $O(R + n)$  communication cost. In the second round, each user sends  $H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}}$  which is also an element in  $\mathbb{Z}_p^*$  to the server, which results in  $O(R)$  communication cost. Thus, the total communication cost of each user is  $O(R + n)$ . As the size of message between the server and each user is the same, the communication cost of the server is  $O(Rn + n^2)$ .

**Computation** We discuss the computation cost of each user first. In the Setup phase, each user  $i$  needs to 1) generate a pair of encryption keys  $pk_i$  and  $sk_i$ , 2) run the key exchange algorithm to obtain  $ek_{i,j}$  for all other users  $j$ , 3) secret shares  $r_i$  among all users, 4) encrypt share  $r_{i,j}$  for each other user  $j$  with  $ek_{i,j}$ , 5) decrypt the cipher text  $c_{j,i}$  for each other user  $j$  with  $ek_{i,j}$ . Thus, the computation cost of each user in the Setup phase is  $O(n)$ . In the Aggregation phase, the computation cost of each user consists of calculating  $H(k)^{x_i+r_i}$  and calculating  $H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}}$ , which is  $O(n)$  in total.

Then we analyze the computation cost of the server. In the Setup phase of the semi-honest protocol, the server only forwards the messages for users. In the Aggregation phase, the server needs to multiply all the masked inputs it receives, reconstruct the sum of the random masks of all online users in the exponent, and calculate the discrete log to get the final result in the second round of the Aggregation phase. Thus, the computation cost of the server is  $O(n + R)$  in Aggregation phase of the semi-honest protocol.

#### 7.1.2 Protocol guaranteeing privacy against malicious adversary

In the protocol that protects privacy against malicious adversaries, in addition to the communication cost listed above, each user also sends a signature and receives signatures of all other users in the first round of the Setup phase and the second round of the Aggregation phase, which results in  $O(n)$  communication cost. Thus, the asymptotic communication cost does not change.

Regarding the computation cost, each user needs to additionally sign the public key  $ek$  in the Setup phase and the online set in the Aggregation phase and also verify all other users' signatures, which involves  $O(n)$  computation cost. The server also needs to verify all signatures from the users. Thus, the asymptotic computation cost is the same as the cost of semi-honest protocol for both users and the server.

### 7.2 Asymptotic Performance of MicroFedML<sub>2</sub>

#### 7.2.1 Semi-Honest Protocol

**Communication** In the Setup phase, each user  $i \in G_d$  needs to 1) send its public encryption key  $pk_i$  and two public masking keys  $pk_i^1$  and  $pk_i^{-1}$  to the server and receive the public keys of the group members of its own group  $G_d$  and two neighboring groups  $G_{d+1}$  and  $G_{d-1}$ , 2) send the encrypted shares of  $sk_i^1$  and  $sk_i^{-1}$  to all group members of  $G_{d+1}$



and  $G_{d-1}$  and receive encrypted shares from them, 3) send the encrypted shares of  $r_i$  and  $h_i$  to the group members in  $G_d$  and receive encrypted shares from them, 4) receive the list of offline users in  $G_{d+1}$  and  $G_{d-1}$  and send the shares of secret masking keys of those offline users to the server. When the size of each user group is set as  $O(\log n)$ , the communication cost for each user in the Setup phase is  $O(R \log n)$ . As messages between the server and each user is the same, the communication cost of the Setup phase for the server is  $O(nR \log n)$ .

The communication cost of the Aggregation phase for both the user and the server is the same as the non-grouping version except that now each user only needs to know the online set of its own group. Thus, assuming the group size is  $O(\log n)$ , the communication cost of one iteration of the Aggregation is  $O(R + \log n)$  for each user and  $O(nR + n \log n)$  for the server.

**Computation** We discuss the computation cost of each user first. In the Setup phase, each user  $i \in G_d$  needs to 1) generate three key pairs, 2) run key exchange algorithm to get the symmetric encryption key  $ek_{i,j}$  for group members  $j$  of  $G_d$  and  $mk_{i,j}$  for group members of  $G_{d+1}$  and  $G_{d-1}$ , secret share its private masking keys among the group members of two neighboring groups  $G_{d+1}$  and  $G_{d-1}$ , 3) decrypt the shares of private masking keys received from the two neighboring groups pick the random mask  $r_i$  and calculate the mutual mask  $h_i$ , secret share both the masks among group members of  $G_d$ , and encrypt each share, 4) decrypt the shares of the masks received from group members of  $G_d$ . Thus, the computation cost of each user of the Setup phase is  $O(\log n)$ .

In each iteration of the Aggregation phase, each user  $i$  needs to compute the masked input  $H(k)_i^X$  and the aggregated shares  $H(k)^{\sum_{j \in \mathcal{O}_d} r_{j,i} - \sum_{j \in G_d \setminus \mathcal{O}_d} h_{j,i}}$ , which involves  $O(\log n)$  computation.

Now, we analyze the computation cost of the server. In the Setup phase, excepting forwarding messages for users, the server also needs to reconstruct  $mk_i^{-1}$  and  $mk_i^1$  and calculate  $h_i$  for the users  $i$  who are online in the second round but offline in the third round. Assuming there are  $\delta n$  offline users in which  $\delta$  is a constant parameter, the server needs to do  $O(n^2)$  computation.

In the Aggregation phase, the server needs to reconstruct the sum of masks in the exponent, multiply the results of all groups together, and calculate the discrete log to get the final result. Assuming each group is of size  $O(\log n)$ , the computation cost of the server is  $O(\log n + \frac{n}{\log n} + R)$ .

### 7.2.2 Protocol Guaranteeing Privacy against Malicious Adversary

**Communication** In addition to the communication listed in the semi-honest case, in this version, each user  $i \in G_d$  needs to send and receive signatures with the public keys and agree on two offline lists of  $G_{d+1}$  and  $G_{d-1}$  respectively before it sends the shares of the secret masking keys of the offline users in these two groups to the server in the Setup phase, and agree on the online set  $\mathcal{O}_i$  by sending and receiving signatures on the set. These introduces  $O(\log n)$  additional communication cost to both the Setup phase and the Aggregation phase for each user (which means  $O(n \log n)$  additional cost for the server), which does not change the asymptotic communication cost of the users and the server.

**Computation** Compared to the semi-honest version of protocol, the user needs to signs the public keys and verify signatures from the members of its own group and two neighboring groups, and the server also needs to verify the signatures it receives. This adds  $O(\log n)$  computation to each user and  $O(n)$  computation to the server, which does not change the asymptotic computation cost for both the users and the server.

## 7.3 Concrete Performance

To measure the concrete performance, we provide end-to-end implementations both of our protocols, MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub>, as well as three benchmark protocols, BIK+17, BBG+20 and Flamingo with ABIDES Byrd et al. (2019) a discrete event simulation framework with modification to enable simulation of federated learning protocol, in Python language. More information about the framework we use is included in Appendix F. In all implementations, we assume semi-honest setting, thus we omit the marked parts of the protocols that is only needed in the malicious setting.

The experiments are run on an AWS EC2 r5.xlarge instance equipped with 4 3.1 GHz Intel Xeon Platinum 8000 series processors CPUs and 32GB memory. We are using large machine instances so that we can simulate a large number of parties. Each user and the server is single-threaded. For each protocol, we run 10 iterations of aggregation and take the average of the running time and bandwidth consumption.

**Quick comparison.** In the first two graphs in Figure 5, we compare the local computation time of four protocols with the length of the result fixed to 20 bits and group/neighbor size fixed to 200 for MicroFedML<sub>2</sub> and BBG+20. As shown in the graph, the computation time of MicroFedML<sub>1</sub> is about 100 times shorter than BBG+20 when the total number of users is 500, and the computation time of MicroFedML<sub>2</sub> is about 20 times faster than BBG+20 when the total number

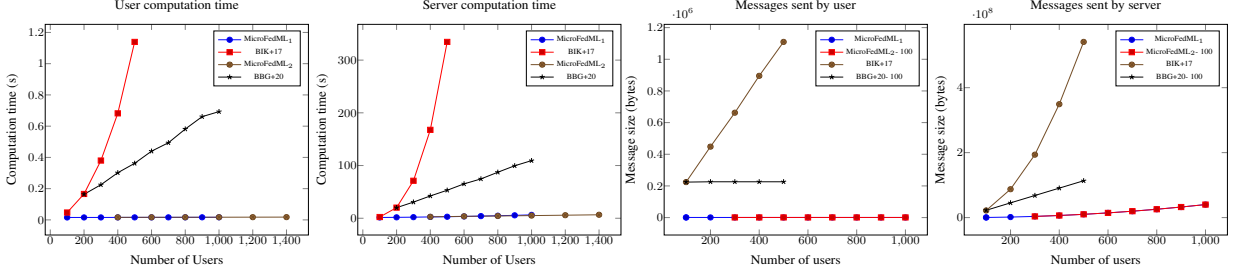


Figure 5: Wall-clock local computation time of one iteration of the Aggregation phase of a user and the server as the number of user increases. The length of the sum of inputs is fixed to  $\ell = 20$  bits in different lines, i.e., the input of each user is in the range  $[2^\ell/n]$  when the total number of users is  $n$ . For the protocol MicroFedML<sub>2</sub> and BBG+20, the group size / neighbor size is set to 200.

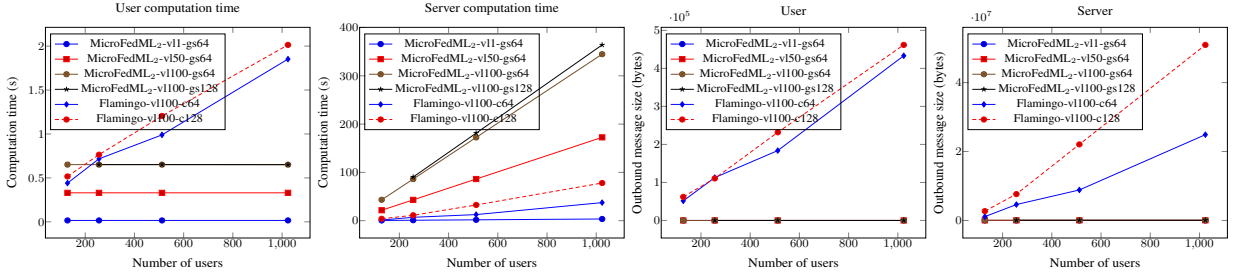


Figure 6: The left two graphs show the wall-clock local computation time of one iteration of the Aggregation phase of a user and the server as the number of users increases. The right two graphs are the outbound bandwidth cost (bytes) on the user and the server side when the total number of users grows. In the legend, “v1” denotes the length of input vector, “gs” denotes group size for MicroFedML<sub>2</sub>, “c” denotes the committee size for Flamingo.

of users is 1000. In the two graphs on the right side, we compare the bandwidth cost per iteration of different protocols, with the length of the result fixed to 20 bits and the group/neighbor size is fixed to 100. The size of outgoing messages of each user of MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub> are almost the same, which is about 1000 times smaller than BIK+17 and about 200 times smaller than BBG+20 when the total number of users is 500. The size of incoming messages from the server per user of MicroFedML<sub>1</sub> is also almost the same as MicroFedML<sub>2</sub>, which is about 50 times smaller than BIK+17 and 10 times smaller than BBG+20 when the total number of users is 500. The improvement of computation time and bandwidth cost will be larger when the total number of users increases.

We stress that all experiments above and the ones in the detailed comparison are run with brute force discrete logarithm. With class group, the server computation time can be further significantly reduced. For example, computing discrete logarithms using brute force on a 20-bit range takes 1.14 seconds, while it takes only 673 nanoseconds to compute on a 32-bit range with class group library Bouvier et al. (2022) in the same environment.

We also give a performance comparison between our protocol instantiated *with the class group* and Flamingo Ma et al. (2023) in Figure 6. As shown in the graph, for a single input, our protocol allows shorter computation time and lower communication costs. When the vector length of inputs is larger, our user and server computation time grows in proportion as the number of exponentiations by each user and the number of reconstructions on exponent by the server grow in proportion to the vector length. This is because our protocol does not utilize a PRG to extend a random secret to a long masking vector as in BIK+17, BBG+20 or Flamingo. On the other hand, the computation time of Flamingo is not affected by the input vector length significantly. However, the communication cost of Flamingo also grows with the vector length. On the other hand, while we provide worse server computation time when the vector length grows, it is of lesser importance as servers typically run on much more powerful machines and the server computation of share reconstructions can also be easily parallelized. Last but not least, our protocol provides better user computation time and bandwidth cost when the total number of participants is large. For example, with 1024 users, neighbor/committee size 128, vector length 100, the user in our protocol is 3 times faster and has 10 times lower bandwidth.

**Detailed comparison.** Figure 7 shows how the local computation time of each iteration of the Aggregation phase (y-axis) of each user and the server changes as the total number of users (x-axis) grows. As shown in the graph, the running time of MicroFedML<sub>1</sub> is not affected significantly by the number of users but more affected by the size of the sum of inputs. On the contrary, the performance of BIK+17 is impacted by the total number of users and does not

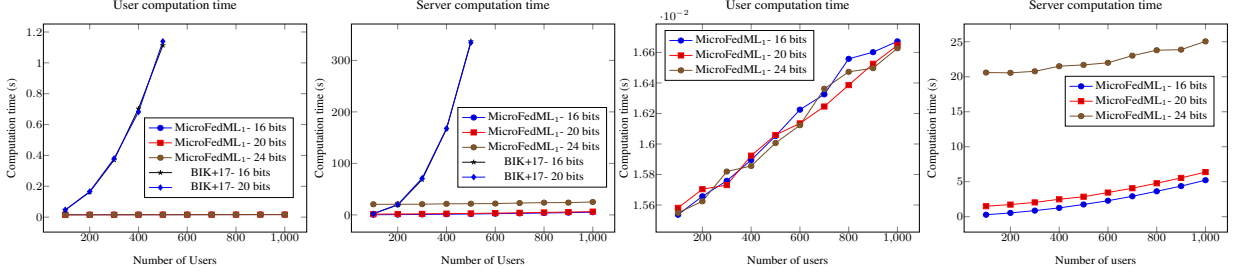


Figure 7: The left two graphs are the wall-clock local computation time of one iteration of the Aggregation phase of a user and the server as the number of user increases. The length of the sum of inputs is set to  $\ell = 16, 20, 24$  bits in different lines, i.e., the input of each user is in the range  $[2^{\ell}/n]$  when the total number of users is  $n$ . The right two graphs are zoom-in which only includes MicroFedML<sub>1</sub> protocol.

change with different input sizes. This is because of the different methods the two protocols use to obtain the result. In BIK+17, in all scenarios we are using the same finite field, which means the size of each share (which is a field element) and the time it requires to share and reconstruct the secret keeps the same in these scenarios. On the other hand, each user needs to share two secrets with all other users and the server needs to reconstruct a secret for each user with shares from a linear fraction of all users in every iteration, thus the running time of both the user and the server increases as the number of users increases. On the contrary, in the Aggregation phase of MicroFedML<sub>1</sub>, each user only calculates and sends one field element to the server in each round and the server only needs to run the reconstruction in the exponent once no matter how many users are participating. Thus, the total running time does not grow obviously with the total number of users when compared with the benchmark protocol. As the server calculates the aggregated result with discrete log, the running time increases when the range of the result enlarges. This relevance does not exist when using class groups.

In Figure 8 we show the local computation time of online users (who stays online in the whole iteration) and the server in one iteration of MicroFedML<sub>1</sub> and BIK+17 with a different fraction of users dropping out. In each iteration, a  $\delta$ -fraction of users are randomly selected from all users and stay online before they send the masked inputs to the server (which happens in the first round of MicroFedML<sub>1</sub> and in the second round of BIK+17) then stay silent for the rest part of the iteration. The size of the sum of inputs is fixed to 20 bits. Different lines show the computation time of one iteration of the online users and the server in the cases with different  $\delta$ . As shown in the left graph, the dropout rate does not affect the computation time of online users significantly in BIK+17, as each online user  $i$  needs to send one share of secret for each other user  $j$  to the server no matter whether user  $j$  is online or not. The second graph shows that the computation time of the server in BBG+20 decreases as more users drop out. This is different from the experiment result reported in Bonawitz et al. (2017), as in Bonawitz et al. (2017) the server needs to extend the symmetric masking key between an offline user  $i$  and all other users  $j$  to a long vector using a pseudorandom generator (PRG) to cover the whole input vector which is costly, while in this work we assume each input is a single element and the server does not apply PRG over the symmetric masking key, which makes impact of the dropout rate less severe. Moreover, we implement the reconstruction of Shamir's secret sharing naively uses all shares received, which means the more users drop out, the fewer shares received by the server in the next round and the less time it takes to run the reconstruction algorithm. In zoom-in graphs of Figure 8, we can see a higher value of  $\delta$  leads to shorter computation time for both users and the server. This is because of the same reason as mentioned above — the fewer users are online, the fewer shares need to be included when computing the sum of shares on the user's side, and the fewer shares are included in the reconstruction in the exponent on the server's side.

Figure 9 shows the local computation time of each party of each iteration of MicroFedML<sub>2</sub> and BBG+20 for different neighbor sizes and total number of users. By neighbor size, we mean the size of one group in our group protocol and the number of neighbors each user has in BBG+20. Note that in real-world applications, the neighbor size should be chosen based on the total number of users and the assumed fraction of corrupt and dropout users. For example, when the total number of users is 1000, the fraction of corrupted users is 0.33, and the fraction of offline users is 0.05, the group size can be chosen as about 80, while to tolerate both 0.33 fraction of corrupt users and 0.33 fraction of offline users, the group size should be chosen as 300 in the semi-honest scenario. We refer the readers to Section C.1 and Section 3.5 of Bell et al. (2020) for the detailed discussion about how to choose the group size or the number of neighbors. In the experiment, we use fixed group size just for efficiency analysis purposes. As shown in the running result, sharing information with only a small set of neighbors significantly improves the performance of the benchmark protocol BBG+20, as the number of users included in secret sharing and reconstruction is a major factor of computation overhead. On the contrary, the improvement brought by the grouping is not significant in MicroFedML<sub>2</sub> as the only

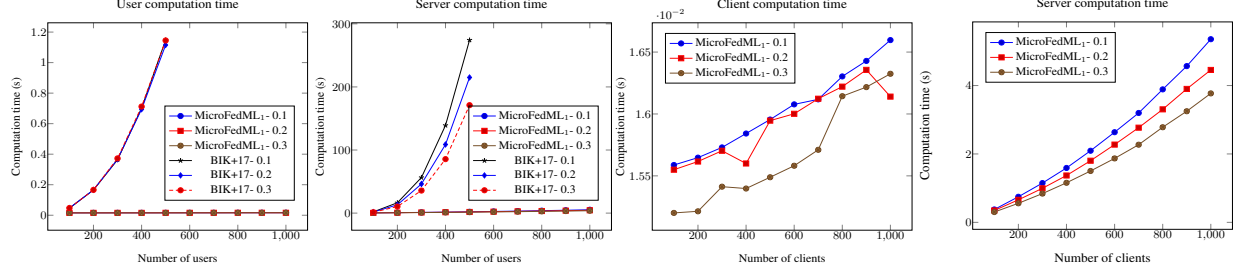


Figure 8: The left two graphs show wall-clock computation time of one iteration of the Aggregation phase of a user and the server, as the number of users increases. Different lines show the running time with different offline rates. The length of the output is fixed to 16 bits. The right two graphs are zoom-in with only MicroFedML<sub>1</sub> included.

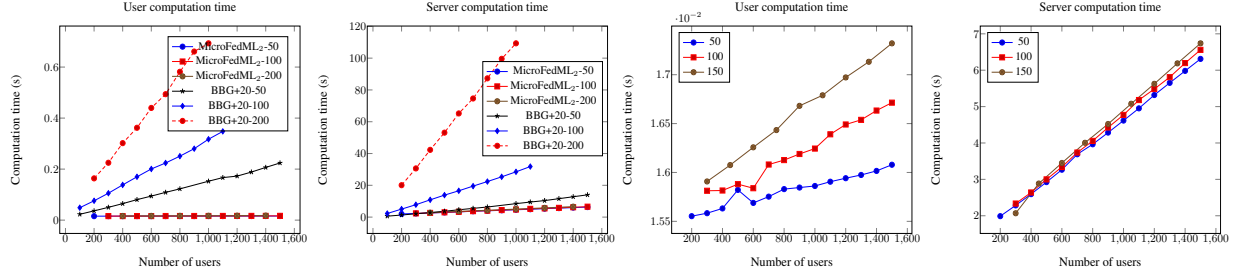


Figure 9: The left two graphs show the Wall-clock local computation time (y-axis) of one iteration of MicroFedML<sub>2</sub> and BBG+20 for a user and the server as the number of users (x-axis) increases. Different lines shows the running result with different group sizes. The length of the output is fixed to 20 bits. The right two graphs are zoom-in which only includes MicroFedML<sub>2</sub>.

two things affected by the number of neighbors in the Aggregation phase are the size of the online set in each group and the number of shares the server uses to do the reconstruction. Both these two parts compose only a very small fraction of the total running time. The computation time of users varies more than the computation time of the server when the group size is different, as in each iteration, the user needs to sum up the shares of the masks of all online group members the running time of which depends on the group size.

In Figure 10, we report the total computation time and bandwidth cost of each user and the server in the Setup phase of MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub> with different group sizes. The graph on the left shows that the computation time of each user in the Setup phase of the basic protocol grows with the total number of users, while in the group protocol the computation time grows when the group size grows. In MicroFedML<sub>1</sub>, the bandwidth cost on the user side grows linearly with the growth of the total number of users as each user needs to send one encrypted share to every other user, which also results in quadratic bandwidth growth on the server's side. In MicroFedML<sub>2</sub>, when the group size is fixed, the bandwidth cost on the users' side does not increase with the total number of users, while the bandwidth cost on the server's side increases linearly as the number of groups increases.

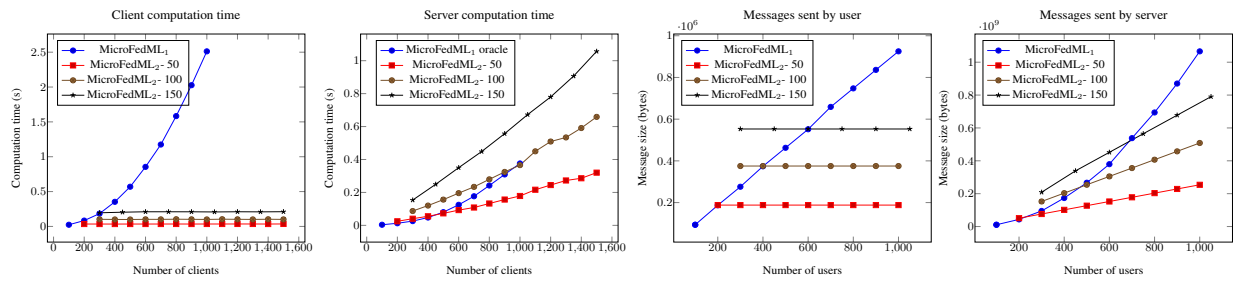


Figure 10: Wall-clock local computation time and outbound bandwidth cost (bytes) of the **Setup** phase for a user and the server, as the number of users increases. Different lines shows the running results for MicroFedML<sub>1</sub> and MicroFedML<sub>2</sub> with different group size.

We also implement a federated learning protocol with MicroFedML<sub>1</sub> on the adult census income dataset Dua and Graff (2017) which provides 14 input features such as age, marital status, and occupation, that can be used to predict a categorical output variable identifying whether (True) or not (False) an individual earns over 50K USD per year. We run a logistic regression algorithm on the preprocessed version used by Byrd et al. Byrd et al. (2022) which is a cleaned version with one constant intercept feature added based on a preprocessed version of the dataset from Jayaraman et al. Jayaraman et al. (2018) The preprocessed dataset contains 105 features and 45,222 records, about 25% (11,208) of which are positive. The dataset is loaded once at the beginning of the protocol execution and randomly split into training set (75%) and testing set (25%). At the beginning of each training iteration, a user randomly selects 200 records from the training data as its local training data and test the model accuracy with the common test set. We run both the plain federated learning in which every user simply sends its update in plain text to the server and the version with MicroFedML<sub>1</sub> in which the model updates are aggregated with the secure aggregation protocol for 5 iterations of aggregation, each with 50 local training iterations. We assess accuracy using the Matthews Correlation Coefficient (MCC) Matthews (1975), a contingency method of calculating the Pearson product-moment correlation coefficient (with the same interpretation), that is appropriate for imbalanced (3:1) classification problems in our case. The accuracy of the models’ output in these two scenarios are both distributed close around 0.81, showing that using the secure aggregation protocol does not affect the accuracy of the model learned.

## 8 Conclusion and Future Work

In this work, we propose a new construction of multi-iteration secure aggregation protocol that has better round complexity while keeping the same asymptotic communication cost. We provide correctness and privacy proofs in semi-honest and malicious settings, respectively.

As the next step, we are interested in using our secure aggregation protocol as a building tool for the more complex setting of vertical federated learning where the feature space for the machine learning model across all devices is different. We are also interested in reducing the communication cost further with compressing techniques when the model update is sparse.

## References

- Damiano Abram, Ivan Damgård, Claudio Orlandi, and Peter Scholl. 2022. An algebraic framework for silent preprocessing with trustless setup and active security. In *Advances in Cryptology—CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part IV*. Springer, 421–452.
- Alham Fikri Aji and Kenneth Heafield. 2017. Sparse Communication for Distributed Gradient Descent. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing* (2017), 440–445. <https://doi.org/10.18653/v1/D17-1045> arXiv: 1704.05021.
- Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. *Advances in neural information processing systems* 30 (2017).
- Mohammad Mohammadi Amiri and Deniz Gunduz. 2020a. Federated Learning over Wireless Fading Channels. *arXiv:1907.09769 [cs, math]* (Feb. 2020). <http://arxiv.org/abs/1907.09769> arXiv: 1907.09769.
- Mohammad Mohammadi Amiri and Deniz Gunduz. 2020b. Machine Learning at the Wireless Edge: Distributed Stochastic Gradient Descent Over-the-Air. *IEEE Transactions on Signal Processing* 68 (2020), 2155–2169. <https://doi.org/10.1109/TSP.2020.2981904> arXiv: 1901.00844.
- Debraj Basu, Deepesh Data, Can Karakus, and Suhas N. Diggavi. 2020. Qsparse-Local-SGD: Distributed SGD With Quantization, Sparsification, and Local Computations. *IEEE Journal on Selected Areas in Information Theory* 1, 1 (May 2020), 217–226. <https://doi.org/10.1109/JSAIT.2020.2985917>
- James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. 2022. ACORN: Input Validation for Secure Aggregation. *Cryptology ePrint Archive* (2022).
- James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. 2020. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1253–1269.
- Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1175–1191. <https://doi.org/10.1145/3133956.3133982>

- Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In *Advances in Cryptology—CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I* 39. Springer, 561–586.
- Cyril Bouvier, Guilhem Castagnos, Laurent Imbert, and Fabien Laguillaumie. 2022. I want to ride my BICYCL: BICYCL Implements CryptographY in CLass groups. (2022).
- Lennart Braun, Ivan Damgård, and Claudio Orlandi. 2022. Secure Multiparty Computation from Threshold Encryption based on Class Groups. *Cryptology ePrint Archive* (2022).
- David Byrd, Maria Hybinette, and Tucker Hybinette Balch. 2019. ABIDES: Towards High-Fidelity Market Simulation for AI Research. *CoRR* abs/1904.12066 (2019). arXiv:1904.12066 <http://arxiv.org/abs/1904.12066>
- David Byrd, Vaikkunth Mugunthan, Antigoni Polychroniadou, and Tucker Hybinette Balch. 2022. Collusion Resistant Federated Learning with Oblivious Distributed Differential Privacy. *arXiv preprint arXiv:2202.09897* (2022).
- Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2019. Two-party ECDSA from hash proof systems and efficient instantiations. In *Advances in Cryptology—CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III* 39. Springer, 191–221.
- Guilhem Castagnos and Fabien Laguillaumie. 2015. Linearly Homomorphic Encryption from DDH. *Cryptology ePrint Archive*, Paper 2015/047. <https://eprint.iacr.org/2015/047> <https://eprint.iacr.org/2015/047>.
- Guilhem Castagnos, Fabien Laguillaumie, and Ida Tucker. 2018. Practical fully secure unrestricted inner product functional encryption modulo  $p$ . In *Advances in Cryptology—ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part II*. Springer, 733–764.
- Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2018. Adacom: Adaptive residual gradient compression for data-parallel distributed training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
- Mingzhe Chen, Nir Shlezinger, H. Vincent Poor, Yonina C. Eldar, and Shuguang Cui. 2021. Communication-efficient federated learning. *Proceedings of the National Academy of Sciences* 118, 17 (April 2021), e2024789118. <https://doi.org/10.1073/pnas.2024789118>
- Laizhong Cui, Xiaoxin Su, Yipeng Zhou, and Yi Pan. 2021. Slashing Communication Traffic in Federated Learning by Transmitting Clustered Model Updates. *IEEE Journal on Selected Areas in Communications* (2021), 1–1. <https://doi.org/10.1109/JSAC.2021.3087262>
- Yi Deng, Shunli Ma, Xinxuan Zhang, Hailong Wang, Xuyang Song, and Xiang Xie. 2021. Promise  $\Sigma$ -protocol: How to construct efficient threshold ECDSA from encryptions based on class groups. In *Advances in Cryptology—ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part IV*. Springer, 557–586.
- Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. <http://archive.ics.uci.edu/ml>
- Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Helen Möllering, Thien Duc Nguyen, Phillip Rieger, Ahmad-Reza Sadeghi, Thomas Schneider, Hossein Yalame, et al. 2021. SAFElearn: Secure aggregation for private FEderated learning. In *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 56–62.
- Farzin Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Viveck Cadambe. 2019. Trading redundancy for communication: Speeding up distributed SGD for non-convex optimization. In *International Conference on Machine Learning*. PMLR, 2545–2554.
- Samuel Horvath, Chen-Yu Ho, Ludovit Horvath, Atal Narayan Sahu, Marco Canini, and Peter Richtarik. 2020. Natural Compression for Distributed Deep Learning. *arXiv:1905.10988 [cs, math, stat]* (Feb. 2020). <http://arxiv.org/abs/1905.10988> arXiv: 1905.10988.
- Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. 2018. Distributed learning without distress: Privacy-preserving empirical risk minimization. *Advances in Neural Information Processing Systems* 31 (2018).
- Aniket Kate, Easwar Vivek Mangipudi, Pratyay Mukherjee, Hamza Saleem, and Sri Aravinda Krishnan Thyagarajan. 2023. Non-interactive VSS using Class Groups and Application to DKG. *Cryptology ePrint Archive* (2023).
- Anders Krogh and John Hertz. 1991. A simple weight decay can improve generalization. *Advances in neural information processing systems* 4 (1991).
- Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. 2020. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv:1712.01887 [cs, stat]* (June 2020). <http://arxiv.org/abs/1712.01887> arXiv: 1712.01887.

- Yang Liu, Zhuo Ma, Ximeng Liu, Siqi Ma, Surya Nepal, Robert H Deng, and Kui Ren. 2020. Boosting Privately: Federated Extreme Gradient Boosting for Mobile Crowdsensing. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1–11.
- Zizhen Liu, Si Chen, Jing Ye, Junfeng Fan, Huawei Li, and Xiaowei Li. 2022a. SASH: Efficient secure aggregation based on SHPRG for federated learning. In *Uncertainty in Artificial Intelligence*. PMLR, 1243–1252.
- Ziyao Liu, Jiale Guo, Wenzhuo Yang, Jiani Fan, Kwok-Yan Lam, and Jun Zhao. 2022b. Privacy-preserving aggregation in federated learning: A survey. *IEEE Transactions on Big Data* (2022).
- Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. 2023. Flamingo: Multi-Round Single-Server Secure Aggregation with Applications to Private Federated Learning. 2023 IEEE Symposium on Security and Privacy (SP).
- Amirhossein Malekijoo, Mohammad Javad Fadaeieslam, Hanieh Malekijou, Morteza Homayounfar, Farshid Alizadeh-Shabdiz, and Reza Rawassizadeh. 2021. FEDZIP: A Compression Framework for Communication-Efficient Federated Learning. *arXiv:2102.01593 [cs]* (Feb. 2021). <http://arxiv.org/abs/2102.01593> arXiv: 2102.01593.
- Brian W Matthews. 1975. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405, 2 (1975), 442–451.
- Milad Nasr, Reza Shokri, and Amir Houmansadr. 2019. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *2019 IEEE symposium on security and privacy (SP)*. IEEE, 739–753.
- Milad Khademi Nori, Sangseok Yun, and Il-Min Kim. 2021. Fast Federated Learning by Balancing Communication Trade-Offs. *IEEE Transactions on Communications* (2021), 1–1. <https://doi.org/10.1109/TCOMM.2021.3083316>
- Felix Sattler, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. 2020. Robust and Communication-Efficient Federated Learning From Non-i.i.d. Data. *IEEE Transactions on Neural Networks and Learning Systems* 31, 9 (Sept. 2020), 3400–3413. <https://doi.org/10.1109/TNNLS.2019.2944481>
- Felix Sattler, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. 2018. Sparse Binary Compression: Towards Distributed Deep Learning with minimal Communication. *arXiv:1805.08768 [cs, stat]* (May 2018). <http://arxiv.org/abs/1805.08768> arXiv: 1805.08768.
- Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. <https://doi.org/10.1145/359168.359176>
- Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. 2019. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM workshop on artificial intelligence and security*. 1–11.
- Twan Van Laarhoven. 2017. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350* (2017).
- Hongyi Wang, Scott Sievert, Zachary Charles, Shengchao Liu, Stephen Wright, and Dimitris Papailiopoulos. 2018. ATOMO: Communication-efficient Learning via Atomic Sparsification. *arXiv:1806.04090 [cs, stat]* (Nov. 2018). <http://arxiv.org/abs/1806.04090> arXiv: 1806.04090.
- Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. *arXiv:1705.07878 [cs]* (Dec. 2017). <http://arxiv.org/abs/1705.07878> arXiv: 1705.07878.
- Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. 2018. Error compensated quantized SGD and its applications to large-scale distributed optimization. In *International Conference on Machine Learning*. PMLR, 5325–5333.
- Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. 2019. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*. 13–23.
- Chien-Sheng Yang, Jinhyun So, Chaoyang He, Songze Li, Qian Yu, and Salman Avestimehr. 2021b. LightSecAgg: Rethinking Secure Aggregation in Federated Learning. *arXiv preprint arXiv:2109.14236* (2021).
- Haibo Yang, Jia Liu, and Elizabeth S. Bentley. 2021a. CFedAvg: Achieving Efficient Communication and Fast Convergence in Non-IID Federated Learning. *arXiv:2106.07155 [cs]* (June 2021). <http://arxiv.org/abs/2106.07155> arXiv: 2106.07155.
- Tsz Hon Yuen, Handong Cui, and Xiang Xie. 2021. Compact zero-knowledge proofs for threshold ECDSA with trustless setup. In *Public-Key Cryptography–PKC 2021: 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10–13, 2021, Proceedings, Part I*. Springer, 481–511.



## A Preliminaries (Continued)

### A.1 Implementation of Shamir's Secret Sharing

We first describe the implementation of the two functions `SS.share` and `SS.recon` defined in Section 3. The first function can be implemented by uniformly randomly choosing  $t - 1$  coefficients  $a_1, \dots, a_{t-1}$  from  $\mathbb{Z}_q$ , and calculates  $s_i = f(x_i)$  for  $f(x) = s + a_1x + \dots + a_{t-1}x^{t-1}$ . The function  $f$  can be reconstructed from the shares with the Lagrange basis polynomials. More specifically, let  $\ell_i(x) = \prod_{j \neq i, j \in [n]} \frac{x - x_j}{x_i - x_j}$ , then  $f(x) = \sum_{i \in [n]} s_i \cdot \ell_i(x)$ . In this way, we can obtain  $s = f(0)$ . In fact, we can obtain shares  $f(x)$  for all values of  $x$  as long as we know the values of  $f(x_i)$  for at least  $t$  different  $x_i$ . Moreover, given the secret  $s$  and  $\{f(x_i)\}_{i \in [m]}$  for  $m < t - 1$ , we can always find the rest of the shares  $\{f(x_i)\}_{i \in [m+1, t]}$  such that  $s = \text{SS.recon}(\{(f(x_i), x_i)\}_{i \in [t]}, t)$  by arbitrarily choosing  $\{f(x_i)\}_{i \in [m+1, t-1]}$ , and calculating  $f(x_t)$  with the Lagrange basis polynomials. For simplicity, we call this process as *calculating the rest of the shares of  $s$  for indices  $\{x_i\}_{i \in [m+1, t]}$  fixing the shares  $\{f(x_i)\}_{i \in [m]}$* .

Additionally, we define the function `SS.expoRecon` as mentioned in Section 3 and its counterpart `SS.expoShare` which is used later in the security proofs as an extension of Shamir's secret sharing. Let  $p, q$  be primes such that  $p = 2q + 1$ . Let  $\mathbb{G}$  be the multiplicative cyclic subgroup of order  $q$  of  $\mathbb{Z}_p^*$  and let  $g$  be a generator of  $\mathbb{G}$ . Let  $s, s_{i_j}, a_i \in \mathbb{Z}_q$  for  $i \in [t]$  and  $i_j \in [q]$  for  $j \in [n]$ . We define two functions:

- `SS.expoRecon` $((g^{s_1}, x_1), \dots, (g^{s_n}, x_n), t) = \{g^s, g^{a_1}, \dots, g^{a_{t-1}}\}$ : With the shares  $g^{s_1}, \dots, g^{s_n}$ , it returns the secret and the polynomial coefficients of the Shamir secret sharing in the exponent. More precisely, it returns  $\{g^s, g^{a_1}, \dots, g^{a_{t-1}}\}$  such that for  $f(x) = s + a_1x + \dots + a_{t-1}x^{t-1}$ ,  $f(x_i) = s_i$  for  $i \in [n]$ . This function can be implemented without knowing  $s_1, \dots, s_n$  by performing all the linear operations of function `SS.recon` in the exponent.
- `SS.expoShare` $(g^s, g^{a_1}, \dots, g^{a_{t-1}}, x) = g^{s_x}$ : with the coefficients of the polynomial in the exponent, it returns a new share in the exponent at index  $x$ . More precisely, it returns  $g^{s_x} = g^s \cdot (g^{a_1})^x \cdot \dots \cdot (g^{a_{t-1}})^{x^{t-1}}$ . This function can also be implemented without knowing the exponents  $s, a_1, \dots, a_{t-1}$ .

Moreover, as all involved calculation is linear operations, we can implement *calculation of the rest of shares of a secret  $s$  for indices  $\{x_i\}_{i \in [m+1, t]}$  fixing the shares  $f(x_i)_{i \in [m]}$*  when both the secret and the shares are in the exponent of  $g$ .

**Shamir secret sharing on integer interval** The above implementation requires the secret and shares to be elements in a finite field. Now we provide a modified version of `SS.share` and `SS.recon` which can secret share and reconstruct secrets in a bounded interval of integers. We follow the algorithm described in Braun et al. (2022). The algorithm is similar to the implementation of Shamir secret sharing over a field with the key difference that as there is no modulo operation, the shares should not leak the remainder of the secret when divided by  $x_i$  and the division reconstruction steps need to happen in the integer set. Thus, we scale the secret  $s$  by  $\Delta = \prod_{i \in [n]} x_i$  times before secret sharing to guarantee that every share modulo  $x_i$  for any  $i$  equals 0, and scales the coefficients in the Lagrange basis polynomials to integers by scaling them by  $\Delta$  times. The reconstructed secret is therefore  $s \cdot \Delta^2$ . For simplicity of exposition, we overload the algorithm names `SS.share`, `SS.recon`, `SS.expoShare`, `SS.expoRecon` when which version of the algorithm is needed can be clearly inferred from the context.

### A.2 DDH Group with an Easy DL Subgroup

To accommodate larger input, we use an assumption from Castagnos and Laguillaumie (2015). Intuitively, it assume the existence of a group  $G = \langle g \rangle$  with a subgroup  $F = \langle f \rangle$  such that the DDH assumption holds in  $G$  and discrete logarithm is easy to calculate in group  $F$ . More formally, we use the following definition from Castagnos and Laguillaumie (2015):

**Definition A.1** (DDH Group with an Easy DL Subgroup Castagnos and Laguillaumie (2015)). *A DDH group with an easy DL subgroup is a pair of algorithms (cl.gen, cl.solve). The cl.gen algorithm is a group generator which takes as input two parameters  $\lambda$  and  $\mu$  and outputs a tuple  $(B, N, p, s, g, f, G, F)$ . The integers  $B, N, p$  and  $s$  are such that  $s$  is a  $\lambda$ -bit integer,  $p$  is a  $\mu$ -bit integer,  $\gcd(p, s) = 1$ ,  $N = p \cdot s$ , and  $B$  is an upper bound for  $s$ . The set  $(G, \cdot)$  is a cyclic group of order  $n$  generated by  $g$ , and  $F \subset G$  is the subgroup of  $G$  of order  $p$  and  $f$  is a generator of  $F$ . The upper bound  $B$  is chosen such that the distribution induced by  $\{g^r, r \xleftarrow{\$} \{0, \dots, Bp - 1\}\}$  is statistically indistinguishable from the uniform distribution on  $G$ . We assume that the canonical surjection  $\pi : G \rightarrow G/F$  is efficiently computable from the description of  $G, H$  and  $p$  and that given an element  $h \in G/F$  one can efficiently lift  $h$  in  $G$ , i.e., compute an element  $h_\ell \in \pi^{-1}(h)$ . We suppose moreover that:*



1. The DL problem is easy in  $F$ . The  $\text{cl.solve}$  algorithm is a deterministic polynomial time algorithm that solves the discrete logarithm problem in  $F$ :

$$\begin{aligned} \Pr[x = x^* : (B, N, p, s, g, f, G, F) \xleftarrow{\$} \text{cl.gen}(1^\lambda, 1^\mu), \\ x \xleftarrow{\$} \mathbb{Z}/p\mathbb{Z}, X = f^x, \\ x^* \leftarrow \text{cl.solve}(B, p, g, f, G, F, X)] = 1 \end{aligned}$$

2. The DDH problem is hard in  $G$  even with access to the  $\text{cl.solve}$  algorithm:

$$\begin{aligned} |\Pr[b = b^* : (B, N, p, s, g, f, G, F) \xleftarrow{\$} \text{cl.gen}(1^\lambda, 1^\mu), \\ x, y, z \xleftarrow{\$} \mathbb{Z}/N\mathbb{Z}, X = g^x, Y = g^y, \\ b \xleftarrow{\$} \{0, 1\}, Z_0 = g^z, Z_1 = g^{xy}, \\ b^* \xleftarrow{\$} \mathcal{A}(B, p, g, f, G, F, X, Y, Z_b, \text{cl.solve}(\cdot))] - \frac{1}{2}| \end{aligned}$$

is negligible for all probabilistic polynomial time adversary  $\mathcal{A}$ .

## B Security Proofs for MicroFedML<sub>1</sub>

In this section, we first recap the idea of the protocol MicroFedML<sub>1</sub> which guarantees privacy against malicious adversary as a whole picture, then we discuss the security properties of the protocol MicroFedML<sub>1</sub> in different adversarial settings.

As described in Section 2, in the Setup phase, each user  $i$  first agree with every other users  $j$  on the symmetric encryption key  $\text{ek}_{i,j}$ , by picking a pair of secret key  $\text{sk}_i$  and public key  $\text{pk}_i$  and sends the public key  $\text{pk}_i$  to all other users  $j$ . On receiving the public key  $\text{pk}_j$  from other users, the user  $i$  combines it with its own secret key  $\text{sk}_i$  to generate  $\text{ek}_{i,j}$ , which should be the same generated in the same way on user  $j$ 's side (see Section 3 for more details). Then it uniformly randomly picks a random mask  $r_i$ , and calculate the secret shares of it for all other users. Before sending the shares, user  $i$  encrypts the share for user  $j$  with  $\text{ek}_{i,j}$  so that the server cannot learn the share from the messages it forwards for them, while they can decrypt the messages between them with  $\text{ek}_{i,j}$ . Moreover, in the key agreement process mentioned above, a server controlled by a malicious adversary might sending users malicious public keys  $\text{pk}'$  rather than forwarding the public keys from honest users so that the user  $i$  is actually agreeing on a corrupt key with the adversary and all its encrypted messages can easily be decrypted by the adversary then. Thus, a public-key infrastructure (PKI) (see Section 3 for more details) which allows users to verify the source of messages is required. More specifically, in PKI, each user holds a secret signing key which is only known to itself and a public verification key which is known to all parties. User  $i$  signs the public key  $\text{pk}_i$  with its signing key generating a signature which cannot be forged by anyone not knowing its signing key and can be verified by anyone holding its public verification key that the message is from user  $i$ . It sends the signature with  $\text{pk}_i$  to the server and verifies all the signatures on  $\text{pk}_j$ 's from other users  $j$  with user  $j$ 's verification key.

After the Setup phase, each user should hold its own random mask  $r_i$  and one share  $r_{j,i}$  of  $r_j$  for each of other users  $j$ . Then, in each iteration, the user  $i$  first sends the masked input in the exponent  $g^{X_i} = g^{x_i + r_i}$  to the server. When user  $i$  receives the online set  $\mathcal{O}$  from the server, it needs to check if all other honest users receive the same online set. After agreeing on the online set by signing on the online set, sending the signature to other users, and verifying other users' signatures, each user sends  $g^{\sum_{j \in \mathcal{O}} r_{j,i}}$  to the server, who then reconstructs  $g^{\sum_{j \in \mathcal{O}} r_j}$ , removes it from  $g^{\sum_{j \in \mathcal{O}} X_j}$ , and recovers  $\sum_{j \in \mathcal{O}} x_j$  by discrete log.

Now, we discuss the security properties. First, we consider the case in which the server is honest and the adversary controls only a subset of users. In this setting, the adversary can never learn anything about the honest users' inputs no matter how many corrupt users it controls. The proof is straight forward: the joint view of any subset of users is independent of the other users' inputs, as each user never receives any information depend on other users' input value from the server by the description of the protocol. More formally, the joint view of any subset of users  $\mathcal{U}' \subset \mathcal{U}$ , can be simulated by a simulator SIM without knowing  $x_i$  for  $i \notin \mathcal{U}'$  by randomly choosing  $x'_i$  in the domain for each  $i \notin \mathcal{U}'$  and simulating the users  $i \notin \mathcal{U}'$  and the server following the protocol.

Now, we discuss the correctness property with dropouts and the privacy property in the semi-honest and malicious settings when the adversary controls both the server and a set of users  $\mathcal{C}$ .

### B.1 Privacy against Semi-Honest Adversary

We give proof of Theorem 4.2.

*Proof.* We first define the behavior of a simulator SIM:

- In the Setup phase:
  - **Round 1:** Each honest user  $i$  follows the protocol description in Algorithm 1.
  - **Round 2:** For each corrupt user  $j \in \mathcal{U}_i \cap \mathcal{C}$ , an honest user  $i$  stores  $\text{ek}_{i,j} = \text{KA.agree}(\text{pk}_j, \text{sk}_i)$ . For each pair of honest users  $i, j$ , the simulator uniformly randomly chooses a symmetric encryption key  $\text{ek}_{i,j}^*$ , and sets  $\text{ek}_{j,i}^* = \text{ek}_{i,j}^*$ .  
Then, for each corrupt user  $j \in \mathcal{U}_i^1 \cap \mathcal{C}$ , user  $i$  uniformly randomly chooses  $r_{i,j}$ , encrypts it by  $c_{i,j} \leftarrow \text{AE.enc}(r_{i,j}, \text{ek}_{i,j})$ ; for each honest user  $j \in \mathcal{U}_i^1 \setminus \mathcal{C}$ , user  $i$  encrypts a dummy value by  $c_{i,j} \leftarrow \text{AE.enc}(\perp, \text{ek}_{i,j}^*)$ . Each honest user  $i$  sends  $\{c_{i,j}\}_{j \in \mathcal{U}_i}$  to the server.
  - **Users Receiving Shares:** For each honest user  $i$ , on receiving  $c_{j,i}$  from the server, each honest user  $i$  follows the protocol except that it additionally aborts if for any honest  $j$ , the decryption succeeds while the result is different from the value user  $j$  encrypts in the previous round.
- In the  $k$ -th iteration of the Aggregation phase:
  - **Round 1:** Each honest user  $i$  uniformly randomly chooses  $X_i^*$  and sends  $H(k)^{X_i^*}$  to the server.
  - **Round 3:** If the online set  $\mathcal{O}$  the honest users receive from the server is of size at least  $t$ , the simulator queries the ideal functionality to get  $w = \text{Ideal}(\mathcal{O} \setminus \mathcal{C}, k)$ . Then for all honest users  $i \in \mathcal{O} \setminus \mathcal{C}$ , the simulator uniformly randomly samples  $w_i^*$  for  $i \in \mathcal{O} \setminus \mathcal{C}$  under the restriction  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} w_i^* = w$ . For each  $i \in \mathcal{O} \setminus \mathcal{C}$ , the simulator SIM calculates  $r_i^* = X_i^* - w_i^*$ , and calculates the shares  $r_{i,j}^*$  for all  $j \in \mathcal{U}_i \setminus \mathcal{C}$  such that  $r_i^* = \text{SS.recon}(\{r_{i,j}^*, j\}_{j \in \mathcal{U}_i \setminus \mathcal{C}}, \{r_{i,j}, j\}_{j \in \mathcal{C}}, t)$ , where  $r_{i,j}$  for  $j \in \mathcal{C}$  are the shares that have already been sent to the corrupt users in the Setup phase. Let  $r_{j,i}^* = r_{j,i}$  for  $j \in \mathcal{C}$  and honest user  $i$ .  
Then for the honest users  $i$  who receives  $\mathcal{O}$ , the simulator sends  $\zeta_i^* = H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}^*}$  to the server on behalf of user  $i$ .

We describe a series of hybrids between the joint view of corrupt parties in the real execution and the output of the simulation described above. Each hybrid is identical to the previous one except the part explicitly described. By proving that each hybrid is computationally indistinguishable from the previous one, we prove that the joint view of the corrupt parties in the real execution is indistinguishable from the simulation.

**Hyb0** This random variable is the joint view of all parties in  $\mathcal{C}$  in the real execution.

**Hyb1** In this hybrid, a simulator which knows all secret inputs of honest parties in every iteration simulates the execution with  $M_{\mathcal{C}}$  following the protocol.

The distribution of this hybrid is exactly the same as the previous one.

**Hyb2** In this hybrid, for any pair of two honest users  $i, j$ , the encryption of shares  $c_{i,j}$  and  $c_{j,i}$  they send between each other are encrypted and decrypted using a uniformly random key  $\text{ek}_{i,j}^*$  instead of  $\text{ek}_{i,j}$  obtained through Diffie Hellman key exchange in Setup Phase.

The indistinguishability between this hybrid and the previous one is guaranteed by 2ODH assumption.

**Hyb3** In this hybrid, we substitute each encrypted share  $c_{i,j}^r = \text{AE.enc}(\text{ek}_{i,j}^*, r_{i,j})$  sent between honest parties in the Setup phase in the previous hybrids with the encryption of a dummy value, i.e.,  $c_{i,j}^{r,*} = \text{AE.enc}(\perp, \text{ek}_{i,j}^*)$ .

The indistinguishability is guaranteed by IND-CPA security of the encryption scheme.

**Hyb4** In this hybrid, in every iteration  $k$ , each honest user  $i$  substitutes  $H(k)^{X_i}$  it sends to the server in the first round with  $H(k)^{X_i^*}$  for a uniformly randomly chosen  $X_i^*$ . Moreover, in the third round, for each honest user  $i$ , SIM calculates  $r_i^* = X_i^* - x_i$  and the shares  $r_{i,j}^*$  for honest users  $j$  based on the shares which have already been sent to corrupt users in the Setup phase, i.e., it calculates  $r_{i,j}^*$  for  $j \in \mathcal{U} \setminus \mathcal{C}$  making sure that  $r_i^* = \text{SS.recon}(\{r_{i,j}^*, j\}_{j \in \mathcal{U} \setminus \mathcal{C}}, \{r_{i,j}, j\}_{j \in \mathcal{C}})$ . For corrupt users  $j \in \mathcal{C}$ , let  $r_{j,i}^* = r_{j,i}$ . Then each honest user  $i$  who receives  $\mathcal{O}$  with at least  $t$  valid signatures calculates  $\zeta_i^* = H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}^*}$  and sends  $\zeta_i^*$  to the server.

By Lemma B.4, this hybrid is indistinguishable from the previous one.

**Hyb5** In this hybrid, in the second round of each iteration, for each user  $i \in \mathcal{O} \setminus \mathcal{C}$ , instead of setting  $r_i^* = X_i^* - x_i^*$ , SIM randomly picks  $r_i^*$  under the constraint that  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} r_i^* = \sum_{i \in \mathcal{O} \setminus \mathcal{C}} X_i^* - \sum_{i \in \mathcal{O} \setminus \mathcal{C}} x_i$ . The simulator then uses  $r_i^*$  to calculate the shares  $\{r_{i,j}^*\}_{j \in \mathcal{U} \setminus \mathcal{C}}$  for user  $i \in \mathcal{O} \setminus \mathcal{C}$ .

This hybrid is indistinguishable from the previous hybrid, as  $r_i^*$  are still uniformly random, and the sum of  $r_i^*$  in the exponent of  $H(k)^{\sum_{i \in \mathcal{O} \setminus \mathcal{C}} r_i^*}$  that the server can reconstruct from the shares keeps the same.

**Hyb6** In this hybrid, in the second round of each iteration, for each honest user  $i \notin \mathcal{O}$ , the simulator sets  $r_i^*$  as 0 and calculates the shares for  $i$ .

This hybrid is identical to the previous one as there is only one unique  $\mathcal{O}$  and if an honest user is not included in  $\mathcal{O}$ , the share  $r_{i,j}^*$  for  $j \in \mathcal{U} \setminus \mathcal{C}$  will not be included in any  $\zeta_j^*$  sent to server. Thus, the adversary will not receive any information about  $r_i^*$  in the third round of the iteration.

**Hyb7** Instead of receiving the inputs from the honest parties and using  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} x_i$  to sample  $r_i^*$  for  $i \in \mathcal{O} \setminus \mathcal{C}$ , the simulator makes a query to the functionality *Ideal* with the user set  $\mathcal{O} \setminus \mathcal{C}$  and iteration counter  $k$  and use the output value to sample random value  $r_i^*$  in every iteration with  $|\mathcal{O} \setminus \mathcal{C}| \geq t - n_{\mathcal{C}}$ . Note that the *Ideal* functionality will not return  $\perp$  in this case.

The distribution of this hybrid is exactly the same as the distribution of the previous hybrid. In this hybrid, the simulator does not know  $x_i$  for any user  $i$ .

Now we have proved that the joint view of  $M_{\mathcal{C}}$  in the real execution is computationally indistinguishable from the view in the simulated execution. □

## B.2 Privacy against Malicious Adversary

In this section, we prove Theorem 4.3 that our protocol protects the privacy of honest users in the active adversary setting with compromised server. In other words, we prove that when executing the protocol with threshold  $t \geq \lfloor \frac{2}{3}n \rfloor + 1$ , the joint view of the server and any set of less than  $t$  users does not leak any information about the other users' inputs other than what can be inferred from the output of the computation. In this work we do not consider the full security guarantee in the malicious setting, which means when a subset of users are malicious, we do not guarantee that the server learns the aggregation of the honest and online users' inputs.

For some iteration  $k$  of the Aggregation phase, We say a user set  $\mathcal{O} \subseteq \mathcal{U}$  is a *common online set* if some honest user receives at least  $t$  valid signatures on  $\mathcal{O}$  in the third round. This set might not exist when the server is corrupt.

**Fact B.1** (Unique Common Online Set). *When  $t > \frac{2}{3}n$  and  $|\mathcal{C}| < \frac{1}{3}n$ , There is at most one common online set  $\mathcal{O}$  in every iteration.*

*Proof.* For the sake of contradiction, assume there exists two different common online set  $\mathcal{O}_1$  and  $\mathcal{O}_2$  in some iteration  $k$ . Let  $\mathcal{U}_1$  and  $\mathcal{U}_2$  denote the set of honest users who sign on  $\mathcal{O}_1$  and  $\mathcal{O}_2$  respectively. By the definition of common online set,  $|\mathcal{U}_1| \geq t - |\mathcal{C}|$  and  $|\mathcal{U}_2| \geq t - |\mathcal{C}|$ . Thus  $|\mathcal{U}_1| + |\mathcal{U}_2| \geq 2t - 2|\mathcal{C}| > \frac{4}{3}n - \frac{1}{3}n - |\mathcal{C}| = n - |\mathcal{C}|$ , which is total number of honest users. Thus we have a contradiction as an honest user will only sign on one online set. □

The Theorem 4.3 shows that the joint view of any subset of less than  $t$  users and the server can be simulated without knowing the secret input of any other users. In other words, the adversary controlling the server and less than  $t$  users cannot learn anything other than the output of the computation. We give the proof below.

*Proof.* We first define the behavior of a simulator SIM:

- In the Setup phase:
  - **Round 1:** Each honest user  $i$  follows the protocol description in Algorithm 1.
  - **Round 2:** Each honest user  $i$  receives  $(pk_j, \sigma_j)$  from the server, and verifies the signatures as described in Algorithm 1, except that the simulator aborts if some honest user  $i$  receives a valid signature of an honest user  $j$  on a public encryption key different from what user  $j$  sends to the server in the previous round. Then for each corrupt user  $j \in \mathcal{U}_i \cap \mathcal{C}$ , an honest user  $i$  stores  $ek_{i,j} = \text{KA.agree}(pk_j, sk_i)$ . For

each pair of honest users  $i, j$ , the simulator uniformly randomly chooses a symmetric encryption key  $ek_{i,j}^*$ , and sets  $ek_{j,i}^* = ek_{i,j}^*$ .

Then, for each corrupt user  $j \in \mathcal{U}_i^1 \cap \mathcal{C}$ , user  $i$  uniformly randomly chooses  $r_{i,j}$ , encrypts it by  $c_{i,j} \leftarrow \text{AE.enc}(r_{i,j}, ek_{i,j}^*)$ ; for each honest user  $j \in \mathcal{U}_i^1 \setminus \mathcal{C}$ , user  $i$  encrypts a dummy value by  $c_{i,j} \leftarrow \text{AE.enc}(\perp, ek_{i,j}^*)$ . Each honest user  $i$  sends  $\{c_{i,j}\}_{j \in \mathcal{U}_i}$  to the server.

- **Users Receiving Shares:** For each honest user  $i$ , on receiving  $c_{j,i}$  from the server, each honest user  $i$  follows the protocol except that it additionally aborts if for any honest  $j$ , the decryption succeeds while the result is different from the value user  $j$  encrypts in the previous round.

- In the  $k$ -th iteration of the Aggregation phase:

- **Round 1:** Each honest user  $i$  uniformly randomly chooses  $X_i^*$  and sends  $H(k)^{X_i^*}$  to the server.
- **Round 2:** Each honest user follows the protocol by signing the online set  $\mathcal{O}$  it receives and sending the signature to the server.
- **Round 3:** If any honest user receives at least  $t$  valid signatures on the online set  $\mathcal{O}$  it receives in the previous round, the simulator queries the ideal functionality to get  $w = \text{Ideal}(\mathcal{O} \setminus \mathcal{C}, k)$ . Then for all honest users  $i \in \mathcal{O} \setminus \mathcal{C}$ , the simulator uniformly randomly samples  $w_i^*$  for  $i \in \mathcal{O} \setminus \mathcal{C}$  under the restriction  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} w_i^* = w$ . For each  $i \in \mathcal{O} \setminus \mathcal{C}$ , the simulator  $\text{SIM}$  calculates  $r_i^* = X_i^* - w_i^*$ , and calculates the shares  $r_{i,j}^*$  for all  $j \in \mathcal{U}_i \setminus \mathcal{C}$  such that  $r_i^* = \text{SS.recon}(\{r_{i,j}^*, j\}_{j \in \mathcal{U}_i \setminus \mathcal{C}}, \{r_{i,j}, j\}_{j \in \mathcal{C}}, t)$ , where  $r_{i,j}$  for  $j \in \mathcal{C}$  are the shares that have already been sent to the corrupt users in the Setup phase. Let  $r_{j,i}^* = r_{j,i}$  for  $j \in \mathcal{C}$  and honest user  $i$ .

Then for the honest users  $i$  who receives  $\mathcal{O}$  with at least  $t$  valid signatures from the server in the second round, the simulator sends  $\zeta_i^* = H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}^*}$  to the server on behalf of user  $i$ .

By Fact B.1, there will be at most one unique set  $\mathcal{O}$  that collects enough number of valid signatures and the Ideal functionality will be queried at most once each iteration.

We describe a series of hybrids between the joint view of corrupt parties in the real execution and the output of the simulation described above. Each hybrid is identical to the previous one except the part explicitly described. By proving that each hybrid is computationally indistinguishable from the previous one, we prove that the joint view of the corrupt parties in the real execution is indistinguishable from the simulation.

**Hyb0** This random variable is the joint view of all parties in  $\mathcal{C}$  in the real execution.

**Hyb1** In this hybrid, a simulator which knows all secret inputs of honest parties in every iteration simulates the execution with  $M_{\mathcal{C}}$  following the protocol.

The distribution of this hybrid is exactly the same as the previous one.

**Hyb2** In this hybrid, the simulator aborts if  $M_{\mathcal{C}}$  provides any of the honest parties  $j$  in the Setup phase with a valid signature with respect to an honest user  $i$ 's public key  $d_i^{PK}$  on  $pk_i^*$  different from what  $i$  provides.

The indistinguishability between this hybrid and the previous one is guaranteed by the security of the signature scheme.

**Hyb3** In this hybrid, for any pair of two honest users  $i, j$ , the encryption of shares  $c_{i,j}$  and  $c_{j,i}$  they send between each other are encrypted and decrypted using a uniformly random key  $ek_{i,j}^*$  instead of  $ek_{i,j}$  obtained through Diffie Hellman key exchange in Setup Phase.

The indistinguishability between this hybrid and the previous one is guaranteed by 2ODH assumption.

**Hyb4** In this hybrid, we substitute each encrypted share  $c_{i,j}^r = \text{AE.enc}(ek_{i,j}^*, r_{i,j})$  sent between honest parties in the Setup phase in the previous hybrids with the encryption of a dummy value, i.e.,  $c_{i,j}^{r,*} = \text{AE.enc}(\perp, ek_{i,j}^*)$ .

The indistinguishability is guaranteed by IND-CPA security of the encryption scheme.

**Hyb5** In this hybrid, in every iteration  $k$ , each honest user  $i$  substitutes  $H(k)^{X_i}$  it sends to the server in the first round with  $H(k)^{X_i^*}$  for a uniformly randomly chosen  $X_i^*$ . Moreover, in the third round, for each honest user  $i$ ,  $\text{SIM}$  calculates  $r_i^* = X_i^* - x_i$  and the shares  $r_{i,j}^*$  for honest users  $j$  based on the shares which have already been sent to corrupt users in the Setup phase, i.e., it calculates  $r_{i,j}^*$  for  $j \in \mathcal{U} \setminus \mathcal{C}$  making sure that  $r_i^* = \text{SS.recon}(\{r_{i,j}^*, j\}_{j \in \mathcal{U} \setminus \mathcal{C}}, \{r_{i,j}, j\}_{j \in \mathcal{C}})$ . For corrupt users  $j \in \mathcal{C}$ , let  $r_{j,i}^* = r_{j,i}$ . Then each honest user  $i$  who receives  $\mathcal{O}$  with at least  $t$  valid signatures calculates  $\zeta_i^* = H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}^*}$  and sends  $\zeta_i^*$  to the server.

By Lemma B.4, this hybrid is indistinguishable from the previous one.

**Hyb6** In this hybrid, in each iteration, if some honest user receives  $\mathcal{O}$  with at least  $t$  valid signatures in the second round, then in the third round, for each user  $i \in \mathcal{O} \setminus \mathcal{C}$ , instead of setting  $r_i^* = X_i^* - x_i^*$ , SIM randomly picks  $r_i^*$  under the constraint that  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} r_i^* = \sum_{i \in \mathcal{O} \setminus \mathcal{C}} X_i^* - \sum_{i \in \mathcal{O} \setminus \mathcal{C}} x_i$ . The simulator then uses  $r_i^*$  to calculate the shares  $\{r_{i,j}^*\}_{j \in \mathcal{U} \setminus \mathcal{C}}$  for user  $i \in \mathcal{O} \setminus \mathcal{C}$ .

This hybrid is indistinguishable from the previous hybrid, as  $r_i^*$  are still uniformly random, and the sum of  $r_i^*$  in the exponent of  $H(k)^{\sum_{i \in \mathcal{O} \setminus \mathcal{C}} r_i^*}$  that the server can reconstruct from the shares keeps the same.

**Hyb7** In this hybrid, in each iteration, if some honest user receives  $\mathcal{O}$  with at least  $t$  valid signatures in the second round, then in the third round, for each honest user  $i \notin \mathcal{O}$ , the simulator sets  $r_i^*$  as 0 and calculates the shares for  $i$ .

This hybrid is identical to the previous one as there is only one unique  $\mathcal{O}$  and if an honest user is not included in  $\mathcal{O}$ , the share  $r_{i,j}^*$  for  $j \in \mathcal{U} \setminus \mathcal{C}$  will not be included in any  $\zeta_j^*$  sent to server. Thus, the adversary will not receive any information about  $r_i^*$  in the third round of the iteration.

**Hyb8** Instead of receiving the inputs from the honest parties and using  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} x_i$  to sample  $r_i^*$  for  $i \in \mathcal{O} \setminus \mathcal{C}$ , the simulator makes a query to the functionality Ideal with the user set  $\mathcal{O} \setminus \mathcal{C}$  and iteration counter  $k$  and use the output value to sample random value  $r_i^*$  in every iteration with  $|\mathcal{O} \setminus \mathcal{C}| \geq t - n_C$ . Note that the Ideal functionality will not return  $\perp$  in this case.

The distribution of this hybrid is exactly the same as the distribution of the previous hybrid. In this hybrid, the simulator does not know  $x_i$  for any user  $i$ .

Now we have proved that the joint view of  $M_C$  in the real execution is computationally indistinguishable from the view in the simulated execution.  $\square$

We prove that Hyb4 and Hyb5 are indistinguishable. First, we prove the following lemma, which is an extension of the DDH assumption.

**Lemma B.2** (Extension of the DDH Assumption). *Let  $p$  and  $q$  be two primes while  $p = 2q + 1$ . Let  $g$  be a generator of field  $\mathbb{Z}_p^*$ . If the DDH assumption holds, then for uniformly random  $a, b_1, \dots, b_t, b'_1, \dots, b'_n \in \mathbb{Z}_q$ , the following two distributions are computationally indistinguishable:*

$$(g^a, g^{b_1}, \dots, g^{b_n}, g^{ab_1}, \dots, g^{ab_n}) \quad (1)$$

$$(g^a, g^{b_1}, \dots, g^{b_n}, g^{ab'_1}, \dots, g^{ab'_n}) \quad (2)$$

*Proof.* We define hybrids  $\text{Hyb}_i = (g^a, g^{b_1}, \dots, g^{b_n}, g^{ab_1}, \dots, g^{ab_i}, g^{ab'_{i+1}}, \dots, g^{ab'_n})$  for  $i \in [0, n]$ . Based on the DDH assumption, we prove that the two neighboring hybrids are computationally indistinguishable. For the sake of contradiction, assume there is a PPT adversary  $\mathcal{A}$  which can distinguish between  $\text{Hyb}_{i-1}$  and  $\text{Hyb}_i$  for some  $i \in [1, n]$ . Then, we construct the following distinguisher  $\mathcal{D}(A, B, C)$  for DDH tuples: it first uniformly randomly picks  $b_1, \dots, b_{i-1}, b_{i+1}, \dots, b_n$  and  $b'_{i+1}, \dots, b'_n$ . Then it invokes  $\mathcal{A}$  with the tuple

$$(A, g^{b_1}, \dots, g^{b_{i-1}}, B, g^{b_{i+1}}, \dots, g^{b_n}, A^{b_1}, \dots, A^{b_{i-1}}, C, A^{b'_{i+1}}, \dots, A^{b'_n})$$

and outputs the bit  $\mathcal{A}$  outputs. Let  $A = g^a$ ,  $B = g^b$ . Then, when  $C = g^{ab}$ , the distribution  $\mathcal{D}$  feeds  $\mathcal{A}$  is just the distribution of  $\text{Hyb}_i$ , and when  $C = g^{ab'}$  for a uniformly random  $b'$ , the distribution  $\mathcal{D}$  feeds  $\mathcal{A}$  is the distribution of  $\text{Hyb}_{i-1}$ .  $\mathcal{D}$  succeeds if  $\mathcal{A}$  successfully distinguishes between the two hybrids. If the DDH assumption holds, such  $\mathcal{A}$  does not exist, and  $\text{Hyb}_{i-1}$  and  $\text{Hyb}_i$  are computationally indistinguishable.

As the distribution (1) is the same as  $\text{Hyb}_n$ , and the distribution (2) is the same as  $\text{Hyb}_0$ , the two distributions in the lemma are computationally indistinguishable.  $\square$

**Lemma B.3.** *Let  $n, t, n_C, K$  be integer parameters,  $n_C \leq n - t < t$ . Let  $x_1, \dots, x_{n-n_C}$  be uniformly random elements of some finite field  $\mathbb{F}$ . Let  $x_{i,j} \in \mathbb{Z}_q$  for  $i \in [n - n_C]$  and  $j \in [n]$  be shares of  $x_i$  calculated with  $t$ -out-of- $n$  Shamir secret sharing algorithm, i.e.,  $\{x_{i,j}\}_{j \in [n]} \leftarrow \text{SS.share}(x_i, [n], t)$  for  $i \in [n - n_C]$ .*

*For each  $k = 1, \dots, K$ , let  $y_1^k, \dots, y_{n-n_C}^k$  also be uniformly randomly chosen from  $\mathbb{F}$ . Let  $y_{i,j}^k$  for  $i, j \in [n - n_C]$  be elements of  $\mathbb{Z}_q$  such that  $y_i^k = \text{SS.recon}(\{y_{i,j}^k, j\}_{j \in [n-n_C]}, \{x_{i,j}^k, j\}_{j \in [n-n_C+1, n]}, t)$ . Let  $H(\cdot)$  be a random oracle that returns a random element of  $\mathbb{Z}_q^*$  on each fresh input.*

Then the following two distributions are computationally indistinguishable if the DDH assumption holds:

$$\{x_{i,j}\}_{i \in [n-n_c], j \in [n-n_c+1, n]}, \{ \{H(k)^{x_i}\}_{i \in [n-n_c]}, \{H(k)^{x_{i,j}}\}_{i,j \in [n-n_c]} \}_{k \in [K]} \quad (3)$$

$$\{x_{i,j}\}_{i \in [n-n_c], j \in [n-n_c+1, n]}, \{ \{H(k)^{y_i^k}\}_{i \in [n-n_c]}, \{H(k)^{y_{i,j}^k}\}_{i,j \in [n-n_c]} \}_{k \in [K]} \quad (4)$$

*Proof.* We prove the indistinguishability with a sequence of hybrids. Let  $\text{Hyb}_0$  equal to the distribution (3). Then for each  $k \in [K]$ ,  $\text{Hyb}_k$  is the same as  $\text{Hyb}_{k-1}$  except that  $\{H(k)^{x_i}\}_{i \in [n-n_c]}, \{H(k)^{x_{i,j}}\}_{i,j \in [n-n_c]}$  are substituted with  $\{H(k)^{y_i^k}\}_{i \in [n-n_c]}, \{H(k)^{y_{i,j}^k}\}_{i,j \in [n-n_c]}$ . Defined in this way,  $\text{Hyb}_K$  is identical to the distribution (4) in the lemma. Then between  $\text{Hyb}_k$  and  $\text{Hyb}_{k+1}$  for  $k \in [0, K-1]$ , we additionally define  $\text{Hyb}_{k,0} = \text{Hyb}_k$  and a sequence of hybrids  $\text{Hyb}_{k,i}$  for  $i \in [n-n_c]$ :  $\text{Hyb}_{k,i}$  is the same as  $\text{Hyb}_{k,i-1}$  except that  $H(k)^{x_i}, \{H(k)^{x_{i,j}}\}_{j \in [n-n_c]}$  in  $\text{Hyb}_{k,i-1}$  is substituted with  $H(k)^{y_i^k}, \{H(k)^{y_{i,j}^k}\}_{j \in [n-n_c]}$ . Note that  $\text{Hyb}_{k,n-n_c}$  is identical to  $\text{Hyb}_{k+1}$ .

Then we prove that the two adjacent hybrids  $\text{Hyb}_{k_0,i_0-1}$  and  $\text{Hyb}_{k_0,i_0}$  are computationally indistinguishable. For the sake of contradiction, assume that there exists a distinguisher

$$\mathcal{D}(\{x_{i,j}\}_{i \in [n-n_c], j \in [n-n_c+1, n]}, \{Z_{k,i}\}_{i \in [n-n_c]}, \{Z_{k,i,j}\}_{i,j \in [n-n_c]})_{k \in [K]}$$

can distinguish between these two distributions. Then we construct the following PPT distinguisher

$$\mathcal{D}'(A, B_1, \dots, B_{t-c}, C_1, \dots, C_{t-c}) :$$

For  $i \in [n-n_c]$ , it uniformly randomly samples  $x_i$  and calculates the  $t$ -out-of- $n$  Shamir secret sharing of  $x_i$  by  $\{x_{i,j}\}_{j \in [n]} \leftarrow \text{SS.share}(x_i, [n], t)$ . For  $k < k_0$ , it also it uniformly randomly samples  $y_i^k$  for  $i \in [n-n_c]$  and secret shares each  $y_i^k$  to generate shares  $\{y_{i,j}^k\}_{j \in [n]}$ . Moreover, for  $k \in [K]$  and  $k \neq k_0$ ,  $\mathcal{D}'$  uniformly randomly chooses  $s_k$  and assigns  $H(k) := g^{s_k}$ , and for  $k_0$ , it assigns  $A$  to  $H(k_0)$ . Then it feeds the following input to the distinguisher  $\mathcal{D}$ : For the first part, it feeds  $\mathcal{D}$  with  $x_{i,j}$  for  $i \in [n-n_c], j \in [n-n_c+1, n]$ ; then for the second part:

- For  $k < k_0$ , it sets  $Z_{k,i} = g^{s_k y_i^k}$  and  $Z_{k,i,j} = g^{s_k y_{i,j}^k}$ ; for  $k > k_0$ , let  $Z_{k,i} = g^{s_k x_i}$  and  $Z_{k,i,j} = g^{s_k x_{i,j}}$  for  $i, j \in [n-n_c]$ .
- For  $i < i_0$ , let  $Z_{k_0,i} = A^{y_i^{k_0}}$  and  $Z_{k_0,i,j} = A^{y_{i,j}^{k_0}}$ ; for  $i > i_0$ , let  $Z_{k_0,i} = A^{x_i}$  and  $Z_{k_0,i,j} = A^{x_{i,j}}$ .
- It sets  $Z_{k_0,i_0}$  and  $\{Z_{k_0,i_0,j}\}_{j \in [n-n_c]}$  in the following way: It calculates  $X_{i_0}$  by

$$X_{i_0}, C_1, \dots, C_{t-1} = \text{SS.expoRecon}(\begin{aligned} &(B_1, 1), \dots, (B_{t-n_c}, t-n_c), \\ &(g^{x_{i_0, n-n_c+1}}, n-n_c+1), \dots, (g^{x_{i_0, n}}, n), t), \end{aligned}$$

and the remaining shares  $X_{i_0,j}$  for  $j \in [t-n_c+1, n-n_c]$  by

$$X_{i_0,j} = \text{SS.expoShare}(j, X_{i_0}, C_1, \dots, C_{t-1}).$$

Let  $Z_{k_0,i_0} = X_{i_0}$ . For  $j \in [t-n_c]$ , let  $Z_{k_0,i_0,j} = C_j$ , and for  $j \in [t-n_c+1, n-n_c]$ , let  $Z_{k_0,i_0,j} = X_{i_0,j}$ .

Then  $\mathcal{D}'$  returns the bit  $\mathcal{D}$  outputs.

When  $C_i = g^{a b_i}$  for  $i \in [t-c]$ , the distribution of the input for  $\mathcal{D}$  is exactly the same as  $\text{Hyb}_{k_0,i_0-1}$ ; when  $C_i = g^{c_i}$  for uniformly random  $c_i$  for  $i \in [t-c]$ , the distribution of the input for  $\mathcal{D}$  is exactly the same as  $\text{Hyb}_{k_0,i_0}$ . Thus,  $\mathcal{D}$  win the games with the same probability as  $\mathcal{A}$  distinguishes between  $\text{Hyb}_{k_0,i_0-1}$  and  $\text{Hyb}_{k_0,i_0}$ . However, by Lemma B.2, there is no such a distinguisher  $\mathcal{D}'$ . Thus, we have a contradiction.  $\square$

**Lemma B.4.** *If the DDH assumption holds, then the distributions of  $\text{Hyb}_4$  and  $\text{Hyb}_5$  are computationally indistinguishable.*

*Proof.* For the sake of contradiction, assume there exists an adversary which can distinguish between Hyb4 and Hyb5. Then we can construct a distinguisher

$$\mathcal{D}(\{r_{i,j}^*\}_{i \in [n-n_c], j \in [n-n_c+1, n]}, \{Z_{k,i}\}_{i \in [n-n_c]}, \{Z_{k,i,j}\}_{i,j \in [n-n_c]}\}_{k \in [K]})$$

in the following way:

$\mathcal{D}$  simulates the protocol execution with  $\mathcal{A}$  as described in Hyb4, except that in Round 2 of the Setup phase, each honest user sends  $r_{i,j}^*$ ; in each iteration  $k$ , for each honest user  $i$ , it substitutes  $H(k)^{X_i}$  with  $Z_{k,i}$  in the first round, and substitutes  $\zeta_i = \prod_{j \in \mathcal{O}} H(k)^{r_{j,i}}$  with  $\zeta_i^* = \prod_{j \in \mathcal{O}} Z_{k,j,i}$  in the last round. Then it outputs the bit  $\mathcal{A}$  outputs.

When the input of  $\mathcal{D}$  is sampled from the distribution (3), the distribution of the simulation is identical to Hyb4, and when the input of  $\mathcal{D}$  is sampled from the distribution (4), the distribution of the simulation is identical to Hyb5. Thus,  $\mathcal{D}$  successfully distinguishes between the two distributions when  $\mathcal{A}$  succeeds. However, by Lemma B.3, such a distinguisher  $\mathcal{D}$  does not exist under the DDH assumption. Thus, we have a contradiction.  $\square$

## C Security Proofs for MicroFedML<sub>2</sub>

### C.1 Group Properties

*Proof. (for Lemma 5.4)* We first prove that constraint 1 in Definition 5.3 is satisfied. Let  $m = \frac{\gamma n N}{n-1} + \sqrt{\frac{N}{2}(\sigma \log 2 + \log n)}$ . Fixing an arbitrary honest user  $i$ , let  $X$  denote the number of corrupt users falling in the same group as user  $i$ . Then  $X \sim \text{HyperGeom}(n-1, \gamma n, N)$ . By the tail bound of the hypergeometric distribution,

$$\begin{aligned} \Pr[X \geq m] &= \Pr\left[X \geq \left(\frac{\gamma n}{n-1} + \sqrt{\frac{1}{2N}(\sigma \log 2 + \log n)}\right) \cdot N\right] \\ &\leq e^{-2N \cdot \frac{1}{2N}(\sigma \log 2 + \log n)} = \frac{1}{n} \cdot 2^{-\sigma}. \end{aligned}$$

As  $t \geq (3 + \gamma - 2\delta)N/4$ , we have that  $2t - N \geq (1 + \gamma - 2\delta)N/2$ . Then as long as

$$N \geq \frac{\sigma \log 2 + \log n}{2} \left( \frac{1 + \gamma - 2\delta}{2} - \frac{\gamma n}{n-1} \right)^{-2},$$

we have that  $2t - N \geq m$ , i.e.,  $\Pr[X \geq 2t - N] \leq \frac{1}{n} \cdot 2^{-\sigma}$ . Taking the union bound over all users, we have that event  $E_1$  happens with overwhelming probability. This can be achieved by choosing

$$c \geq \frac{1}{2} \left( \frac{1 + \gamma - 2\sigma}{2} - \frac{\gamma n}{n-1} \right)^{-2}.$$

For the second constraint, let  $Y$  denote the number of offline users in the honest and online user  $i$ 's group. Then  $Y \sim \text{HyperGeom}(n-1, \delta n, N)$ . Let  $m' = \frac{\delta n N}{n-1} + \sqrt{\frac{N}{2}(\eta \log 2 + \log n)}$ . Then we have

$$\Pr[Y \geq m'] \leq \frac{1}{n} \cdot 2^{-\eta}.$$

If  $N - t \geq m'$ , then by the same argument, we have the second constraint holds. By choosing

$$N \geq \frac{\eta \log 2 + \log n}{2} \left( \frac{3 + \gamma - 2\delta}{4} - \frac{(1 - \delta)n}{n-1} \right)^{-2},$$

we have  $N - t \geq m'$  when  $n$  is sufficiently large. This can be achieved by setting

$$c > \frac{1}{2} \left( \frac{3 + \gamma - 2\delta}{4} - \frac{(1 - \delta)n}{n-1} \right)^{-2}.$$

Both bounds of  $c$  are bounded when  $n$  is sufficiently large. Thus, we can choose a constant  $c$  that is larger than these two bounds.  $\square$

## C.2 Correctness

We show that with every party following the protocol except a subset of users dropping offline in every iteration, if the grouping algorithm is  $(\sigma, \eta, \mathcal{C})$ -nice, then the server can learn the sum of at least  $Bt$  clients' inputs at the end of iteration  $k$  with one but negligible probability for each  $k \in [K]$ .

*Proof.* As all users and the server are assumed to follow the protocol, the participants of the Aggregation phase should be the same in all users' and the server's view. Denote the participants of the Aggregation phase with  $\mathcal{U}$ , and let  $G_d$  denote the participants of the Aggregation phase in group  $d$ . As the grouping algorithm is  $(\sigma, \eta, \mathcal{C})$ -nice, by definition, event  $E_2$  fails to happen with probability  $2^{-\eta}$ . By union bound, the event that in every iteration there are at least  $t$  users online in every group happen with probability at least  $1 - K \cdot 2^{-\eta}$ . If in iteration  $k$ , for each group  $G_d$ , there are at least  $t$  users online at the end of the iteration, then in the last round of the iteration, the server can reconstruct

$$R_d = \text{SS.expoRecon}(\{\zeta_i\}_{i \in \mathcal{O}_d}, t) = H(k)^{\sum_{i \in \mathcal{O}_d} r_i - \sum_{i \in G_d \setminus \mathcal{O}_d} h_i},$$

and by calculating the discrete log of  $H(k)^{\sum_{i \in \mathcal{O}} X_i} / \prod_{d \in [B]} R_d$ , the server obtains  $z = \sum_{i \in \mathcal{O}} X_i - \sum_{i \in \mathcal{O}} r_i + \sum_{i \in \mathcal{U} \setminus \mathcal{O}} h_i = \sum_{i \in \mathcal{O}} x_i + \sum_{i \in \mathcal{U}} h_i$ . As  $\sum_{i \in \text{offline}} h_i + \sum_{i \in \mathcal{U}} h_i = 0$ , by adding  $h_S$  to  $z$ , the server gets the sum  $\sum_{i \in \mathcal{O}} x_i$ .  $\square$

## C.3 Privacy

Before proving the privacy guarantee against malicious adversary, we define several notions used in the proof.

**Participation in the Aggregation phase** For a user  $i \in G_d$ , we say *the user  $i$  participates in the Aggregation phase* if it is included in less than  $t - |G_d \cap \mathcal{C}|$  honest users' offline lists at the end of the fourth round of the Setup phase.

**Lemma C.1.** *Assume event  $E_1$  happens, i.e., there are less than  $2t - \frac{n}{B}$  corrupt users in any group. If for any user  $i$ ,  $\text{sk}_i^{-1}$  and  $\text{sk}_i^1$  are reconstructed by the server, then  $i$  must not participate in the Aggregation phase; if user  $i$  participates in the Aggregation phase, at least one of  $\text{sk}_i^{-1}$  and  $\text{sk}_i^1$  is hidden from the server.*

*Proof.* If  $\text{sk}_i^1$  for a user  $i \in G_d$  is reconstructed by the server, there must be at least  $t$  members of group  $G_{d+1}$  sending the shares to the server in the fifth round of the Setup phase, at least  $t - |\mathcal{C} \cap G_{d+1}|$  of which are from honest users. All of these honest users must have received at least  $t$  valid signatures on offline sets that includes user  $i$ . At least  $t - |\mathcal{C} \cap G_d|$  of these signatures come from honest users in  $G_d$  who have put  $i$  in their offline list. By definition,  $i$  is not participating in the Aggregation phase.  $\square$

**Common online set of a group** For some iteration  $k$  of the Aggregation phase, we say a user set  $\mathcal{O}_d \subseteq G_d$  is a *common online set of group  $d$*  if some honest user in  $G_d$  receives at least  $t$  valid signatures on  $\mathcal{O}_d$  in the third round. This set might not exist when the server is corrupt. Then we have the following fact:

**Fact C.2** (Unique Common Online Set each Group). *When  $2t > n/B + |\mathcal{C} \cap G_d|$ , there is at most one common online set  $\mathcal{O}_d$  for each group  $d$  in every iteration.*

This statement can be proved with the same reasoning as the proof of Fact B.1. Now, we give the proof for Theorem 5.6.

*Proof. (of Theorem 5.6)* By saying that an honest user uses  $r'$  (or  $h'$ ) as  $r$ -mask (or  $h$ -mask) in iteration  $k$  of the Aggregation phase, we mean that in the first round of the iteration  $k$ , the user uses  $r'$  (or  $h'$ ) to calculate  $X_i$ ; it also calculates the shares  $r'_{i,j}$  of  $r'$  for honest users in its group fixing the shares that have already been sent to its corrupt neighbors. Then in the third round, every honest user  $j$  in its group uses  $r'_{i,j}$  or  $h'_{i,j}$  to calculate the sum of shares.

As the good events happen with overwhelming probability when the grouping algorithm is  $(\sigma, \eta, \mathcal{C})$ -nice, we only considers the case when both events  $E_1$  and  $E_2$  happen. We first define the behavior of the simulator SIM:

- In the Setup phase:
  - **Round 1:** The simulator simulates each honest user following the protocol.
  - **Round 2:** Each honest user  $i$  receives the public keys and the signatures  $(\text{pk}_j, \sigma_j)$  from the server, and verifies the signatures as described in Algorithm 3, except that the simulator additionally aborts if some honest user  $i$  receives a valid signature of an honest user  $j$  on a public encryption key different from what user  $j$  sends to the server in the previous round. Then for each corrupt user  $j \in \mathcal{U}_i^1 \cap \mathcal{C}$ , an honest user  $i$



stores  $ek_{i,j} = \text{KA.agree}(pk_j, sk_i)$ . For each pair of honest users  $i, j$ , the simulator uniformly randomly chooses a symmetric encryption key  $ek_{i,j}^*$ , and sets  $ek_{j,i}^* = ek_{i,j}^*$ . For each  $j \in \mathcal{U}_i^1 \cap (G_{d-1} \cup G_{d+1})$ , each honest user  $i$  also stores  $mk_{i,j}$ . Then it follows the protocol, except that for honest user  $j \in G_{d-1}$ , instead of encrypting the share  $sk_{i,j}^{-1}$ , it encrypts some dummy value by  $c_{i,j}^{sk^{-1}*} \leftarrow \text{AE.enc}(0, ek_{i,j}^*)$ , and for honest user  $j \in G_{d+1}$  it does the same symmetrically.

- **Round 3:** Each honest user  $i$  decrypts the shares as described in the protocol to get  $sk_{j,i}^{-1}$  (or  $sk_{j,i}^1$ ) for each  $j \in \mathcal{U}_i^2 \cap G_{d+1}$  (or  $j \in \mathcal{U}_i^2 \cap G_{d-1}$ ). In this process, the simulator additionally aborts if for any honest user  $j \in \mathcal{U}_i^2$ , the decryption succeeds while the result is different from what  $j$  encrypts in the previous round. Moreover, the simulator uniformly randomly chooses  $mk_{i,j}^*$  for each pair of honest users  $i \in G_d$  and  $j \in G_{d+1}$ , and let  $mk_{i,j}^* = mk_{i,j}$  for each honest user  $i \in G_d$  and each corrupt user  $j \in G_{d+1}$ . Then each honest user  $i$  calculates  $h_i^* = \sum_{j \in \mathcal{U}_i^2 \cap G_{d-1}} mk_{i,j}^* - \sum_{j \in \mathcal{U}_i^2 \cap G_{d+1}} mk_{i,j}^*$ . Then it follows the protocol to secret shares  $r_i$  and  $h_i$  among group members of  $G_d$  and encrypts the shares except that it substitutes the encrypted shares sent to honest user  $j \in G_d$  with the encryption of a dummy value with  $ek_{i,j}^*$ .
- **Round 4:** Each honest user  $i$  decrypts the shares for  $j \in \mathcal{U}_i^3$  as described in protocol. In this process, the simulator additionally aborts if for any honest user  $j \in \mathcal{U}_i^3$ , the decryption succeeds while the result is different from what  $j$  encrypts in the previous round. Then each user  $i$  follows the protocol to sign the offline list  $\text{offline}_i$  and sends the list and the signature to the server.
- **Round 5:** On receiving  $(\text{offline}_j, \sigma_j)$  from the server, the user  $i$  aborts if any signature is invalid. The simulator additionally aborts if any honest user  $i$  receives  $\text{offline}'_j \neq \text{offline}_j$  for another honest user  $j$  with valid signature with respect to  $pk_j$ . Otherwise, each honest user  $i$  follows the protocol in this round.

At the end of the Setup phase, for each group  $G_d$ , the simulator checks if there is any honest user  $i \in G_d$  such that at least  $t - n_C$  shares  $sk_{i,j}^1$  (or  $sk_{i,j}^{-1}$ ) are sent to the server from honest users  $j \in G_{d+1}$  (or  $\mathcal{A} \in G_{d-1}$ ) and puts such users  $i$  in a user list  $\text{offline}_{\text{SIM}}$ .

- In the  $k$ -th iteration of the Aggregation phase:
  - **Round 1:** Each honest user  $i$  uniformly randomly chooses  $X_i^*$  and sends  $H(k)^{X_i^*}$  to the server.
  - **Round 2:** For all group  $G_d$ , each honest user  $i \in G_d$  follows the protocol, signs  $\mathcal{O}_d$  and sends the signature to the server.
  - **Round 3:** For each group  $G_d$ , the simulator checks if there are some honest users  $i \in G_d$  receiving at least  $t$  valid signatures on  $\mathcal{O}_d$  it receives in Round 2.
    - \* If there are such users in every group, then let  $\mathcal{O} = \cup_{d \in [B]} \mathcal{O}_d$ , the simulator queries the ideal functionality to get  $w = \text{Ideal}(\mathcal{O} \setminus \mathcal{C}, k)$ . The simulator then uniformly randomly chooses  $w_i^*$  for  $i \in \mathcal{O}$  under the restriction that  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} w_i^* = w$ . For each iteration  $k \in [K]$ , it uniformly randomly picks  $h_i^*$  under the constraint that  $\sum_{i \in \text{offline}_{\text{SIM}}} h_i^* = \sum_{i \in \text{offline}_{\text{SIM}}} h_i$ . It then calculates  $r_i^* = X_i^* - w_i^* - h_i^*$ , and calculates the shares  $r_{i,j}^*$  of  $r_i^*$  for  $i \in G_d$  and  $j \in G_d \setminus \mathcal{C}$  based on  $r_{i,j}$  for  $j \in G_d \cap \mathcal{C}$  that have already been sent to the corrupt users in the Setup phase. Let  $r_{i,j}^* = r_{i,j}$  for  $i \in \mathcal{C}$ . The simulator sends  $\zeta_i^{r*}$  and  $\zeta_i^{h*}$  calculated as described in the protocol to the server, except that they are calculated with  $r_{j,i}^*$  and  $h_{j,i}^*$ .
    - \* if for any group  $d \in [B]$  there is no such  $\mathcal{O}_d$ , the simulator uniformly randomly chooses the random mask  $r_i^*$  and the mutual mask  $h_i^*$  of each honest user  $i$ , and calculates the shares of  $r_i^*$  and  $h_i^*$  based on the shares that have already been sent to the corrupt users in the Setup phase. Then each honest user calculates  $\zeta_i^*$  using the new shares and sends to the server.

We describe a series of hybrids between the joint view of corrupt parties in the real execution and the output of the simulation. Each hybrid is identical to the previous one except the part explicitly described. By proving that each hybrid is computationally indistinguishable from the previous one, we prove that the joint view of corrupt parties in the real execution is indistinguishable from the simulation.

**Hyb0** This random variable is the joint view of all parties in  $\mathcal{C}$  in the real execution.

**Hyb1** In this hybrid, a simulator which knows all secret inputs of honest parties in every iteration simulates the execution with  $M_{\mathcal{C}}$ .

The distribution of this hybrid is exactly the same as the previous one.

Hyb2 In this hybrid, the simulator aborts if  $M_C$  provides any of the honest parties  $j$  in the Setup phase with a valid signature with respect to an honest user  $i$ 's public key  $d_i^{PK}$  on public encryption and masking keys different from what  $i$  provides.

The indistinguishability between this hybrid and the previous one is guaranteed by the security of the signature scheme.

Hyb3 In this hybrid, for any pair of two honest users  $i, j$ , the encryption of shares they send between each other in Round 2 and Round 3 of the Setup Phase are encrypted and decrypted using a uniformly random key  $ek_{i,j}^*$  instead of  $ek_{i,j}$  obtained through Diffie Hellman key exchange in Round 1 of the Setup Phase.

The indistinguishability between this hybrid and the previous one is guaranteed by 2ODH assumption.

Hyb4 In this hybrid, each encrypted share sent between each two honest parties  $i, j$  in the Setup phase in the previous hybrids is substituted with the encryption of a dummy value  $\perp$  with  $ek_{i,j}^*$ .

The indistinguishability is guaranteed by IND-CPA security of the encryption scheme.

Hyb5 In this hybrid, in every iteration  $k$ , each honest user  $i$  substitutes  $H(k)^{X_i}$  it sends to the server in the first round with  $H(k)^{X_i^*}$  for a uniformly randomly chosen  $X_i^*$ . Moreover, in the third round, for each honest user  $i$ , SIM calculates  $r_i^* = X_i^* - x_i - h_i$  and the shares  $r_{i,j}^*$  for honest users  $j$  based on the shares which have already been sent to corrupt users in the Setup phase, i.e., it calculates  $r_{i,j}^*$  for  $j \in \mathcal{U} \setminus \mathcal{C}$  making sure that  $r_i^* = \text{SS.recon}(\{r_{i,j}^*, j\}_{j \in \mathcal{U} \setminus \mathcal{C}}, \{r_{i,j}, j\}_{j \in \mathcal{C}})$ . For corrupt users  $j \in \mathcal{C}$ , let  $r_{j,i}^* = r_{j,i}$ . Then each honest user  $i \in G_d$  who receives the common online set  $\mathcal{O}_d$  with at least  $t$  valid signatures calculates  $\zeta_i^* = H_c(k)^{\sum_{j \in \mathcal{O}_d} r_{j,i}^* - \sum_{j \in \mathcal{U} \setminus \mathcal{O}_d} h_{j,i}}$  and sends  $\zeta_i^*$  to the server.

With the same reasoning as in the proof of Theorem B.4, this hybrid is indistinguishable from the previous one.

Hyb6 In this hybrid, in the third round of each iteration, for each honest user  $i \in G_d$  not included in  $\mathcal{O}_d$  and each  $\rho \in [a]$ , instead of setting  $r_i^* = X_i^* - x_i - h_i$ , SIM uniformly randomly picks  $r_i^*$  and uses it to calculate the shares for the honest users as described in the previous hybrid.

This hybrid is identical to the previous one as there is at most one unique  $\mathcal{O}_d$  and if an honest user is not included in  $\mathcal{O}_d$ , the share  $r_{i,j}^*$  for honest user  $i$  not in  $\mathcal{O}$  will not be included in  $\zeta_j^*$  of honest user  $j$ . Thus, the adversary will not receive any information about  $r_i^*$  in the third round of the iteration.

Hyb7 In this hybrid, in the third round of each iteration, for each user  $i \in \mathcal{O}_d \setminus \mathcal{C}$ , instead of setting  $r_i^* = X_i^* - x_i - h_i$ , SIM randomly picks  $r_i^*$  under the constraint that  $\sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} r_i^* = \sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} X_i^* - \sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} (x_i + h_i)$ .

This hybrid is indistinguishable from the previous hybrid, as  $r_i^*$  are still uniformly random, and the sum of  $r_i^*$  the server can reconstruct from the shares for each group keeps the same.

Hyb8 In this hybrid, at the end of the Setup phase, the simulator uniformly randomly chooses  $mk_{i,j}^*$  for each pair of honest users  $i, j \notin \text{offline}_{\text{SIM}}$  from two adjacent groups. For honest user  $i \in G_d \setminus \text{offline}_{\text{SIM}}$  and user  $j \in (\mathcal{C} \cup \text{offline}_{\text{SIM}}) \cap G_{d \pm 1}$ , let  $mk_{i,j}^* = mk_{i,j}$  obtained in the Setup phase. The simulator then uses  $mk_{i,j}^*$  to calculate  $h_i^*$  for each honest user  $i \notin \text{offline}_{\text{SIM}}$  and uses  $h_i^*$  as  $h_i$  in the Aggregation phase.

This hybrid is indistinguishable from the previous one, as the server does not know any information about  $sk_i^{\pm 1}$  for any honest user  $i \notin \text{offline}_{\text{SIM}}$  (guaranteed by the security of Shamir secret sharing). Thus,  $mk_{i,j}^*$  for honest users  $i, j \notin \text{offline}_{\text{SIM}}$  is indistinguishable from  $mk_{i,j}$  chosen uniformly randomly.

Hyb9 Instead of choosing  $mk_{i,j}^*$  for each pair of honest users  $i, j \notin \text{offline}_{\text{SIM}}$ , the simulator just choose  $h_i^*$  for each honest user  $i \notin \text{offline}_{\text{SIM}}$  uniformly at random under the constraint that  $\sum_{i \notin \text{offline}_{\text{SIM}}} h_i^* = \sum_{i \notin \text{offline}_{\text{SIM}}} h_i$ .

By Lemma C.5, this hybrid is identical to the previous one.

Hyb10 In this hybrid, instead of using fixed  $h_i^*$  chosen at the end of the Setup phase, the simulator uniformly randomly chooses  $h_i^*$  for each honest user  $i \notin \text{offline}_{\text{SIM}}$  under the same constraint  $\sum_{i \notin \text{offline}_{\text{SIM}}} h_i^* = \sum_{i \notin \text{offline}_{\text{SIM}}} h_i$  at the beginning of each iteration.

This hybrid is indistinguishable from the previous one.

Hyb11 When  $\mathcal{O}_d$  exists for each  $d \in [B]$ , instead using the constraint  $\sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} r_i^* = \sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} X_i^* - \sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} (x_i + h_i^*)$  for each  $d \in [B]$  to randomly pick  $r_i^*$  for each  $i \in \mathcal{O}_d \setminus \mathcal{C}$ , the simulator uses the constraint  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} r_i^* = \sum_{i \in \mathcal{O} \setminus \mathcal{C}} X_i^* - \sum_{i \in \mathcal{O} \setminus \mathcal{C}} (x_i + h_i^*)$ .

This hybrid is identical to the previous one, as it is the same as the following hybrid: in the third round of iteration  $k$ , each honest user first chooses  $h_i^*$  under the constraint that  $\sum_{i \notin \text{offline}_{\text{SIM}}} h_i^* = \sum_{i \notin \text{offline}_{\text{SIM}}} h_i$ , then it uniformly randomly chooses  $h_i^{**}$  for each  $i \in \mathcal{O}_d \setminus \mathcal{C}$  under the constraint that  $\sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} h_i^{**} = \sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} h_i^*$ . The  $h_i^{**}$  for other honest user  $i \notin \text{offline}_{\text{SIM}}$  are randomly chosen such that  $\sum_{i \notin \text{offline}_{\text{SIM}}} h_i^{**} = \sum_{i \notin \text{offline}_{\text{SIM}}} h_i$ . Then  $r_i^*$  is chosen under the constraint that  $\sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} r_i^* = \sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} X_i^* - \sum_{i \in \mathcal{O}_d \setminus \mathcal{C}} (x_i + h_i^{**})$ . In this hybrid,  $\{h_i^*\}$  and  $\{h_i^{**}\}$  have the same distribution. Thus, the distribution of  $r_i^*$  does not change, either.

**Hyb12** In this hybrid, if for some group  $G_d$ , there is no large enough  $\mathcal{O}_d$  with at least  $t$  valid signatures in the view of any honest node in  $G_d$ , the simulator uniformly randomly chooses  $r_i^*$  for honest users  $i$  in group  $G_{d'}$  such that  $\mathcal{O}_{d'}$  exists.

This hybrid is indistinguishable from the previous one, as no information about  $r_i^*$  or  $h_i^*$  for honest  $i \in G_d$  will be revealed to the server by the security of Shamir's secret sharing scheme. Thus, the distribution  $h_i^*$  for  $i \in G_{d'}$  is identical to uniformly random distribution in the server's view.

**Hyb13** Instead of using the inputs  $x_i$  to calculate  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} x_i$ , the simulator queries the ideal functionality by  $w = \text{Ideal}(\mathcal{O}, k)$  if there is a common online set  $\mathcal{O}$  exists in iteration  $k$  and uses  $w$  as the sum.

The distribution of this hybrid is exactly the same as the distribution of the previous hybrid. In this hybrid, the simulator does not know  $x_i$  for any user  $i$ .

Now we have proved that the joint view of  $M_{\mathcal{C}}$  in the real execution is computationally indistinguishable from the view in the simulated execution.  $\square$

**Lemma C.3.** Let  $n, t, n_{\mathcal{C}}, K$  be integer parameters,  $n_{\mathcal{C}} \leq n - t < t$ . Let  $w$  be an element in  $\mathbb{Z}_q$ , and  $w_i \in \mathbb{Z}_q$  for  $i \in [n]$  be shares of  $w$  calculated with  $t$ -out-of- $n$  Shamir secret sharing algorithm, i.e.,  $\{w_i\}_{i \in [n]} \leftarrow \text{SS.share}(w, [n], t)$ .

For each  $k \in [K]$ , let  $w_i^k$  for  $i \in [n - n_{\mathcal{C}}]$  be elements of  $\mathbb{Z}_q$  such that  $w = \text{SS.recon}(\{w_i^k, i\}_{i \in [n - n_{\mathcal{C}}]}, \{w_i, i\}_{i \in [n - n_{\mathcal{C}} + 1, n]}, t)$ . Let  $H(\cdot)$  be a random oracle that returns a random element of  $\mathbb{Z}_p^*$  on each fresh input.

The following two distributions are computationally indistinguishable:

$$w, \{w_i\}_{i \in [n - n_{\mathcal{C}} + 1, n]}, \{H(k)^{w_i}\}_{i \in [n - n_{\mathcal{C}}], k \in [K]} \quad (5)$$

$$w, \{w_i\}_{i \in [n - n_{\mathcal{C}} + 1, n]}, \{H(k)^{w_i^k}\}_{i \in [n - n_{\mathcal{C}}], k \in [K]} \quad (6)$$

*Proof.* We define a hybrid  $\text{Hyb}_0$  to be identical to the distribution (5), and a sequence of hybrids  $\text{Hyb}_k$  for  $k \in [K]$  as following:  $\text{Hyb}_k$  is the same as  $\text{Hyb}_{k-1}$  except that in  $\text{Hyb}_k$ ,  $H(k)^{w_i}_{i \in [n - n_{\mathcal{C}}]}$  are substituted with  $H(k)^{w_i^k}_{i \in [n - n_{\mathcal{C}}]}$ . Thus,  $\text{Hyb}_K$  is identical to distribution (6). Then we prove that any two adjacent hybrids  $\text{Hyb}_{k_0-1}$  and  $\text{Hyb}_{k_0}$  for  $k_0 \in [K]$  are computationally indistinguishable.

For the sake of contradiction, assume there exists a PPT distinguisher

$$\mathcal{D}(w, \{w_i\}_{i \in [n - n_{\mathcal{C}} + 1, n]}, \{Z_i^k\}_{i \in [n - n_{\mathcal{C}}], k \in [K]})$$

which distinguishes between the two distributions. Then, we construct the following distinguisher

$$\mathcal{D}'(A, B_1, \dots, B_{t-n_{\mathcal{C}}-1}, C_1, \dots, C_{t-n_{\mathcal{C}}-1}) :$$

$\mathcal{D}'$  uniformly randomly picks  $\{w_i\}_{i \in [n - n_{\mathcal{C}} + 1, n]}$  as the second part of the input to  $\mathcal{D}$ , and calculates  $W_i = \text{SS.expoRecon}((g^w, 0), \{B_j, j\}_{j \in [t - n_{\mathcal{C}} - 1]}, \{g^{w_j}, j\}_{j \in [n - n_{\mathcal{C}} + 1, n]}, t, i)$  for  $i \in [t - n_{\mathcal{C}}, n - n_{\mathcal{C}}]$ . For  $k \in [K]$  and  $k \neq k_0$  it uniformly randomly picks  $s_k \in \mathbb{Z}_q$ , and sets  $H(k) = g^{s_k}$ .

- For  $k \in [k_0 - 1]$ , it calculates fresh shares  $w_i^k$  of  $w$  such that  $w = \text{SS.recon}(\{w_i^k\}_{i \in [n - n_{\mathcal{C}}]}, \{w_i\}_{i \in [n - n_{\mathcal{C}}]}, t)$  and it sets  $Z_i^k = g^{w_i^k s_k}$  for  $i \in [n - n_{\mathcal{C}}]$ ;
- For  $k \in [k_0 + 1, K]$ , it sets  $Z_i^k = B_i^{s_k}$  for  $i \in [t - n_{\mathcal{C}} - 1]$  and  $Z_i^k = W_i^{s_k}$  for  $i \in [t - n_{\mathcal{C}}, n - n_{\mathcal{C}}]$ ;
- Then it sets  $H(k_0) = A$ ,  $Z_i^{k_0} = C_i$  for  $i \in [t - n_{\mathcal{C}} - 1]$ , and runs

$$Z_j^k = \text{SS.expoRecon}((A^w, 0), \{C_i, i\}_{i \in [t - n_{\mathcal{C}} - 1]}, \{A^{w_i}, i\}_{i \in [n - n_{\mathcal{C}} + 1, n]}, t, j)$$

for  $j \in [t - n_{\mathcal{C}}, n - n_{\mathcal{C}}]$ .

Then it outputs the bit  $\mathcal{D}$  outputs.

When the input to  $\mathcal{D}'$  is from the distribution (1), then distribution of  $\mathcal{D}'$ 's input is identical to  $\text{Hyb}_{k_0-1}$ , and if the input to  $\mathcal{D}'$  is from the distribution (2), then distribution of  $\mathcal{D}'$ 's input is identical to  $\text{Hyb}_{k_0}$ . Thus,  $\mathcal{D}'$  wins with the probability that  $\mathcal{D}$  succeeds. By Lemma B.2, such a distinguisher  $\mathcal{D}'$  does not exist. Thus, we have a contradiction.  $\square$

**Lemma C.4.** *Let  $n, t, n_C, K$  be integer parameters,  $n_C \leq n - t < t$ . Let  $x_1, \dots, x_{n-n_C}$  be uniformly random elements in  $\mathbb{Z}_q$ , and  $\sum_{i \in [n-n_C]} x_i = w$ . Let  $x_{i,j} \in \mathbb{Z}_q$  for  $i \in [n - n_C]$  and  $j \in [n]$  be shares of  $x_i$  calculated with  $t$ -out-of- $n$  Shamir secret sharing algorithm, i.e.,  $\{x_{i,j}\}_{j \in [n]} \leftarrow \text{SS.share}(x_i, [n], t)$  for  $i \in [n - n_C]$ .*

*For each  $k = 1, \dots, K$ , let  $y_1^k, \dots, y_{n-n_C}^k$  also be uniformly randomly chosen from  $\mathbb{Z}_q$  such that  $\sum_{i \in [n-n_C]} y_i^k = w$ . Let  $y_{i,j}^k$  for  $i, j \in [n - n_C]$  be elements of  $\mathbb{Z}_q$  such that*

$$y_i^k = \text{SS.recon}(\{y_{i,j}^k, j\}_{j \in [n-n_C]}, \{x_{i,j}^k, j\}_{j \in [n-n_C+1, n]}, t).$$

*Let  $H(\cdot)$  be a random oracle that returns a random element of  $\mathbb{Z}_p^*$  on each fresh input.*

*Then the following two distributions are computationally indistinguishable if the DDH assumption holds:*

$$w, \{x_{i,j}\}_{i \in [n-n_C], j \in [n-n_C+1, n]}, \{\{H(k)^{x_i}\}_{i \in [n-n_C]}, \{H(k)^{x_{i,j}}\}_{i,j \in [n-n_C]}\}_{k \in [K]} \quad (7)$$

$$w, \{x_{i,j}\}_{i \in [n-n_C], j \in [n-n_C+1, n]}, \{\{H(k)^{y_i^k}\}_{i \in [n-n_C]}, \{H(k)^{y_{i,j}^k}\}_{i,j \in [n-n_C]}\}_{k \in [K]} \quad (8)$$

*Proof.* We prove the indistinguishability between the two distributions by proving that any two adjacent hybrids defined below are computationally indistinguishable:

**Hyb1** It is the same as distribution (7), except that in this hybrid, we calculate  $t$ -out-of- $n$  shares of  $w$  by  $\{w_j\}_{j \in [n]} \leftarrow \text{SS.share}(w, [n], t)$  first, then secret shares  $x_i$  for  $i \in [n - n_C - 1]$  as described in the Lemma. Then, instead of secret sharing  $x_{n-n_C}$ , we calculate  $x_{n-n_C,j} = w_j - \sum_{i \in [n-n_C-1]} x_{i,j}$  for each  $j \in [n]$ .

This hybrid is identical to distribution (7) by the additive homomorphic property of Shamir Secret sharing scheme.

**Hyb2** It is the same as the previous hybrid, except that for each  $k \in [K]$ , we calculate  $w_j^k$  for  $j \in [n - n_C]$  such that  $w = \text{SS.recon}(\{w_j^k, j\}_{j \in [n-C]}, \{w_j, j\}_{j \in [n-n_C+1, n]}, t)$ , and we calculate  $y_{n-n_C,j}^k = w_{n-n_C}^k - \sum_{i \in [n-n_C-1]} x_{i,j}$ . We substitute  $H(k)^{x_{n-n_C}}$  with  $H(k)^{y_{n-n_C}^k}$  and  $H(k)^{x_{n-n_C,j}}$  with  $H(k)^{y_{n-n_C,j}^k}$  for  $j \in [n - C]$ .

By Lemma C.3, This hybrid is indistinguishable from the previous one.

**Hyb3** It is the same as the previous hybrid, except that in this hybrid, for each  $k \in [K]$ , and  $i \in [n - n_C - 1]$ , we choose  $y_i^k$  uniformly at random, calculate  $\{y_{i,j}^k\}_{j \in [n-n_C]}$  such that  $y_i = \text{SS.recon}(\{y_{i,j}^k, j\}_{j \in [n-n_C]}, \{x_{i,j}, j\}_{j \in [n-n_C+1, n]})$ . Then we substitute  $H(k)^{x_i}$  with  $H(k)^{y_i^k}$  and  $H(k)^{x_{i,j}}$  with  $H(k)^{y_{i,j}^k}$  for  $i \in [n - C - 1], j \in [n - C]$ .

By Lemma B.3, This one is indistinguishable from the previous one. This hybrid is also identical to distribution (8).  $\square$

**Lemma C.5.** *Let  $n, B$  be two integer parameters. Let  $x_{i,j}^d$  for  $i, j \in [n]$  and  $d \in [B]$  be uniformly random elements from some finite field  $\mathbb{F}$ . Let  $h_i^d = \sum_{j \in [n]} x_{j,i}^{d-1} - \sum_{j \in [n]} x_{i,j}^d$  for each  $i \in [n]$  and  $d \in [B]$ , in which we define  $x_{j,i}^0 = x_{j,i}^B$  for  $i, j \in [n]$  for convenience. Let  $y_i^d$  for  $i \in [n]$  and  $d \in [B]$  also be uniformly randomly chosen elements in  $\mathbb{F}$  such that  $\sum_{i \in [n], d \in [B]} y_i^d = 0$ . Then the following two distributions are the same:*

$$\{h_i^d\}_{i \in [n], d \in [B]} \quad \text{and} \quad \{y_i^d\}_{i \in [n], d \in [B]}.$$

This lemma can also be easily proved with induction.

## D Handling Larger Input Domain with Class Groups

In this section, we describe another secure aggregation protocol which allows a larger input domain based on class groups as defined in Definition A.1. Let  $\mathbb{G}$  be the group with the DDH assumption and  $\mathbb{F}$  be its subgroup in which discrete logarithm is easy, the core idea is to leverage the subgroup  $\mathbb{F}$  where the discrete logarithm is easy to hide secret inputs while still using the group  $\mathbb{G}$  to hide the masks. Once the sum of masks in the exponent of the generator of the large group is reconstructed and removed, the server only needs to calculate the easy discrete logarithm in the subgroup to recover the sum of secret inputs. One important thing is that the adversary should not know the order of  $\mathbb{G}$ . Otherwise, the DDH assumption no longer holds and the corrupt server can recover information about shares of masks in the exponent. Thus, the masks and the shares cannot be sampled from a finite field but from integers within a bounded interval, as the server who performs reconstruction in the exponent of the generator of  $\mathbb{G}$  does not know the order of group  $\mathbb{G}$ . Therefore, we use the modified version of Shamir secret sharing which works on a bounded integer interval as introduced in Appendix A.1. We present the modified version of MicroFedML<sub>1</sub> while the same technique also applies to the grouped version MicroFedML<sub>2</sub>. As the modified version of Shamir secret sharing on integer interval scales the shares of masks up by  $n!$  in which  $n$  denotes the number of parties the secret is shared with, this method is more suitable for a relatively smaller set of users or the group version in which each secret is only shared within a small group of users.

### D.1 Protocol

The Setup phase is the same as the Setup phase of MicroFedML<sub>1</sub> in Algorithm 1. We describe the Aggregation phase in Algorithm 5. We mark the part of the execution that only needed in malicious settings with red color and underlines.

---

#### Algorithm 5 Aggregation (MicroFedML<sub>3</sub>)

---

This protocol uses the following algorithms defined in Section 3: a Public key infrastructure, a Shamir’s secret sharing scheme (SS.share, SS.recon, SS.expoShare, SS.expoRecon), a hash function  $H(\cdot)$ . It proceeds as follows:

**Input:** Every user  $i$  holds its own signing key  $d_i^{SK}$  and all users’ verification key  $d_j^{PK}$  for  $j \in [n]$ ,  $r_i$ , a list of users  $\mathcal{U}_i$ , and  $r_{j,i}$  for every  $j \in \mathcal{U}_i$  it obtains in the Setup phase. Moreover, it also holds a secret input  $x_i^k$  for every iteration  $k$ . All users also hold a tuple  $(B, N, p, s, f, G, F)$  which is honestly generated by a pair of algorithms (cl.gen, cl.solve) as described in Definition A.1.

The server  $\mathcal{S}$  holds a tuple  $(B, p, s, f, G, F)$  and has access to algorithm cl.solve( $\cdot$ ). It also holds all users’ verification keys, all public parameters it receives in the Setup phase, and a list of users  $\mathcal{U}_{\mathcal{S}}$  which is its output of the Setup phase.

Moreover, in every iteration  $k$ , both all users and the server has access to a fresh random generator  $g_k$  of group  $G$ .

**Output:** For each iteration  $k$ , if there are at least  $t$  users being always online during iteration  $k$ , then at the end of iteration  $k$ , the server  $\mathcal{S}$  outputs  $\sum_{i \in \mathcal{O}^k} x_i^k$ , in which  $\mathcal{O}^k$  denotes a set of users of size at least  $t$ .

**Note:** For simplicity of exposition, we omit the superscript  $k$  of all variables when it can be easily inferred from the context.

1: **for** Iteration  $k = 1, 2, \dots$  **do**

**Round 1: Secret Sharing:**

2:     **User  $i$ :** It calculates and sends  $g_k^{r_i} f^{x_i}$  to the server.

3:     **Server  $\mathcal{S}$ :** Denote the set of users it receives messages from with  $\mathcal{O}$ . If  $|\mathcal{O}| < t$ , abort. Otherwise, it sends  $\mathcal{O}$  to all users  $i \in \mathcal{O}$ .

**Round 2: Online Set Checking (Only needed in Malicious setting):**

4:     **User  $i$ :** On receiving  $\mathcal{O}$  from the server, it first checks that  $\mathcal{O} \subseteq \mathcal{U}_i$  and  $|\mathcal{O}| \geq t$ , then signs the set  $\mathcal{O}$  and sends the signature  $\sigma_i$  to the server.

5:     **Server  $\mathcal{S}$ :** If it receives less than  $t$  valid signatures on  $\mathcal{O}$ , abort. Otherwise, it forwards all valid signatures to all users in  $\mathcal{O}$ .

**Round 3: Mask Reconstruction on the Exponent:**

6:     **User  $i$ :** On receiving signatures from the server, it first verifies the signatures with  $\mathcal{O}$  and the verification keys of the other users. If there are less than  $t$  valid signatures, abort. Otherwise, it calculates  $\zeta_i = g_k^{\sum_{j \in \mathcal{O}} r_{j,i}}$ . It sends  $\zeta_i$  to the server.

7:     **Server  $\mathcal{S}$ :** If it receives  $\zeta_i$  from less than  $t$  users, abort. Otherwise, let  $\mathcal{O}'$  denote the set of users  $i$  successfully sends  $\zeta_i$  to the server. The server reconstructs  $R_{\mathcal{O}} = \text{SS.expoRecon}(\{\zeta_j, j\}_{j \in \mathcal{O}'}, t)$  and calculates

$$\sum_{i \in \mathcal{O}} x_i \leftarrow \text{cl.solve}(B, p, g, f, G, F, g_k^{\sum_{i \in \mathcal{O}} r_i} f^{\sum_{i \in \mathcal{O}} x_i} / R_{\mathcal{O}})]$$

to obtain  $\sum_{i \in \mathcal{O}} x_i$ .  
 8: **end for**

---

## D.2 Privacy

In this section, we discuss the privacy guarantee against malicious adversaries with a compromised server. We give a sketch of the proof with the simulator specification and hybrids which is very similar to those in the proof of Appendix B.2.

The same as in previous protocols, for some iteration  $k$  of the Aggregation phase, We say a user set  $\mathcal{O} \subseteq \mathcal{U}$  is a *common online set* if some honest user receives at least  $t$  valid signatures on  $\mathcal{O}$  in the third round. This set might not exist when the server is corrupt. The Fact B.1 that there is at most one common online set also holds here.

We first define the behavior of a simulator SIM:

- In the Setup phase:
  - **Round 1:** Each honest user  $i$  follows the protocol description in Algorithm 1.
  - **Round 2:** Each honest user  $i$  receives  $(pk_j, \sigma_j)$  from the server, and verifies the signatures as described in Algorithm 1, except that the simulator aborts if some honest user  $i$  receives a valid signature of an honest user  $j$  on a public encryption key different from what user  $j$  sends to the server in the previous round. Then for each corrupt user  $j \in \mathcal{U}_i \cap \mathcal{C}$ , an honest user  $i$  stores  $ek_{i,j} = \text{KA.agree}(pk_j, sk_i)$ . For each pair of honest users  $i, j$ , the simulator uniformly randomly chooses a symmetric encryption key  $ek_{i,j}^*$ , and sets  $ek_{j,i}^* = ek_{i,j}^*$ .  
 Then, for each corrupt user  $j \in \mathcal{U}_i^1 \cap \mathcal{C}$ , user  $i$  uniformly randomly chooses  $r_{i,j}$ , encrypts it by  $c_{i,j} \leftarrow \text{AE.enc}(r_{i,j}, ek_{i,j})$ ; for each honest user  $j \in \mathcal{U}_i^1 \setminus \mathcal{C}$ , user  $i$  encrypts a dummy value by  $c_{i,j} \leftarrow \text{AE.enc}(\perp, ek_{i,j}^*)$ . Each honest user  $i$  sends  $\{c_{i,j}\}_{j \in \mathcal{U}_i}$  to the server.
  - **Users Receiving Shares:** For each honest user  $i$ , on receiving  $c_{j,i}$  from the server, each honest user  $i$  follows the protocol except that it additionally aborts if for any honest  $j$ , the decryption succeeds while the result is different from the value user  $j$  encrypts in the previous round.
- In the  $k$ -th iteration of the Aggregation phase:
  - **Round 1:** Each honest user  $i$  uniformly randomly chooses  $X_i^*$  and sends  $g_k^{X_i^*}$  to the server.
  - **Round 2:** Each honest user  $i$  follows the protocol described in Algorithm 5.
  - **Round 3:** If any honest user receives at least  $t$  valid signatures on the online set  $\mathcal{O}$  it receives in the previous round, the simulator queries the ideal functionality to get  $w = \text{Ideal}(\mathcal{O} \setminus \mathcal{C}, k)$ . Then for each  $i \in \mathcal{O} \setminus \mathcal{C}$ , the simulator uniformly randomly samples  $w_i^*$  for  $i \in \mathcal{O} \setminus \mathcal{C}$  under the restriction  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} w_i^* = w$ . Then for all honest users  $i \in \mathcal{O}_i \setminus \mathcal{C}$ , the simulator uniformly randomly samples  $w_i^*$  for  $i \in \mathcal{O}_i \setminus \mathcal{C}$  under the restriction  $\sum_{i \in \mathcal{O}_i \setminus \mathcal{C}} w_i^* = w$ . Moreover, the simulator SIM calculates the shares  $R_{i,j}^*$  for all honest users in the online set  $j \in \mathcal{O}_i \setminus \mathcal{C}$  such that  $\text{SS.expoRecon}(\{R_{i,j}^*, j\}_{j \in \mathcal{O}_i \setminus \mathcal{C}}, \{g_k^{r_{i,j}}\}_{j \in \mathcal{C}}) = g_k^{X_i^*} / f^{w_i^*}$  where  $r_{i,j}$  for  $j \in \mathcal{C}$  are the shares that have already been sent to the corrupt users in the Setup phase. Let  $R_{j,i}^* = g_k^{r_{j,i}}$  for  $j \in \mathcal{C}$  and honest user  $i$ . We refer the readers to Section A.1 for the implementation of this part.  
 Then for the honest users  $i$  who receives  $\mathcal{O}$  with at least  $t$  valid signatures from the server in the second round, the simulator sends  $\zeta_i^* = \prod_{j \in \mathcal{O}_i} R_{j,i}^*$  to the server on behalf of user  $i$ .

By Fact B.1, there will be at most one unique set  $\mathcal{O}$  that collects enough number of valid signatures and the Ideal functionality will be queried at most once each iteration.

We describe a series of hybrids between the joint view of corrupt parties in the real execution and the output of the simulation described above. Each hybrid is identical to the previous one except the part explicitly described. By proving that each hybrid is computationally indistinguishable from the previous one, we prove that the joint view of the corrupt parties in the real execution is indistinguishable from the simulation.

**Hyb0** This random variable is the joint view of all parties in  $\mathcal{C}$  in the real execution.

**Hyb1** In this hybrid, a simulator which knows all secret inputs of honest parties in every iteration simulates the execution with  $M_C$  following the protocol.

The distribution of this hybrid is exactly the same as the previous one.

- Hyb2** In this hybrid, the simulator aborts if  $M_C$  provides any of the honest parties  $j$  with a valid signature with respect to an honest user  $i$ 's public key  $d_i^{PK}$  on  $pk_i^*$  different from what  $i$  provides.  
The indistinguishability between this hybrid and the previous one is guaranteed by the security of the signature scheme.
- Hyb3** In this hybrid, for any pair of two honest users  $i, j$ , the encryption of shares  $c_{i,j}$  and  $c_{j,i}$  they send between each other are encrypted and decrypted using a uniformly random key  $ek_{i,j}^*$  instead of  $ek_{i,j}$  obtained through Diffie Hellman key exchange in Setup Phase.  
The indistinguishability between this hybrid and the previous one is guaranteed by 2ODH assumption.
- Hyb4** In this hybrid, we substitute each encrypted share  $c_{i,j}^r = \text{AE.enc}(ek_{i,j}^*, r_{i,j})$  sent between honest parties in the Setup phase in the previous hybrid with the encryption of a dummy value, i.e.,  $c_{i,j}^{r*} = \text{AE.enc}(\perp, ek_{i,j}^*)$ .  
The indistinguishability is guaranteed by IND-CPA security of the encryption scheme.
- Hyb5** In this hybrid, in every iteration  $k$ , each honest user  $i$  substitutes  $g_k^{r_i} f^{x_i}$  it sends to the server in the first round with  $g_k^{X_i^*}$  for a uniformly randomly chosen  $X_i^*$ . Moreover, in the third round, for each honest user  $i$ , SIM calculates  $R_i^* = g_k^{X_i^*} / f^{x_i}$  the shares  $R_{i,j}^*$  for honest users  $j$  based on the shares which have already been sent to corrupt users in the Setup phase, i.e., it calculates  $R_{i,j}^*$  for  $j \in \mathcal{U} \setminus \mathcal{C}$  making sure that  $R_i^* = \text{SS.expoRecon}(\{R_{i,j}^*, j\}_{j \in \mathcal{U} \setminus \mathcal{C}}, \{g_k^{r_{i,j}}, j\}_{j \in \mathcal{C}})$ . For corrupt users  $j \in \mathcal{C}$ , let  $R_{j,i}^* = g_k^{r_{j,i}}$ . Then each honest user  $i$  who receives  $\mathcal{O}$  with at least  $t$  valid signatures calculates  $\zeta_i^* = \prod_{j \in \mathcal{O}} R_{j,i}^*$  and sends  $\zeta_i^*$  to the server.  
The proof of this hybrid is indistinguishable from the previous one follows the same reasoning as the proof of Lemma B.4.
- Hyb6** In this hybrid, in each iteration, if some honest user receives  $\mathcal{O}$  with at least  $t$  valid signatures in the second round, then in the third round, for each user  $i \in \mathcal{O} \setminus \mathcal{C}$ , instead of using  $R_i^* = g_k^{X_i^*} / f^{x_i}$  when calculating the shares for honest users, SIM randomly picks  $R_i^*$  under the constraint that  $\prod_{i \in \mathcal{O} \setminus \mathcal{C}} R_i^* = \prod_{i \in \mathcal{O} \setminus \mathcal{C}} g_k^{X_i^*} / f^{x_i}$ . The simulator then uses  $R_i^*$  to calculate the shares  $\{R_{i,j}^*\}_{j \in \mathcal{U} \setminus \mathcal{C}}$  for user  $i \in \mathcal{O} \setminus \mathcal{C}$ .  
This hybrid is indistinguishable from the previous hybrid, as  $R_i^*$  are still uniformly random, and  $\prod_{i \in \mathcal{O} \setminus \mathcal{C}} R_i^*$  that the server can reconstruct from the shares keeps the same.
- Hyb7** In this hybrid, in each iteration, if some honest user receives  $\mathcal{O}$  with at least  $t$  valid signatures in the second round, then in the third round, for each honest user  $i \notin \mathcal{O}$ , the simulator sets  $R_i^*$  as  $g_k$  and calculates the shares for  $i$  and  $j \in \mathcal{U} \setminus \mathcal{C}$ .  
This hybrid is identical to the previous one as there is only one unique  $\mathcal{O}$  and if an honest user is not included in  $\mathcal{O}$ , the share  $R_{i,j}^*$  for  $j \in \mathcal{U} \setminus \mathcal{C}$  will not be included in any  $\zeta_j^*$  sent to server. Thus, the adversary will not receive any information about  $R_i^*$  in the third round of the iteration.
- Hyb8** Instead of receiving the inputs from the honest parties and using  $\sum_{i \in \mathcal{O} \setminus \mathcal{C}} x_i$  to sample  $R_i^*$  for  $i \in \mathcal{O} \setminus \mathcal{C}$ , the simulator makes a query to the functionality  $\text{Ideal}$  with the user set  $\mathcal{O} \setminus \mathcal{C}$  and iteration counter  $k$  and use the output value to sample random value  $R_i^*$  in every iteration with  $|\mathcal{O} \setminus \mathcal{C}| \geq t - n_C$ . Note that the  $\text{Ideal}$  functionality will not return  $\perp$  in this case.  
The distribution of this hybrid is exactly the same as the distribution of the previous hybrid. In this hybrid, the simulator does not know  $x_i$  for any user  $i$ .

Now we have proved that the joint view of  $M_C$  in the real execution is computationally indistinguishable from the view in the simulated execution.

## E Related Works

**Secure aggregation** There are several other works exploring the secure aggregation problem. Liu et al. propose a privacy preserving federated learning scheme for XGBoost in Liu et al. (2020). However, it does not allow offline nodes to rejoin the training process later without sacrificing privacy. Several recent works also employ the idea of reconstructing one layer of mask of online users. Yang et al. proposes a secure aggregation protocol LightSecAgg Yang et al. (2021b) in which each user chooses a local mask, shares an encoding of it first, then it sends the input covered with the mask to the server, and sends the server the aggregated value of masks of the online users to the server so that the server can decode the aggregated mask from the sum of the masked inputs. The authors also discuss secure aggregation solution in asynchronous federated learning which allows the stale updates from slow users to also contribute in learning tasks. In SAFElearn Fereidooni et al. (2021) proposed by Fereidooni et al., each user the encryption of its local update

encrypted with fully homomorphic encryption (FHE) to the server who only performs the aggregation computation on the cipher text when there is only one server available, or shares its update among more than one non-colluding servers who collaboratively calculate the aggregation of the model updates with multiparty computation (MPC) or secure two-party computation (STPC). However, both of these works consider only semi-honest adversary model and the users also need to generate and share the random masks in every iteration of aggregation.

**Differential Privacy** Another line of works adopt differential privacy which is a generic privacy protection technique in database and machine learning area. The high level idea is to add artificial noises to the gradients to prevent inverting attack without losing too much accuracy. Applying differential privacy technique in federated learning is more challenging than in traditional machine learning scenario, as in federated learning every single user needs to add the noises by its own. The individual noise should not be either too weak to lose the functionality of hiding the data, or too strong to radically harm the accuracy of the learning result. Truex et al. propose a hybrid approach Truex et al. (2019) which protects the privacy during learning process with secure multiparty computation and prevents inference over the outputs of learning with differential privacy. This work assumes all clients are online. HybridAlpha proposed by Xu et al. in Xu et al. (2019) also adopts both differential privacy and functional encryption. It assumes honest but curious server and dishonest users.

**Quantization, gradient sparsification, and weight regularization** Both *quantization* and *gradient sparsification* are methods commonly used to reduce the cost of communicating gradients between nodes in the scenario of data-parallel Stochastic Gradient Descent (SGD). There is a collection of works Lin et al. (2020); Amiri and Gunduz (2020a,b); Byrd et al. (2019); Sattler et al. (2018); Malekijoo et al. (2021); Basu et al. (2020); Yang et al. (2021a); Nori et al. (2021); Cui et al. (2021); Chen et al. (2021); Horvath et al. (2020); Aji and Heafield (2017); Alistarh et al. (2017); Chen et al. (2018); Wen et al. (2017); Wu et al. (2018); Haddadpour et al. (2019); Sattler et al. (2020); Wang et al. (2018) proposing methods for quantization, gradient compression and sparsification as well as clustering leading to smaller weights. Our secure aggregation protocols, suitable to smaller weights, can be used to execute the above methods in a privacy-preserving way for federated learning setting. Moreover, weight regularization Van Laarhoven (2017); Krogh and Hertz (1991) is a widely used technique to reduce overfitting by keeping the weights of the model small.

## F ABIDES Framework

We use a discrete event simulation framework ABIDES Byrd et al. (2019) to implement the simulation of our protocol. In this section, we give an overview of this framework.

### F.1 Creating Parties for a Protocol

Within the context of a discrete event simulation, any actor which can affect the state of the system is generically called an *agent*. In ABIDES, all agents inherit (in the sense of object-oriented programming) from the base `Agent` class. This class provides a minimal implementation for the methods required to properly interact with the simulation kernel. It is expected that experimental agents will override those methods which require non-default behavior.

Each party in a cryptographic protocol will therefore be an instance of some subclass of `Agent`, customized to contain that agent’s portion of the protocol. When a protocol calls for multiple parties of the same type, only one specialized agent class must be created, with the relevant parties each being a distinct instance during the simulation, with potentially different timing, randomness, and attributes.

The following subsections briefly describe these minimum required methods. Note that agents may additionally contain any other arbitrary methods as required for their protocol participation.

#### F.1.1 Methods Called Once per Agent

The `kernelInitializing` method is called after all agents have been created. It gives each agent a reference to the simulation kernel. This method is a good place to conduct any necessary agent initialization that could not be handled in the agent’s `__init__` method for some reason, for example if it required interaction with the kernel.

The `kernelStarting` method is called just before simulated time begins to flow. It tells each agent the starting simulated time. Agents that need to take action at the beginning of the simulation, without being prompted by a message from another agent, should use this event to schedule a wakeup call using `setWakeup`. Otherwise, the agent may never act.

The `kernelStopping` method is called just after simulated time has ended, to let each agent know that no more messages will be delivered. This is a good place to compute statistics and write logs.



The `kernelTerminating` method is called just before the simulation kernel exits. It allows a final chance for each agent to release memory or otherwise clean up its resources.

### F.1.2 Methods Called Many Times per Agent

The `wakeup` method is called by the kernel when this agent had previously requested to be activated at a specific simulated time. The agent is given the current time, but it is otherwise expected the agent will have retained any required information in its internal state. An agent can request a wakeup call with `setWakeup`.

The `receiveMessage` method is called when a communication has arrived from another agent. The agent is given the current time and an instance of the `Message` class. The kernel imposes no particular constraint on the contents of a message. It is up to the agents in a simulation to interpret the messages they may receive. Most of the existing messages simply hold a Python dictionary in `Message.body` that contains key-value pairs, varying with message type. An agent can transmit a message with the `sendMessage` method, and computation or latency delays will be automatically added by the kernel during delivery scheduling.

### F.1.3 Example: Shared Sum Protocol

While there may be a server agent in a client-server protocol, it is important to understand that there is no “one place” to write protocol logic in a linear fashion. Just as in the real world, progress through the protocol will be driven by individual agent actions, and each agent must constantly work out where it stands in the protocol and what it should do next.

For example, imagine a simple multi-party computation (MPC) protocol to securely compute a shared sum. There will need to be two agent classes created, because a client party and a computation service will behave quite differently. We might call them `SumClient` and `SumService`. Both will inherit from the basic `Agent` class.

**The Central View** Thinking centrally, we could write English instructions for a simple shared sum protocol using MPC:

1. Each party  $i$  should send to each other party  $j$  a randomly generated large number  $n_{ij}$ .
2. Each party  $i$  should calculate and retain  $s_i^{out} = \sum_j n_{ij}$ .
3. After receiving a message  $n_{ji}$  from all other parties  $j$ , each party  $i$  should compute  $s_i^{in} = \sum_j n_{ji}$ .
4. Each party  $i$  should send to the summation service encrypted operand value  $V_i = v_i - s_i^{out} + s_i^{in}$ , where  $v_i$  is the cleartext value of its operand.
5. After receiving a message containing operand  $V_i$  from all client parties  $i$ , the service should compute result  $R = \sum_i V_i$ , and send messages containing result  $R$  to all parties  $i$ .

All parties will now have an accurate summation result despite the summation service receiving encrypted operands and being unable to reveal any party’s cleartext operand.

**Summary of Distributed Implementation** But how will we implement the above protocol in a multi agent discrete event simulation without “central logic”? We will need to carefully control the flow of the simulation through individual agent actions and internal agent state. The client parties require code for Steps 1-4 of the protocol and the summation service requires code for Step 5.

**SumClient.kernelStarting** will need to request an initial wakeup call for this client party at, or shortly after, the given `start_time`, which will be the earliest possible simulated timestamp. This can be done by calling `self.setWakeup`.

**SumClient.\_\_init\_\_** will need to receive a list of peer party ids within the same connected subgraph and store this in an instance variable for later use. This list is necessary to send shared secrets to peers, and to know when all “expected” shared secrets have been received from peers.

**SumClient.wakeup** will need to implement Steps 1 and 2 of the protocol. The protocol must begin with wakeup calls to the agents, because there is not yet any message flow to trigger party activities. To send the shared secrets, the party will call `self.sendMessage` once per peer client discovered during `__init__`. The body of a message is typically a Python dictionary, so we can set `Message.body['type'] = 'SHARED_SECRET'` and `Message.body['secret']` to the randomly generated value for a given peer. The sent shared secrets can be accumulated into an instance variable for later use, for example as `self.sent_sum`.

**SumClient.receiveMessage(msg)** will need to implement steps 3 and 4 of the protocol, because this phase is triggered by receipt of messages from other parties. The client party can test `msg.body['type']` to determine what kind of message has roused it. Upon receiving a `SHARED_SECRET` message, the party should accumulate the secret value and a count of received values into instance variables, for example as `self.received_sum` and `self.received_count`. There is no outside signal to tell a party when it has received the final shared secret, so at receipt of each secret, the party must compare its received count to the known size of its peer network. When the final secret has arrived and been accumulated, the party will call `sendMessage` one time with the id of the summation service and set `Message.body['type'] = 'SUM_REQUEST'` and `Message.body['value']` to the encrypted operand value, which is the cleartext operand value plus the sum of received secrets minus the sum of sent secrets.

**SumService.\_\_init\_\_** will need to receive a count of client parties from whom it should expect summation requests, and store this in an instance variable, for example as `self.num_clients`.

**SumService.receiveMessage(msg)** will need to implement step 5 of the protocol, because it is triggered by receipt of messages from client parties. Note that there is no need for a non-default implementation of **SumService.wakeup**, because the service does nothing until it receives client requests. Each time a `SUM_REQUEST` message is received, the service must store as instance variables the received operand values, the clients from which they were received, and a count of received values. Once the service has received the expected number of `SUM_REQUEST` messages, it can sum the operands to a single result and call `self.sendMessage` once per communicating client party to deliver the result in an appropriate message type, perhaps `SUM_RESULT`.

If the client parties should do something with the summation result, `SumClient.receiveMessage(msg)` is the appropriate location for that code. Note that the client party must distinguish incoming `SUM_RESULT` messages from `SHARED_SECRET` messages by testing `msg.body['type']`.

## F.2 Connecting Parties in a Protocol

For a multi agent discrete event simulation to be useful, the parties must be able to exchange messages. For the simulation to be realistic, those messages should experience variable, non-zero communication latency or *time in flight*, and various parties should be able to have different latency characteristics.

The ABIDES framework supports this through the `model.LatencyModel` class, which defines a (potentially) fully-connected pairwise network among the agents in a simulation, or the parties in a protocol. Once defined, the model will be automatically applied to all messages within the simulated environment. The preferred latency model is currently the 'cubic' model.

The cubic latency model accepts up to five parameters: `connected`, `min_latency`, `jitter`, `jitter_clip`, and `jitter_unit`. Only the parameter `min_latency` is required. The others have reasonable default values.

In brief, `min_latency` must be a 2-D numpy array defining the minimum latency in nanoseconds between each pair of agents. The matrix can be diagonally symmetric if communication speed should be independent of communication direction, but this is not required. The `connected` parameter must be either `True` (all parties are pairwise connected) or a 2-D boolean numpy array denoting connectivity. Parties that are not connected will be prohibited from calling `sendMessage` with each other's id. The remaining parameters describe the cubic randomness added to the minimum latency when each message is scheduled for delivery. Detailed documentation is contained in the docstring at the top of the `LatencyModel` class code.

## F.3 Realistic Computation Delays within a Protocol

Reasonable estimation of computation time is another important piece of a realistic simulation. The ABIDES framework supports a per-party computation delay that represents how long the party requires to complete a task and generate resulting messages. This delay will be used both to determine the "sent time" for any messages originated during the activity and the next available time at which the party could act again. Computation delays are stored in a 1-D numpy array with nanosecond precision.

A specific party (simulation agent) has only one computation delay value at a time, but these values can be updated at any time. We can therefore observe the actual computation time of the activity as it happens in the simulation, and use this to set the appropriate delay in simulated time.

A straightforward way to handle this is to assign `pandas.Timestamp('now')` to a variable when the activity begins, and subtract it from a second call to `pandas.Timestamp('now')` when the activity ends. The difference between these two can be passed to `self.setComputationDelay` to update the party's computation cost for the current activity.

The same technique can be used to accumulate time spent by a party in various sections of the protocol, so aggregated statistics can be logged or displayed at the conclusion of the protocol.