MicroSecAgg: Streamlined Single-Server Secure Aggregation

Yue Guo
J.P. Morgan AI Research &
AlgoCRYPT CoE
yue.guo@jpmchase.com

Antigoni Polychroniadou
J.P. Morgan AI Research &
AlgoCRYPT CoE
antigoni.polychroniadou@jpmorgan.com

Elaine Shi
Carnegie Mellon University
runting@gmail.com

David Byrd
Bowdoin College
d.byrd@bowdoin.edu

Tucker Balch
J.P. Morgan AI Research
tucker.balch@jpmchase.com

ABSTRACT

This work introduces MicroSecAgg, a framework that addresses the intricacies of secure aggregation in the single-server landscape, specifically tailored to situations where distributed trust among multiple non-colluding servers presents challenges. Our protocols are purpose-built to handle situations featuring multiple successive aggregation phases among a dynamic pool of clients who can drop out during the aggregation. Our different protocols thrive in three distinct cases: firstly, secure aggregation within a small input domain; secondly, secure aggregation within a large input domain; and finally, facilitating federated learning for the cases where moderately sized models are considered. Compared to the prior works of Bonawitz et al. (CCS 2017), Bell et al. (CCS 2020), and the recent work of Ma et al. (S&P 2023), our approach significantly reduces the overheads. In particular, MicroSecAgg halves the round complexity to just 3 rounds, thereby offering substantial improvements in communication cost efficiency. Notably, it outperforms Ma et al. by a factor of n on the user side, where n represents the number of users. Furthermore, in MicroSecAgg the computation complexity of each aggregation per user exhibits a logarithmic growth with respect to n, contrasting with the linearithmic or quadratic growth observed in Ma et al. and Bonawitz et al., respectively. We also require linear (in n) computation work from the server as opposed to quadratic in Bonawitz et al., or linearithmic in Ma et al. and Bell et al. In the realm of federated learning, a delicate tradeoff comes into play: our protocols shine brighter as the number of participating parties increases, yet they exhibit diminishing computational efficiency as the sheer volume of weights/parameters increases significantly.

We report an implementation of our system and compare the performance against prior works, demonstrating that MicroSecAgg significantly reduces the computational burden and the message size.

1 INTRODUCTION

In the realm of secure aggregation protocols, a group of n users P_i for $i \in [n]$, each holding a private value x_i, wish to learn the sum $\sum_i x_i$ in collaboration with one or more server(s) without leaking any information about the individual x_i. Diverging from the conventional multi-party computation setup that relies on point-to-point channels, communication in this context is confined to interactions between individual clients and the server(s). Consequently, direct inter-client communication is precluded, necessitating clients to exclusively engage with the intermediary server(s) for communication.

Versatile in their scope, secure aggregation protocols find relevance across a spectrum of domains, including secure voting, preserving privacy in browser telemetry [2, 24, 25, 41], federated learning [10, 11, 37] and driving analytics for digital contact tracing [8]. Unlike the multi-server setting of [2, 24, 25, 41], our paper centers its focus on the single-server context where multiple users can drop out during the execution of the protocol. This setting emerges as the preferred option when grappling with the complexities of entrusting multiple non-colluding entities with mutual trust. On the flip side, this scenario presents greater challenges. Unlike the multi-server scenario where parties can secret share their inputs among servers for aggregation, the single-server setting lacks this option due to the presence of only one server. The single-server model also comes as the preferred model in privacy-preserving federated learning introduced by [11]. Secure aggregation protocols designed to withstand dropout parties within the single server setting have solely emerged within the realm of federated learning [10, 11, 37]. However, the works of [10, 11] suffer from some inefficiencies. More specifically, in every aggregation (learning) iteration the protocols in [10, 11] reveal part of the masks used to protect the inputs from the server. That said, the masks cannot be reused to mask the inputs in later 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 [11] or a subset of users in [10]) 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 is 5 in [11] and 6 in [10].

This paper presents a novel overarching framework for secure aggregation within the single-server context. Our protocols are specifically designed to suit scenarios in which multiple consecutive aggregation phases occur within a pool of clients. Even in cases where a fraction of clients may dropout during certain phases, our protocols remain resilient. At the heart of our methodology lies the initiation of a secret sharing configuration among the clients —

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–30

https://doi.org/XXXXXXX.XXXXXXX

an initial setup executed only once yet seamlessly reused throughout all subsequent aggregation phases. This results in a notable reduction in both round complexity and communication complexity compared to previous approaches. Numerous situations involving a group of clients demand recurrent aggregation sessions spanning a duration, such as browser telemetry, contact tracing, federated learning and IoT devices periodically gathering area data to understand daily usage patterns.

The subsequent works, Flamingo and Lerna [32, 37] borrow our idea of reusable setup and reduce the round complexity of [10] while relying on a special groups of users to help with the masking process.

1.1 Our Results

We propose secure aggregation protocols in which the server and the users interact several times (multi-iteration) to compute multiple sums. Our novelty is the use of two phases, the Setup phase and the Aggregation phase. The Setup phase is independent of the user private inputs and runs only once at the beginning of the execution. We model an adversary that 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 γ fraction of users which it decides to corrupt before each protocol execution, and that at most δ fraction of users are offline in every iteration. We assume a static adversary (Flamingo does too) since the neighborhoods per iteration are fixed in the setup phase.

More specifically, we propose three multi-iteration secure aggregation protocols, MicroSecAgg$_{DL}$, MicroSecAgg$_{gDL}$ and MicroSecAgg$_{CL}$ which are applicable across three aggregation scenarios:

- **Small Input Domain Aggregation.** Our aggregation protocols MicroSecAgg$_{DL}$ and MicroSecAgg$_{gDL}$ (a version which uses groups of neighbour users a la Bell et. al [10]) involve the consolidation and analysis of data with a restricted range of possible values. This concept is crucial in scenarios where data falls within a constrained input domain, offering benefits in terms of computational efficiency and privacy preservation. In various domains, especially in data analysis and machine learning, the concept of Small Input Domain aggregation finds significance. One common scenario where Small Input Domain aggregation is employed is in the aggregation of survey responses. For example, to calculate the average rating of merchandise, the aggregation is usually performed on a small domain, e.g., the rating from 1 to 5 or 1 to 10. Other applications include voting, browser telemetry, contact tracing, and gradient sparsification [3–6, 23, 33, 38, 42], quantization, and weight regularization areas [17, 26, 31, 46, 49] in federated learning. 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 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 settings. Moreover, weight regularization [31, 46] is a widely used technique to reduce overfitting by keeping the weights of the model small.
  For the small input domain scenarios our algorithm is based on general cyclic groups based on the Decisional Diffie–Hellman (DDH) assumption. It is not efficient for large input domain because the server needs to compute discrete logarithm to recover the final sum.
- **Large Input Domain aggregation.** This type of aggregation is tailored to handle the amalgamation of data stemming from a vast range of possible inputs. Our algorithm MicroSecAgg$_{CL}$ for large input domain is based on class groups of unknown order which allow us to avoid the computation of discrete logarithm. Since the order of group with class groups needs to stay unknown, the protocol relies on secret sharing over an integer interval which involves computation over large integers containing $n!$ where n is the number of shares generated. That said, as the current computation architectures do not have very efficient support for computation over large integers, this solution is more suitable for the group version (using the same technique as MicroSecAgg$_{gDL}$) with each group containing $O(\log n)$ users where n is the total number of users as the secret sharing only happens within each group.
- **Aggregation for Federated Learning for moderate sized models.** Our protocol can be used in federated learning and considerably reduce the communication and computation overheads of [10, 11] with reusable setup as mentioned above while achieving the same provable security guarantees provided by [10, 11]. Our protocol demonstrates better performance as the user count increases. However, it excels particularly with moderately sized models that possess fewer weights, as the number of exponentiations in our approach scales with the length of weight/input vectors. The subsequent works of [32, 37] do not face this caveat.

In Tables 1 and 2 we list the communication and computational complexity of our protocols compared to prior work, respectively. Notable improvements of MicroSecAgg over previous single server aggregation protocols BIK+17, BBG+20, and Flamingo [10, 11, 37] include the following. MicroSecAgg halves the round complexity to 3 rounds and thus improving the communication cost further. It also improves the communication cost over Flamingo by a factor of n where n is the number of users. Moreover, the computation complexity of each aggregation for each user increases logarithmically with n in MicroSecAgg$_{CL}$ while that of [10, 11, 37] both grow linearithmic or quadratically in n. The full asymptotic computation and communication cost comparison is included in Table 2, We refer the reader to Appendix E.1 and E.2 for the detailed analysis of the asymptotic performance.

Standard vs. group version of our protocols. MicroSecAgg$_{gDL}$ and the group version MicroSecAgg$_{CL}$ are more suitable for a larger number of parties and for use cases where weaker security guarantees are sufficient. For example, the adversary cannot adaptively corrupt all parties in a single group neighborhood. If such an event happens the privacy is lost. The group version in [10] also cannot support adaptive corruptions and has a weaker security definition: for some parameter α between $[0, 1]$, honest inputs are guaranteed to be aggregated at most once with at least α fraction of other inputs from honest users. Flamingo [37] also uses groups of neighbours and additionally relies on a special group of users called decryptors to do partial decryption of masks used to hide user's input. That said, Flamingo also needs to accommodate additional corruption of the decryptors (additionally to the corruption of users and neighbhoods).

We summarize our results in the following two (informal) theorems:

Theorem 1.1. *The aggregation protocol MicroSecAgg$_{DL}$ running with a server $\mathcal{S}$ 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 γ, δ be two parameters such that $\gamma + 2\delta < 1$. The aggregation protocol MicroSecAgg$_{gDL}$ (and MicroSecAgg$_{CL}$) running with a server $\mathcal{S}$ 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 δ fraction of the users.*

Note: the requirement $\gamma + 2\delta < 1$ is required for the proof to go through. As in this protocol we divide all users into small groups, this condition is needed to guarantee that each group only has too many offline or corrupt users with negligible probability. It is the same requirement as in [10]. The detailed analysis can be found in Appendix C.1.

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 a logistic regression algorithm on the Census adult dataset [27] and compare the learning result with plain federated learning in which the users send the plain text of model update to the server. The two experiments provide models with the same accuracy (0.81).

Implementation and evaluation. We implemented the proposed system and report performance in Section 6. Our protocol MicroSecAgg$_{DL}$ outperforms BIK+17 [11] by 100 times in computation time with 500 total participants, while MicroSecAgg$_{gDL}$ runs more than 5 times faster than BBG+20 [10] when the connectivity of the user communication graph is 100 and the total number of users is 500. For 1000 participants, MicroSecAgg$_{gDL}$ is 5 times faster than MicroSecAgg$_{DL}$. 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, MicroSecAgg$_{CL}$ with class group is 10 times faster than Flamingo on both user and server side.

1.2 Related Works

Our work is inspired by the line of works on secure aggregation protocols [10, 11]. Federated learning empowers users, often with limited resources, to collectively improve a global model guided by a central server. The raw data remains private. In each iteration, the central server shares the model with users, who refine it using local data. These updates are merged by the server to enhance the global model, which is then redistributed. However, this approach can risk individual user's privacy due to trained models and local updates can leak information about training data [40]. To tackle this problem, Bonawitz et al. [11] introduced a secure aggregation protocol, letting users mask their local updates, preserving privacy for individual updates except for the aggregate result. Bell et al. [10] later reduced bandwidth costs, increasing round complexity and guaranteeing probabilistic correctness i.e., when there are enough number of honest users online, the server learns the correct aggregated result with overwhelming probability.

There are several other works [28, 34, 36, 48] exploring the secure aggregation problem. However, all of them either only consider semi-honest adversaries or do not allow offline users to come back online again. Another line of works [44, 47] adopt differential privacy which is a generic privacy protection technique in database and machine learning areas.

The work, ACORN, by Bell et al. [9] follows the same line of work [10, 11] and introduces input verifiability and robustness to the secure aggregation protocol. However, as it does not give the reusability of the secret masks, ACORN still suffers from higher round complexity and communication costs. Our concept of reusability has the potential to enhance the complexity of their work while maintaining input verification functionality. Yet, our exploration revealed multiple challenges in merging the two approaches, making it a complex task. As such, we acknowledge this as an area for future investigation and development.

Our protocol can also be combined with other secure aggregation solutions. A recent work by Liu et al. [35] 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 MicroSecAgg as MicroSecAgg is more efficient than the protocol [10, 11] 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 introduced in [44] can also be applied to our protocol. See Appendix F for more discussion.

2 OUR APPROACH

In this section, we explain the high-level idea of our constructions.

2.1 Revisiting BIK+17

We first revisit the idea of BIK+17 [11] which sprouts from the following simple idea: to let the server learn the sum of the inputs

Round	Communication cost per user (bits)					
	BIK+17	BBG+20	Flamingo	MicroSecAgg$_{DL}$	MicroSecAgg$_{gDL}$	MicroSecAgg$_{CL}$
1	$O(n\kappa)$	$\underline{O(\kappa)}$	$O(L\ell + n\kappa)$	$O(\kappa L)+n$	$O(\kappa L)+O(\log n)$	$O(\kappa L)+O(\log n)$
2	$O(n\kappa)$	$O(\kappa \log n)$	$\underline{O(\kappa \log n)}$	$\underline{O(n\kappa)}$	$\underline{O(\kappa \log n)}$	$\underline{O(\kappa \log n)}$
3	$O(L\ell)+n$	$O(\kappa \log n)$	$O(n\kappa \log n)$	$O(\kappa L)$	$O(\kappa L)$	$O(\kappa L)$
4	$\underline{O(n\kappa)}$	$O(L\ell + \log n)$				
5	$O(n\kappa)$	$\underline{O(\kappa \log n)}$				
6		$O(\kappa \log n)$				

Table 1: Communication overhead per user in every round of each aggregation protocol with semi-honest/malicious security (the extra rounds required for privacy in the malicious setting is marked as blue and underlined in the table). κ denotes the security parameter, n the total number of users, L the vector length of the input vector and ℓ the bit-length of each element in the input vector. The overhead includes both sent and received messages per user. The cost refers to field elements with high constant coefficients as opposed to the teal colored costs which indicate bit vectors.

Protocol	round	Computation cost				Communication cost			
		Server		User		Server		User	
		Setup	Agg.	Setup	Agg.	Setup	Agg.	Setup	Agg.
BIK+17	– | 5	–	$O(n^2L)$	–	$O(n^2 + nL)$	–	$O(nL\ell + n^2\kappa)$	–	$O(L\ell + n\kappa)$
BBG+20	– | 6	–	$O(n\log^2 n + nL\log n)$	–	$O(\log^2 n + L\log n)$	–	$O(nL\ell + n\kappa\log n)$	–	$O(L\ell + \kappa\log n)$
Flamingo	5 | 3	forwarding	$O(nL + n\log^2 n)$	$O(\log^2 n)$	$O(L + n\log n)$	$O(\kappa\log^3 n)$	$O(nL\ell + n\kappa\log^2 n)$	$O(\kappa\log^2 n)$	$O(L\ell + n\kappa\log n)$
MicroSecAgg$_{DL}$	3 | 3	$O(n)$	$O(nL + 2^{\ell/2}L)$	$O(n^2)$	$O(nL)$	$O(n^2\kappa)$	$O(n\kappa(L+n))$	$O(n\kappa)$	$O(\kappa(L+n))$
MicroSecAgg$_{gDL}$	5 | 3	$O(n\log^2 n)$	$O(nL + 2^{\ell/2}L)$	$O(\log^2 n)$	$O(L\log n)$	$O(n\kappa\log n)$	$O(n\kappa(L+\log n))$	$O(\kappa\log n)$	$O(\kappa(L+\log n))$
MicroSecAgg$_{CL}$	5 | 3	$O(n\log^2 n)$	$O(nL)$	$O(\log^2 n)$	$O(L\log n)$	$O(n\kappa\log n)$	$O(n\kappa(L+\log n))$	$O(\kappa\log n)$	$O(\kappa(L+\log n))$

Table 2: Total asymptotic computation cost for all rounds per aggregation with malicious security. n denotes the total number of users, L denotes the length of the input vector, and ℓ denotes the bit length of each element in the input vector. The round column indicates the number of rounds in the one-time setup phase (on the left, if applicable) and in each aggregation phase (on the right). We assume the number of offline users is $O(n)$ in every aggregation iteration in this table. See Table 3 for the case where a δ fraction of users is offline each iteration. "forwarding" means that the server only forwards the messages from the users.

$x_1, \ldots, x_n$ while hiding each individual input x_i, each 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 canceled 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 $\mathsf{mk}_{i,j} = \mathsf{mk}_{j,i}$, then they mask their inputs by user i adding $\mathsf{mk}_{i,j}$ to x_i while user j subtracting $\mathsf{mk}_{j,i}$ from x_j. In other words, each user i computes a mask $h_i = \sum_{j<i} \mathsf{mk}_{i,j} - \sum_{j>i} \mathsf{mk}_{i,j}$ 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 $\mathsf{mk}_{i,j}$ and $-\mathsf{mk}_{i,j}$ 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. More specifically, each user i shares its masks $\mathsf{mk}_{i,j}$ 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 those users' 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 $\mathsf{ek}_{i,j}$ (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 $\mathsf{mk}_{i,j}$ 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 O, the server needs to remove $\sum_{i\in O} r_i$ of the online users from $\sum_{i\in O} X_i$ and cancel $\sum_{i\in O} h_i$ with $\sum_{i\notin O} h_i$. 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 $\mathsf{mk}_{i,j}$. 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 sets of users to help it reconstruct the masks of different subsets 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 securely covered.

2.2 Our first solution: MicroSecAgg$_{DL}$

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 need 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 O denote the set of online users i who successfully send the server X_i. Then the server requires $\sum_{i\in O} r_{i,j}$ from online users j. As long as at least t users j reply, the server can reconstruct $\sum_{i\in O} r_i$ 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 sums 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. Instead of sending $X_i = r_i + x_i$ and the sum of shares $\sum_{j\in O} r_{j,i}$, now each user sends $H(k)^{X_i}$ and $H(k)^{\sum_{j\in O} r_{j,i}}$ to the server. The server can then reconstruct $H(k)^{\sum_{i\in O} r_i}$ 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 $H(k)^{\sum_{i\in O} r_i}$ which is uniformly random. At the same time, as $\sum_{i\in O} x_i$ is much smaller, the server can obtain it by calculating the discrete log of $H(k)^{\sum_{i\in O} X_i}/H(k)^{\sum_{i\in O} r_i}$.

2.3 Reducing Communication Cost: MicroSecAgg$_{gDL}$

In this section, we introduce the second construction, MicroSecAgg$_{gDL}$, which improves the communication efficiency of MicroSecAgg$_{DL}$.

In MicroSecAgg$_{DL}$, in every iteration, each user still needs to receive the online set from the server which is of size $O(n)$. 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 MicroSecAgg$_{DL}$ 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. The protocol description of the Aggregation phase is included in Section 4. Due to the page limit, we delay the security proofs to Appendix C. With analysis shown in Appendix C, we can choose groups of size $O(\log n)$, thus reduce the asymptotic communication cost per user from $O(n)$ to $O(\log n)$.

2.4 Handling larger Input Domain: MicroSecAgg$_{CL}$

The solutions introduced above require calculating discrete logarithm in a large group. As there is no known efficient algorithm for a large range of discrete logarithms, it limits the domain of user inputs. To solve problems with a 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 compute. This primitive is formalized in [20] and 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 inputs is easy while the random mask is still in the exponent of g to cover the secret input. The same technique can be applied to both our protocols MicroSecAgg$_{DL}$ and MicroSecAgg$_{gDL}$. In Section 5 we give a modified version of MicroSecAgg$_{gDL}$ using class groups instead of general cyclic groups.

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 [1, 12, 14, 17, 22, 26, 30, 49]. For a more comprehensive understanding of the class group assumptions and more constructions, we refer the readers to Section 3, Appendix A.3, and the BICYCL work [13].

Remark (About Sybil attack). *Sybil attacks in federated learning involve malicious entities creating multiple fake or forged identities to manipulate the collaborative learning process. In this context, these attackers generate numerous false client nodes or identities to participate in the federated learning network, aiming to influence the model's training process or learn information on the summation of the honest parties sum. Sybil-attacks are a common issue in single-server protocols using threshold-secret-sharing if the threshold of corrupted parties is exceeded by the attacker. If the adversary obtains enough number of secret shares in the threshold-secret-sharing schemes, it can recover the secret and thus breach the privacy. Sybil-attacks affect all secure aggregation protocols and it has been an ongoing challenge that we do not address in this work if the server introduces more than our corruption threshold fake devices. Only the recent work of David Byrd et al. [16] which is based on oblivious distributed differential privacy to counter client collusion privacy provides an aggregation solution to limit such attacks.*

3 PRELIMINARIES

3.1 Notations and Cryptographic Primitives

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

Let p, q be two primes such that $p = 2q+1$. A finite field $\mathbb{Z}_p$ is a set of elements $\{0, 1, \ldots, p-1\}$ 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.

Negligibility and Indistinguishability. A function $f : \mathbb{N} \to \mathbb{R}$ 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 indistingsuishable (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 $\{0, 1, \ldots, p-1\}$ and $\mathbb{Z}_p^*$ denotes a group $\{1, \ldots, p-1\}$. $\mathbb{G}$ 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 $g \in \mathbb{G}$, $\mathbb{G} = \{g^0, g^1, \ldots, g^{q-1}\}$. 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 comparing the power and discrete log computation happening in group $\mathbb{G}$.

Shamir's Secret Sharing. We use Shamir's t-out-of-n secret sharing in [43] 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 $s, X_1, \ldots, X_n \in \mathbb{Z}_q$ for some prime q. The Shamir's Secret Sharing scheme consists of two algorithms:

- SS.share$(s, \{X_1, X_2, \ldots, X_n\}, t) \to \{(s_1, X_1), \ldots, (s_n, X_n)\}$, in which s denotes the secret, $X_1, \ldots, X_n$ 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.
- SS.recon$(\{(s_1, X_1), \ldots, (s_n, X_n)\}, t) = s$, 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, SS.expoRecon$((g^{s_1}, X_1), \ldots, (g^{s_n}, X_n), t) = g^s$, which allows the reconstruction of the secret s with shares $s_1, \ldots, s_n$ being performed in the exponents of finite field elements. 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 three algorithms: AE.gen$(1^\kappa) \to k$, which randomly generates a key k; AE.enc$(m, k) \to c$, which encrypts message m with a key k and generates a ciphertext c; and AE.dec$(c, k) \to m$, 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.

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 $\mathbb{Z}_q$:

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

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 DDH problem is computationally hard. It consists of three algorithms,

- KA.setup$(\kappa) \to (\mathbb{G}', g, q, H)$, in which $\mathbb{G}'$ is a group of order q with a generator g, H is a cryptographically secure hash function;
- KA.gen$(\mathbb{G}', g, q, H) \to (x, g^x)$ 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.
- KA.agree$(x_u, g^{x_v}) \to s_{u,v} = H((g^{x_v})^{x_u})$. 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.

DDH Group with an Easy DL Subgroup. To accommodate larger input, we use an assumption from [20]. 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 [20]:

Definition 3.1 (DDH Group with an Easy DL Subgroup [20]). *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 λ and μ 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 λ-bit integer, p is a μ-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 $\{g^r, r \overset{\$}{\leftarrow} \{0, \ldots, Bp-1\}\}$ is statistically indistinguishable from the uniform distribution on G. We assume that the canonical surjection $\pi : G \to G/F$ 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 cl.solve *algorithm is a deterministic polynomial time algorithm that solves the discrete logarithm problem in F:*

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

(2) The DDH problem is hard in G even with access to the cl.solve *algorithm:*

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

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

3.2 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 property of the protocol below and the privacy property in Appendix A.4.

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

4 MicroSecAgg$_{gDL}$ PROTOCOL

In this section, we introduce the construction of the secure aggregation protocol MicroSecAgg$_{gDL}$. Due to the length limit, we include the security proofs in Appendix C.

For simplicity, we assume that 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 on the assignment together. We choose the group size in the same way and under the same assumptions as how the neighborhood size is chosen in BBG+20 [10].

4.1 Protocol

In this section, we describe the Setup phase and the Aggregation phase of MicroSecAgg$_{gDL}$ in Algorithm 1 and Algorithm 2 respectively. The parts that are only needed to protect privacy against a malicious adversary are marked with blue color and underlines. In the protocol description, we call one round of back-and-forth messaging between the users and the server a *round*. In other words, in one round, the users first sends the server messages and then receive messages from the server.

We also depict the high-level idea of the Setup phase and the Aggregation phase in Figure 7 and Figure 8 in Appendix B.

Algorithm 1 Setup phase (MicroSecAgg$_{gDL}$)

This protocol uses the following algorithms defined in Section 3: 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 $\mathcal{S}$ 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 public parameters: the security parameter κ, the number of users n, a threshold value t, honestly generated $pp \leftarrow \mathsf{KA.setup}(\kappa)$ for key agreement, the input space $\mathcal{X}$, a field $\mathbb{F}$ for secret sharing.

Moreover, all clients are uniformly randomly divided into B groups $G_1, \ldots, G_B$, 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 $\mathcal{S}$ 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 $\mathcal{U}_\mathcal{S}$ such that $|\mathcal{U}_\mathcal{S}| \ge Bt$ and a global mask $h_\mathcal{S}$ or abort.

Round 1: Key Exchange:

1: Each user $i \in G_d$: It generates a pair of encyption keys $(\mathsf{pk}_i, \mathsf{sk}_i)$ and two pairs of masking keys $(\mathsf{pk}_i^1, \mathsf{sk}_i^1)$ and $(\mathsf{pk}_i^{-1}, \mathsf{sk}_i^{-1})$. It sends the three public keys with signatures on them to the server.

2: Server $\mathcal{S}$: Let $\mathcal{U}_\mathcal{S}^1$ denotes the set of users send the server public keys with valid signatures. For $i \in \mathcal{U}_\mathcal{S}^1 \cap G_d$, the server distributes the public encryption key pk_i and the signature received from user $i \in G_d$ to all users in $\mathcal{U}_\mathcal{S}^1 \cap (G_d \cup G_{d-1} \cup G_{d+1})$, and distributes the public masking key pk_i^b with the signatures to all users in $\mathcal{U}_\mathcal{S}^1 \cap G_{d+b}$ for $b \in \{-1, 1\}$.

Round 2: Secret Mask Key Sharing:

3: Each user $i \in G_d$:

On receiving $(\mathsf{pk}_j, \mathsf{pk}_j^b \sigma_j)$ from the server where $b = 1$ or -1, each user i verifies the signature σ_j with pk_j. It aborts if any signature verification fails, or if it receives messages from less than t users from any one of groups G_d, G_{d-1} and G_{d+1} after processing all messages. Otherwise, it puts j into a user list $\mathcal{U}_i^1$ and stores $\mathsf{ek}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j, \mathsf{sk}_i)$ and $\mathsf{mk}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j^b, \mathsf{sk}_i^{-b})$.

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

4: Server $\mathcal{S}$: Let $\mathcal{U}_\mathcal{S}^2$ denote the set of users who successfully send the server messages. If for any group G_d, $|\mathcal{U}_\mathcal{S}^2 \cap G_d| < t$, abort. Otherwise, For each group $d \in [B]$ and each $i \in \mathcal{U}_\mathcal{S}^2 \cap G_d$, The server sends each encrypted share $c_{i,j}^{\mathsf{sk}^b}$ 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 $c_{j,i}^{\mathsf{sk}^b}$ for $b = \{1, -1\}$ with $\mathcal{U}_i^2$. It decrypts the encrypted share by $\mathsf{sk}_{j,i}^b = \mathsf{AE.dec}(c_{j,i}^{\mathsf{sk}^b}, \mathsf{ek}_{i,j})$. If any $c_{j,i}^{\mathsf{sk}^b}$ 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 $h_i = \sum_{j \in \mathcal{U}_i^2 \cap G_{d-1}} \mathsf{mk}_{i,j} - \sum_{j \in \mathcal{U}_i^2 \cap G_{d+1}} \mathsf{mk}_{i,j}$. 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 $\mathsf{ek}_{i,j}$ to generate ciphertext $c_{i,j}^r, c_{i,j}^h$ for each share for user j. 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 $\mathcal{U}_\mathcal{S}^3$. If for any group G_d, $|\mathcal{U}_\mathcal{S}^3 \cap G_d| < t$, abort. Otherwise, It sends the shares to the corresponding receiver j for each $i \in \mathcal{U}_\mathcal{S}^3$, and an offline set of group d $\mathsf{offline}_d = G_d \cap (\mathcal{U}_\mathcal{S}^2 \backslash \mathcal{U}_\mathcal{S}^3)$ to users in $\mathcal{U}_\mathcal{S}^3 \cap (\mathcal{U}_{d-1} \cup \mathcal{U}_{d+1})$. (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 $r_{j,i} = \mathsf{AE.dec}(c_{j,i}^r, \mathsf{ek}_{i,j})$, and $h_{j,i} = \mathsf{AE.dec}(c_{j,i}^h, \mathsf{ek}_{i,j})$. 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 $\mathsf{offline}_i = (\mathcal{U}_i^1 \cap G_d) \backslash \mathcal{U}_i^3$ with the signature σ_i on the list to the server.

8: Server $\mathcal{S}$: If the server receives $\mathsf{offline}_i$ with valid signatures σ_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 $\mathcal{U}_\mathcal{S}^4$. The server sends the list and the signature $(\mathsf{offline}_i, \sigma_i)$ for all $i \in G_d \cap \mathcal{U}_\mathcal{S}^4$ to all users in $(G_{d-1} \cup G_{d+1}) \cap \mathcal{U}_\mathcal{S}^4$.

Round 5: Reconstructing Offline Users' Masks:

9: Each user $i \in G_d$: After receiving all user lists with the signature $(\mathsf{offline}_j, \sigma_j)$ 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 $\mathsf{offline}_{d-1}$. It repeats the process on group G_{d+1}. it sends $\mathsf{sk}_{j',i}^1$ for $j' \in \mathsf{offline}_{d-1}$ and $\mathsf{sk}_{j',i}^{-1}$ for $j' \in \mathsf{offline}_{d+1}$ 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 $\mathsf{sk}_{i,j}^b$ from both groups G_{d+b} for $b \in \{-1, 1\}$, the server puts i into a user list $\mathsf{offline}_\mathcal{S}$. 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 $\mathsf{mk}_{i,j}$ between itself and the member j of i's neighbor group have been included in the h_j. Thus, the server needs to calculates h_i by reconstructing sk_i^b for $b = \pm 1$, running the key exchange algorithm for i and user $j \in \mathcal{U}_\mathcal{S}^2 \cap G_{d-1}$ by $\mathsf{mk}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j^1, \mathsf{sk}_i^{-1})$ and for i and user $j \in \mathcal{U}_\mathcal{S}^2 \cap G_{d+1}$ by $\mathsf{mk}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j^{-1}, \mathsf{sk}_i^1)$, and calculating $h_i = \sum_{j \in \mathcal{U}_\mathcal{S}^2 \cap G_{d-1}} \mathsf{mk}_{i,j} - \sum_{j \in \mathcal{U}_\mathcal{S}^2 \cap G_{d+1}} \mathsf{mk}_{i,j}$. Then it obtains $\mathcal{U}_\mathcal{S} = \mathcal{U}_\mathcal{S}^2 \backslash \mathsf{offline}_\mathcal{S}$ and $h_\mathcal{S} = \sum_{i \in \mathsf{offline}_\mathcal{S}} h_i$.

Algorithm 2 Aggregation phase ($\mathsf{MicroSecAgg}_{gDL}$)

This protocol uses the following algorithms defined in Section 3: public key infrastructure, a Shamir's secret sharing scheme ($\mathsf{SS.share}, \mathsf{SS.recon}, \mathsf{SS.expoShare}, \mathsf{SS.expoRecon}$); 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 $\mathcal{U}_\mathcal{S}$ 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 $\sum_{i \in O_k} x_{i,k}$, in which 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 $k = 1, 2, \ldots$ **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 $\mathcal{U}_\mathcal{S}$, ignore the message. Otherwise, let O denote the set of users i who successfully send the masked input to the server. For each group G_d, the server sends $O_d = O \cap G_d$ to all users $i \in O_d$.

Round 2: Online Set Checking:

```
    User $i$: On receiving $O_d$ from the server, it checks that $|O_d| \geq t$ and signs it with its signing key. Then it sends the signature $\sigma_i$ to the server.
    Server $\mathcal{S}$: On receiving the signatures from user $i \in G_d$, It verifies the signature on $O_d$ and user $i$'s verification key. If the signature is invalid, ignore it. If it receives less than $t$ valid signatures from any group, abort. Otherwise, it forwards all valid signatures on $O_d$ to user $i \in O_d$ for each group $G_d$.
    Round 3: Mask Reconstruction:
    User $i$: it checks that it receives at least $t$ valid signatures from the members of $G_d$ on $O_d$. If any signature is invalid, abort. Then it calculates $\zeta_i = H(k)^{\sum_{j \in O_d} r_{j,i} - \sum_{j \in \mathcal{U}_i \setminus O_d} h_{j,i}}$ and sends $\zeta_i$ to the server.
    Server $\mathcal{S}$: If it receives $\zeta_i$ from less than $t$ users in any group $G_d$, abort. Otherwise, the server calculates
        $z = \log(H(k)^{\sum_{i \in O} X_i} / \prod_{d \in [B]} \mathsf{SS.expoRecon}(\{\zeta_j, j\}_{j \in O'_d}, t))$
    by brute force. Here $O'_d$ denotes the users from $G_d$ who send $\zeta$ to the server in the previous round. Then it calculates $\sum_{i \in O} x_i = z + h_{\mathcal{S}}$, in which $h_{\mathcal{S}}$ is from the server's output in the Setup phase.
end for
```

For simplicity of exposition, in the description of the protocol, we assume that the input vector is of length 1. To handle input vectors of length L, we only need to use a vector of independently randomly chosen generator in every iteration as the base of exponentiation, for example, by substituting $H(k)$ with a vector $H((k-1)L+1), H((k-1)L+2), \ldots, H((kL)$, and for each index $l \in [L]$, masking the input with $H((k-1)L+l)^{X_{i,l}}$ (in which $X_{i,l} = x_{i,l} + r_i + h_i$) in Round 1 of Algorithm 2 of MicroSecAgg$_{gDL}$.

Remark (About group sizes and dynamic participation). *In this work, we make the assumption that all groups are of the same size and all the participants are predetermined for simplicity in analyzing group properties. Our protocol can work with various group sizes as long as each group contains a proportion of honest and online nodes greater than the secret-sharing threshold. To allow dynamic joining, we need a more complex model describing the behavior of the users, e.g., the distribution of corrupt users in the new joiners, the communication model of new users, etc., which we leave out of the scope of this paper.*

4.2 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 [10]. First, we define the requirements as the following two good events:

Definition 4.1 (Not too many corrupt members). *Let N, t be integers such that $N < n$ and $t \in (N/2, N)$, and let $C \subset [n]$. Let $\mathbf{G} = (G_1, \ldots, G_B)$ is a partition of $[n]$ so that $|G_d| = N$ for each $d \in [B]$. We define event E_1 as*

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

Definition 4.2 (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 $\mathbf{G} = (G_1, \ldots, G_B)$ is a partition of $[n]$ so that $|G_d| = N$ for each $d \in [B]$. We define event E_2 as*

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

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

(1) $\Pr[E_1(C, \mathbf{G}', N, t') = 1 \mid (\mathbf{G}', t') \leftarrow \mathcal{D}] > 1 - 2^{-\sigma}$,
(2) $\Pr[E_2(D, \mathbf{G}', N, t') = 1 \mid (\mathbf{G}', t') \leftarrow \mathcal{D}] > 1 - 2^{-\eta}$.

Lemma 4.4. *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 $C \subset [n]$, such that $C \leq \gamma n$, 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 (σ, η, C)-nice.

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

4.3 Correctness with Dropouts

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

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 (σ, η, C)-nice, by definition, event $E2$ 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

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

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 $h_{\mathcal{S}}$ to z, the server gets the sum $\sum_{i\in\mathcal{O}} x_i$. □

4.4 Privacy against Semi-Honest and Malicious Adversaries

It is easy to see that 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 4.6. We defer the proof to Appendix C due to the page limit.

Theorem 4.6 (Privacy against Malicious Adversary). *Let γ and δ be two parameters such that $\gamma < 1/3$, $\gamma + 2\delta < 1$, and σ and η be two security parameters. The protocol Π being an instance of Algorithm 1 and Algorithm 2 guarantees privacy against γ-fraction of malicious adversary with δ offline rate with probability $1 - 2^{-\sigma}$ when the grouping is (σ, η, 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.*

Remark (About Output Delivery with Malicious Adversary). *In the malicious setting, if the attack the adversary performs is only controlling the corrupt users to drop out during the execution of the protocol and the total number of users who stay online and follow the protocol is more than $(1 - \delta)n$ where n is the total number of users, MicroSecAgg$_{gDL}$ can guarantee that the server obtains the aggregation result. In other words, the corrupt users maliciously keeping silent does not halt the protocol, as long as there are enough number of users online and following the protocol. The intuition is that every round only requires the number of online participants to be larger than the threshold to continue when every online user is following the protocol. The protocol does not provide full robustness which means the server can both identify misbehaving clients and remove them. If the malicious users perform other attacks, only the privacy of honest users is guaranteed and the server might not obtain the result. Note that none of previous works other than ACORN [9] achieves robustness, and ACORN gives full robustness, in a theoretical construction without implementation, but at the expense of $O(\log n)$ round complexity which is prohibitive.*

4.5 MicroSecAgg$_{DL}$: Special Case with One Single Group

The first solution, MicroSecAgg$_{DL}$, introduced in Section 2 can be viewed as a special case of MicroSecAgg$_{gDL}$ where there is only one group which contains all the users. As there is only one group, the h-mask which is intended to hide the sums of individual groups is not needed. Therefore, for the Setup phase, the Protocol 1 can be simplified as described below:

- In Round 1 (line 1 and 2), each user i only needs to generate one pair of encryption keys $\mathsf{pk}_i, \mathsf{sk}_i$ and send pk_i to the server who then forwards it to all users.
- Round 2 and Round 3 can be combined and simplified as following. In line 3, each user i runs one instance of key agreement with every other user j to agree on the symmetric encryption key $\mathsf{ek}_{i,j}$. As now the h-mask is not needed, the second part of line 3 where user i secret shares the secret keys for h-mask generation is no needed. Thus, user i can directly continue to line 5 and it only needs to randomly choose and secret share an r-mask. User i then sends the ciphertext of the shares $c^r_{i,j}$ encrypted with $\mathsf{ek}_{i,j}$ to the server who forwards them to the corresponding receivers as described in line 5 and 6.
- Round 4 and 5 can be skipped, as they are used to help the server reconstruct the h-masks for users who drop offline in the Setup phase. Now that the h-mask is removed, these two rounds can also be reduced.

Moreover, for the Aggregation phase, the Protocol 2 can be simplified as described below:

- In line 2, user i computes $X_i = x_i + r_i$.
- In line 6, user i computes $\zeta_i = H(k)^{\sum_{j\in\mathcal{O}} r_{j,i}}$, where $\mathcal{O}$ denotes the online set of all users.
- In line 7, no $h_{\mathcal{S}}$ is needed.

The simplified protocol MicroSecAgg$_{DL}$ has a Setup phase requiring only two rounds and achieves stronger security guarantee.

5 MicroSecAgg$_{CL}$: HANDLING LARGE INPUT DOMAINS WITH CLASS GROUP

To overcome the limitation that the solution using discrete logarithm in general cyclic groups can only handle small-scale problems, we propose an alternative version of our protocol based on the class group primitive as defined in Definition 3.1. 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 $\mathbb{F}$ to hide secret inputs so that large inputs can be easily recovered 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.

To guarantee the security of the protocol, the server should only learn the order of the easy subgroup F to perform discrete logarithm in it while not knowing the order of the hard large group G. Section 2.6.2 of [45] further states that the running time to find the order of group G is computationally infeasible (at least as hard as factoring). Thus, even if the server colludes with users, the security of the protocol won't be compromised as the order of group G is unknown to all parties. 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.

We describe the protocol of MicroSecAgg$_{CL}$. The Setup phase is the same as the Setup phase of MicroSecAgg$_{gDL}$ in Algorithm 1. We describe the Aggregation phase in Algorithm 3. We mark the part of execution that only needed in malicious settings with blue color and underlines.

Same as in Algorithm 2, we assume at the end of Setup phase, each user i in group G_d holds a random mask r_i, a mutual mask h_i, and a share $r_{j,i}$ and $h_{j,i}$ of masks for each of its neighbor $j \in G_d$ and the server holds a value $h_{\mathcal{S}}$. We assume that $\sum_{i\in\mathcal{U}} h_i + h_{\mathcal{S}} = 0$.

Algorithm 3 Aggregation (MicroSecAgg$_{CL}$)

This protocol uses the following algorithms defined in Section 3: public key infrastructure, a Shamir's secret sharing scheme (SS.share, SS.recon, SS.expoShare, SS.expoRecon). 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, 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, 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 3.1.

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 and inputs it receives in the Setup phase, and the list of users $\mathcal{U}_{\mathcal{S}}$ it outputs in 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 $\sum_{i\in O_k} x_{i,k}$, in which 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.

```
for Iteration $k = 1, 2, \ldots$ do
    Round 1: Secret Sharing:
    User $i$: It calculates and sends $X_i = g_k^{r_i+h_i} f^{x_i}$ to the server.
    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 $\mathcal{U}_{\mathcal{S}}$, ignore the message. Otherwise, let $O$ denote the set of users $i$ who successfully send the masked input to the server. For each group $G_d$, the server sends $O_d = O \cap G_d$ to all users $i \in O_d$.
    Round 2: Online Set Checking (Only needed in Malicious setting):
    User $i$: On receiving $O_d$ from the server, it checks that $|O_d| \geq t$ and signs it with its signing key. Then it sends the signature $\sigma_i$ to the server.
    Server $\mathcal{S}$: On receiving the signatures from user $i \in G_d$, It verifies the signature on $O_d$ and user $i$'s verification key. If the signature is invalid, ignore it. If it receives less than $t$ valid signatures from any group, abort. Otherwise, it forwards all valid signatures on $O_d$ to user $i \in O_d$ for each group $G_d$.
    Round 3: Mask Reconstruction on the Exponent:
    User $i$: On receiving signatures from the server, it first verifies the signatures of the other users. If there are less than $t$ valid signatures, abort. Otherwise, it calculates $\zeta_i = g_k^{\sum_{j\in O_d} r_{j,i} - \sum_{j\in\mathcal{U}_i\setminus O_d} h_{j,i}}$. It sends $\zeta_i$ to the server.
    Server $\mathcal{S}$: If it receives $\zeta_i$ from less than $t$ users in any group $G_d$, abort. Otherwise, the server calculates
        $$R_{O_d} = \text{SS.expoRecon}(\{\zeta_j, j\}_{j\in O'_d}, t)$$
    in which $O'_d$ denotes the users from $G_d$ who send $\zeta$ to the server in the previous round. It then calculates
        $$\sum_{i\in O} x_i \leftarrow \text{cl.solve}\left(B, p, g, f, G, F, \prod_{i\in O} X_i / \prod_d R_{O_d}\right)$$
    to obtain $\sum_{i\in O} x_i$.
end for
```

6 PERFORMANCE

To measure the concrete performance, we provide end-to-end implementations of our protocols [7], as well as three benchmark protocols, BIK+17, BBG+20 and Flamingo with ABIDES [15], a discrete event simulation framework with modifications to enable simulation of multi-iterative aggregation protocols, in Python language. In all implementations, we assume semi-honest settings, thus we omit the marked parts of the protocols that are 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.

Comparison with BIK+17 and BBG+20. In the first two graphs in Figure 1, we compare the local computation time of four protocols (taking the average time of 10 aggregations) with the length of the result fixed to 20 bits and group/neighbor size fixed to 100 for MicroSecAgg$_{gDL}$ and BBG+20. As shown in the graph, the computation time of MicroSecAgg$_{DL}$ is about 100 times shorter than BBG+20 when the total number of users is 500, and the computation time of MicroSecAgg$_{gDL}$ is about 20 times faster than BBG+20 when the total number 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 MicroSecAgg$_{DL}$ and MicroSecAgg$_{gDL}$ 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 MicroSecAgg$_{DL}$ is also almost the same as MicroSecAgg$_{gDL}$, 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 in Figures 1 are run with brute force discrete logarithm. With class groups in MicroSecAgg$_{CL}$, 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

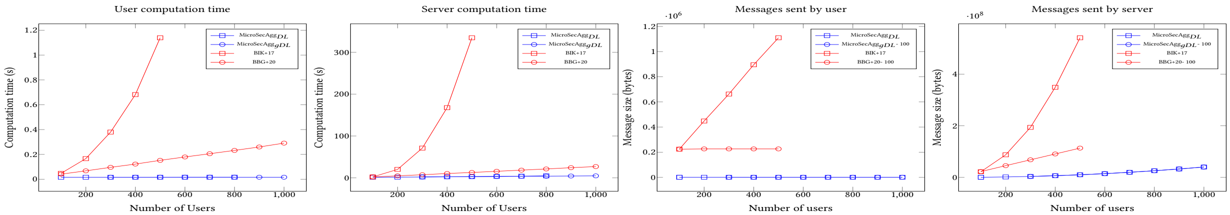

Figure 1: Wall-clock local computation time and outbound message sizes of one iteration of the aggregation phase as the number of users increases. The length of the sum of inputs is fixed to $\ell = 20$ bits, i.e., the input of each user is in the range $[2^\ell/n]$ when the total number of users is n. For the protocol MicroSecAgg$_{gDL}$ and BBG+20, the group/neighbor size is set to 100.

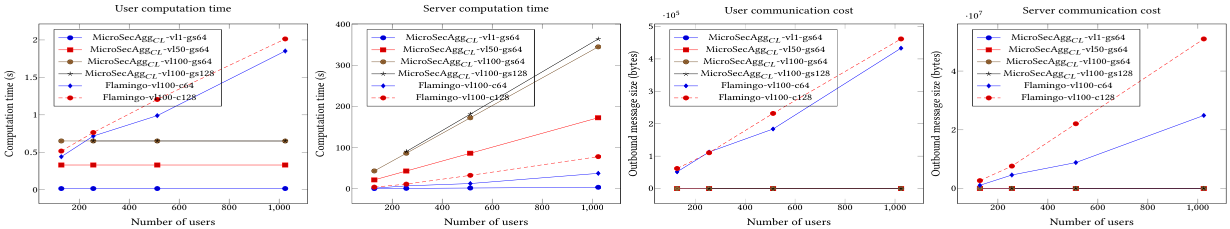

Figure 2: The left two graphs show the wall-clock local computation time of one iteration of the aggregation phase as the number of users increases. The right two graphs are the outbound bandwidth cost (bytes) on the user and the server side when the total number of users grows. In the legend, "vl" denotes the length of input vector, "gs" denotes group size for MicroSecAgg$_{CL}$, "c" denotes the number of decryptors for Flamingo.

compute on a 32-bit range with class group library [13] instantiated with 256-bit primes in the same environment.

Comparison with Flamingo. We also give a performance comparison between our protocol instantiated *with the class group* MicroSecAgg$_{CL}$ and Flamingo [37] in Figure 2. 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. 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.

Aggregation phase performance. Figure 3 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 MicroSecAgg$_{DL}$ is not affected significantly by the number of users but is affected more 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 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 is the same in these scenarios. 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. In the Aggregation phase of MicroSecAgg$_{DL}$, 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 when the input is within the group order of the easy subgroup, which allows input size of hundreds of bits.

Full execution performance. In Figure 4, we report the computation and communication costs of both the setup phase (if applicable, e.g., in MicroSecAgg and Flamingo) and 10 iterations of the aggregation phase to give a whole picture of the performance of different protocols. All the protocols are executed with one single input, 20-bit aggregation result, and group/neighborhood/committee of size 100 when applicable. We assume all users are online during the whole process. As shown in the figure, all three of our protocols outperform BIK+17, BBG+20, and Flamingo regarding computation and communication cost under this setting. More specifically,

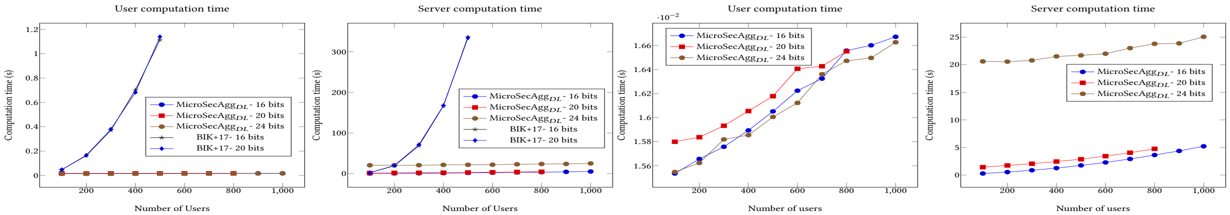

Figure 3: The left two graphs are the wall-clock local computation time of one iteration of the aggregation phase as the number of user increases. The length of the sum of inputs is set to $\ell = 16, 20, 24$ bits in different lines, i.e., the input of each user is in the range $[2^{\ell}/n]$ when the total number of users is n. The right two graphs are zoom-in which only includes MicroSecAgg$_{DL}$ protocol.

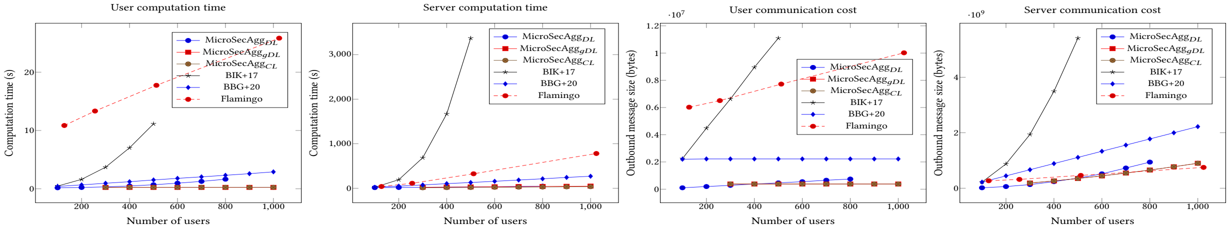

Figure 4: The total local computation time and the total communication cost of different protocols, including the Setup phase (if applicable) and 10 iterations of the Aggregation phase. Bit-length of the aggregation result: 20; group/neighborhood size (if applicable): 100; vector length: 1.

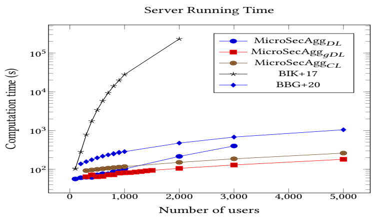

Figure 5: The server running time from the start to the end of different protocols in the simulated WAN environment, including the Setup phase (if applicable) and 10 iterations of the Aggregation phase. Bit-length of the aggregation result: 20; group/neighborhood size (if applicable): 100; vector length: 1.

for 500 users, MicroSecAgg$_{CL}$ is 6 times faster than BBG+20 and 42 times faster than Flamingo (decryptor) on the user side and 7 times faster than BBG+20 and 10 times faster than Flamingo on the server side; it also causes communication overhead 5 times less than BBG+20 and 11 times less than Flamingo on the user (decryptor for Flamingo) side, and 3 times less than BBG+20 and 19 times less than Flamingo on the server side.

One thing to note is that in practice, the device on the user side is usually with more limited resource while the server usually runs on more powerful machines than the AWS machine we use in the experiments.

In WAN environment. In Figure 5, we provide a comparison of the server running time from the start of the protocol to the end of 10 aggregation iterations. The Setup phase is included if applicable. We simulate the WAN environment assuming users are distributed in major cities in different continents over the world. We assume the server resides in New York City and the longest latency from the users to the server is from Sydney, Australia. The ping latency between the two cities is on average 200ms. We assume the bandwidth to be 100 Mbps. We only report the server running time as the users are responsive and reply only when they receive messages from the server. In particular, in each round, the server needs to wait for a fixed amount of time to collect the messages from all users, we set the waiting interval of the server based on the estimated calculation time of the users and the estimated longest communication time between the users and the server.

Other results. In Appendix E.4, we also include results comparing the performance of different group sizes between MicroSecAgg$_{gDL}$ and BBG+20 (Figure 9), the performance of MicroSecAgg$_{DL}$ and MicroSecAgg$_{gDL}$ with different offline rates (Figure 10), and the performance of the Setup phase of MicroSecAgg$_{gDL}$ (Figure 11). We also run a federated learning protocol with MicroSecAgg$_{DL}$ on the adult census income dataset [27] to show that our secure aggregation solution does not affect accuracy (see Appendix E.4).

7 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.

8 ACKNOWLEDGEMENT

This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("J.P. Morgan") and is not a product of the Research Department of J.P. Morgan. J.P. Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

REFERENCES

[1] 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.

[2] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. 2022. Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares. In *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13409)*, Clemente Galdi and Stanislaw Jarecki (Eds.). Springer, 516–539. https://doi.org/10.1007/978-3-031-14791-3_23

[3] 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.

[4] 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).

[5] Mohammad Mohammadi Amiri and Deniz Gunduz. 2020. Federated Learning over Wireless Fading Channels. *arXiv:1907.09769 [cs, math]* (Feb. 2020). http://arxiv.org/abs/1907.09769 arXiv: 1907.09769.

[6] Mohammad Mohammadi Amiri and Deniz Gunduz. 2020. 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.

[7] Anonymous. 2023. Secure Aggregation Simulation. https://github.com/AnonymousOctupus/MicroFedML.

[8] Apple and Google. 2021. Exposure Notification Privacy-preserving Analytics (ENPA). https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf

[9] 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).

[10] 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.

[11] 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

[12] 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.

[13] Cyril Bouvier, Guilhem Castagnos, Laurent Imbert, and Fabien Laguillaumie. 2022. I want to ride my BICYCL: BICYCL Implements CryptographY in CLass groups. (2022).

[14] Lennart Braun, Ivan Damgård, and Claudio Orlandi. 2023. Secure multiparty computation from threshold encryption based on class groups. In *Annual International Cryptology Conference*. Springer, 613–645.

[15] 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

[16] David Byrd, Vaikkunth Mugunthan, Antigoni Polychroniadou, and Tucker Balch. 2022. Collusion Resistant Federated Learning with Oblivious Distributed Differential Privacy. In *3rd ACM International Conference on AI in Finance, ICAIF 2022, New York, NY, USA, November 2-4, 2022*, Daniele Magazzeni, Senthil Kumar, Rahul Savani, Renyuan Xu, Carmine Ventre, Blanka Horvath, Ruimeng Hu, Tucker Balch, and Francesca Toni (Eds.). ACM, 114–122. https://doi.org/10.1145/3533271.3561754

[17] 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.

[18] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. 2020. Bandwidth-Efficient Threshold EC-DSA. 266–296. https://doi.org/10.1007/978-3-030-45388-6_10

[19] Guilhem Castagnos, Laurent Imbert, and Fabien Laguillaumie. 2017. Encryption Switching Protocols Revisited: Switching Modulo p. 255–287. https://doi.org/10.1007/978-3-319-63688-7_9

[20] 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.

[21] Guilhem Castagnos and Fabien Laguillaumie. 2015. Linearly Homomorphic Encryption from DDH. 487–505. https://doi.org/10.1007/978-3-319-16715-2_26

[22] 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.

[23] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2018. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[24] Henry Corrigan-Gibbs. 2020. Privacy-preserving Firefox telemetry with Prio. https://rwc.iacr.org/2020/slides/Gibbs.pdf

[25] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 259–282. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs

[26] Yi Deng, Shunli Ma, Xinxuan Zhang, Hailong Wang, Xuyang Song, and Xiang Xie. 2021. Promise Σ-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.

[27] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[28] 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.

[29] 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).

[30] 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).

[31] Anders Krogh and John Hertz. 1991. A simple weight decay can improve generalization. *Advances in neural information processing systems* 4 (1991).

[32] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. 2023. LERNA: Secure Single-Server Aggregation via Key-Homomorphic Masking. In *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14438)*, Jian Guo and Ron Steinfeld (Eds.). Springer, 302–334. https://doi.org/10.1007/978-981-99-8721-4_10

[33] 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.

[34] 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.

[35] Zizhen Liu, Si Chen, Jing Ye, Junfeng Fan, Huawei Li, and Xiaowei Li. 2022. SASH: Efficient secure aggregation based on SHPRG for federated learning. In *Uncertainty in Artificial Intelligence*. PMLR, 1243–1252.

[36] Ziyao Liu, Jiale Guo, Wenzhuo Yang, Jiani Fan, Kwok-Yan Lam, and Jun Zhao. 2022. Privacy-preserving aggregation in federated learning: A survey. *IEEE Transactions on Big Data* (2022).

[37] 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).

[38] 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.

[39] 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.

[40] 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.

[41] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. 2023. ELSA: Secure Aggregation for Federated Learning with Malicious Actors. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1961–1979. https://doi.org/10.1109/SP46215.2023.10179468

[42] 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.

[43] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176

[44] 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.

[45] Ida Tucker. 2020. *Functional encryption and distributed signatures based on projective hash functions, the benefit of class groups.* Theses. Université de Lyon. https://theses.hal.science/tel-03021689

[46] Twan Van Laarhoven. 2017. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350* (2017).

[47] 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.

[48] Chien-Sheng Yang, Jinhyun So, Chaoyang He, Songze Li, Qian Yu, and Salman Avestimehr. 2021. LightSecAgg: Rethinking Secure Aggregation in Federated Learning. *arXiv preprint arXiv:2109.14236* (2021).

[49] 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 Extension of Shamir's Secret Sharing

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:

- $\mathsf{SS.expoRecon}((g^{s_1}, X_1), \ldots, (g^{s_n}, X_n), t) = \{g^s, g^{a_1}, \ldots, g^{a_{t-1}}\}$: With the shares $g^{s_1}, \ldots, g^{s_n}$, it returns the secret and the polynomial coefficients of the Shamir secret sharing in the exponent. More precisely, it returns $\{g^s, g^{a_1}, \ldots, g^{a_{t-1}}\}$ such that for $f(x) = s + a_1 x + \ldots + a_{t-1} x^{t-1}$, $f(X_i) = s_i$ for $i \in [n]$. This function can be implemented without knowing $s_1, \ldots, s_n$ by performing all the linear operations of function SS.recon in the exponent.
- $\mathsf{SS.expoShare}(g^s, g^{a_1}, \ldots, g^{a_{t-1}}, X) = g^{s_X}$: with the coefficients of the polynomial in the exponent, it returns a new share in the exponent at index X. More precisely, it returns $g^{s_X} = g^s \cdot (g^{a_1})^X \cdot \ldots \cdot (g^{a_{t-1}})^{X^{t-1}}$. This function can also be implemented without knowing the exponents $s, a_1, \ldots, a_{t-1}$.

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 [14]. 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 Δ 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 Two Oracle Diffie-Hellman (2ODH) Assumption

Same as [11], we adopt Two Oracle Diffie-Hellman (2ODH) assumption:

Definition A.1. *Let $\mathcal{G}(k) \rightarrow (\mathbb{G}', g, q, H)$ be an efficient algorithm which samples a group $\mathbb{G}'$ of order q with generator g and a function $H : \{0,1\}^* \rightarrow \{0,1\}^k$. Consider the following probabilistic experiment, parameterized by a PPT adversary M, a bit b and a security parameter k:*

$\mathbf{2ODH} - \mathbf{Exp}^b_{\mathcal{G},M}(k)$:

(1) $(\mathbb{G}', g, q, H) \leftarrow \mathcal{G}(k)$
(2) $a \leftarrow \mathbb{Z}_q; A \leftarrow g^a$
(3) $b \leftarrow \mathbb{Z}_q; B \leftarrow g^b$
(4) if $b = 1$, $s \leftarrow H(g^{ab})$, *else* $s \xleftarrow{\$} \{0,1\}^k$
(5) $M^{O_a(\cdot), O_b(\cdot)}(\mathbb{G}', g, q, H, A, B, s) \rightarrow b'$
(6) Output 1 *if* $b = b'$, 0 *otherwise.*

where $O_a(X)$ returns $H(X^a)$ on any $X \neq B$ (and an error on input B) and similarly $O_b(X)$ returns $H(X^b)$ on any $X \neq A$. The advantage of the adversary is defined as

$$Adv^{2ODH}_{\mathcal{G},M}(k) :=$$
$$|Pr[\mathbf{2ODH} - \mathbf{Exp}^1_{\mathcal{G},M}(k) = 1] - Pr[\mathbf{2ODH} - \mathbf{Exp}^0_{\mathcal{G},M}(k) = 1]|$$

We say that the Two Oracle Diffie-Hellman assumption holds for $\mathcal{G}$ if for all PPT adversaries M, there exists a negligible function $\epsilon(\cdot)$ such that $Adv^{2ODH}_{\mathcal{G},M}(k) \leq \epsilon(k)$.

A.3 Class Groups Framework

Castagnos et al. [21] first introduced the idea of a composite order group whose order is unknown, but there is a subgroup of known order where the discrete algorithm is easy. Subsequently, this definiton has been refined and updated by a series of works which have successfully leveraged the group to create an assortment of cryptographic primitives [14, 17, 18, 22].

Broadly, the framework is defined by two functions - cl.gen, cl.solve with the former outputting a tuple of public parameters. The elements of this framework are the following:

- *Input Parameters*: κ_c is the computational security parameter, κ_s is a statistical security parameter, a prime p such that $p > 2^{\kappa_c}$, and uniform randomness ρ that is used by the cl.gen algorithm and is made public.
- *Groups*: $\hat{\mathbb{G}}$ is a finite multiplicative abelian group, $\mathbb{G}$ is a cyclic subgroup of $\hat{\mathbb{G}}$, $\mathbb{F}$ is a subgroup of $\mathbb{G}$, $\mathbb{G}^p = \{x^p, x \in \mathbb{G}\}$
- *Orders*: $\mathbb{F}$ has order p, $\hat{\mathbb{G}}$ has order $p \cdot \hat{s}$, $\mathbb{G}$ has order $p \cdot s$ such that s divides $\hat{s}$ and $gcd(p, \hat{s}) = 1, gcd(p, s) = 1$, $\mathbb{G}^p$ has order s and therefore $\mathbb{G} = \mathbb{F} \times \mathbb{G}^p$.
- *Generators*: f is the generator of $\mathbb{F}$, g is the generator of $\mathbb{G}$, and g_p is the generator of $\mathbb{G}^p$ with the property that $g = f \cdot g_p$
- *Upper Bound*: Only an upper bound $\bar{s}$ of $\hat{s}$ (and s) is provided.
- *Additional Properties*: Only encodings of $\hat{\mathbb{G}}$ can be recognized as valid encodings and $s, \hat{s}$ are unknown.
- *Distributions*: $\mathcal{D}$ (resp. $\mathcal{D}_p$) be a distribution over the set of integers such that the distribution $\{g^x : x \overset{\$}{\leftarrow} \mathcal{D}\}$ (resp. $\{g_p^x : x \overset{\$}{\leftarrow} \mathcal{D}_p\}$) is at most distance $2^{-\kappa_s}$ from the uniform distribution over $\mathbb{G}$ (resp. $\mathbb{G}^p$).
- *Additional Group and its properties*: $\hat{\mathbb{G}}^p = \{x^p, x \in \hat{\mathbb{G}}\}$, $\hat{\mathbb{G}}$ factors as $\hat{\mathbb{G}}^p \times \mathbb{F}$.[1] Let $\bar{\omega}$ be the group exponent of $\hat{\mathbb{G}}^p$. Then, the order of any $x \in \hat{\mathbb{G}}^p$ divides $\bar{\omega}$.[2]

Remark. *The motivations behind these additional distributions are as follows. One can efficiently recognize valid encodings of elements in $\hat{\mathbb{G}}$ but not $\mathbb{G}$. Therefore, a malicious adversary can run our constructions by inputting elements belonging to $\hat{\mathbb{G}}^p$ (rather than in $\mathbb{G}^p$). Unfortunately, this malicious behavior cannot be detected which allows to obtain information on the sampled exponents modulo $\bar{\omega}$ (the group exponent of $\hat{\mathbb{G}}^p$). By requiring the statistical closeness of the induced distribution to uniform in the aforementioned groups allows flexibility in proofs. Note that the assumptions do not depend on the choice of these two distributions. Further, the order s of $\mathbb{G}^p$ and group exponent $\bar{\omega}$ of $\hat{\mathbb{G}}^p$ are unknown and the upper bound $\bar{s}$ is used to instantiate the aforementioned distribution. Specifically, looking ahead we will set $\mathcal{D}_p$ to be the uniform distribution over the set of integers $[B]$ where $B = 2^{\kappa_s} \cdot \bar{s}$. Using Lemma A.2, we get that the distribution is less than $2^{-\kappa_s}$ away from uniform distribution in $\mathbb{G}^p$. In our constructions we will set $\kappa_s = 40$. We will make this sampling more efficient for our later constructions. We refer the readers to Tucker [45, §3.1.3, §3.7] for more discussions about this instantiation. Finally, as stated we will also set $\hat{\mathcal{D}} = \mathcal{D}$ and $\hat{\mathcal{D}}_p = \mathcal{D}_p$.*

We also have the following lemma from Castagnos, Imbert, and Laguillaumie [19] which defines how to sample from a discrete Gaussian distribution.

Lemma A.2. *Let $\mathbb{G}$ be a cyclic group of order n, generated by g. Consider the random variable X sampled uniformly from $\mathbb{G}$; as such it satisfies* $\Pr[X = h] = \frac{1}{n}$ *for all $h \in \mathbb{G}$. Now consider the random variable Y with values in $\mathbb{G}$ as follows: draw y from the discrete Gaussian distribution $\mathcal{D}_{\mathbb{Z},\sigma}$ with $\sigma \geq n\sqrt{\frac{\ln(2(1+1/\epsilon))}{\pi}}$ and set $Y := g^y$. Then, it holds that:*

$$\Delta(X, Y) \leq 2\epsilon$$

Remark. *By definition, the distribution $\{g^x : x \overset{\$}{\leftarrow} \mathcal{D}\}$ is statistically indistinguishable from $\{g^y : y \overset{\$}{\leftarrow} \{0, \ldots, p \cdot s - 1\}\}$. Therefore, it follows that $\{x \mod p \cdot s : x \overset{\$}{\leftarrow} \mathcal{D}\}$ is statistically indistinguishable from $\{x : x \overset{\$}{\leftarrow} \{0, \ldots, p \cdot s - 1\}\}$. Similarly, $\{x \mod s : x \overset{\$}{\leftarrow} \mathcal{D}_p\}$ is statistically indistinguishable from $\{x : x \overset{\$}{\leftarrow} \{0, \ldots, s - 1\}\}$. Furthermore, sampling a value x corresponding to $\mathcal{D}$ is statistically indistinguishable from the uniform distribution in $\{0, \ldots, s-1\}$ because s divides $p \cdot s$.*

Definition A.3 (Class Group Framework). *The framework is defined by two algorithms* (cl.gen, cl.solve) *such that:*

- $\mathsf{pp} = (p, \kappa_c, \kappa_s, \bar{s}, f, g_p, \hat{\mathbb{G}}, \mathbb{F}.\mathcal{D}, \mathcal{D}_p, \hat{\mathcal{D}}, \hat{\mathcal{D}}_p, \rho) \overset{\$}{\leftarrow} \mathsf{cl.gen}(1^{\kappa_c}, 1^{\kappa_s}, p; \rho)$
- *The* DL *problem is easy in $\mathbb{F}$, i.e., there exists a deterministic polynomial algorithm* cl.solve *that solves the discrete logarithm problem in $\mathbb{F}$:*

$$\Pr\left[x = x' \;\middle|\; \begin{array}{l} \mathsf{pp} =\overset{\$}{\leftarrow} \mathsf{cl.gen}(1^{\kappa_c}, 1^{\kappa_s}, p; \rho) \\ x \overset{\$}{\leftarrow} \mathbb{Z}/p\mathbb{Z}, X = f^x; \\ x' \leftarrow \mathsf{cl.solve}(\mathsf{pp}, X) \end{array}\right] = 1$$

When dealing with groups of known order, one can sample elements in a group $\mathbb{G}$ easily by merely sampling exponents modulo the group order and then raising the generator of the group to that exponent. Unfortunately, note that here neither the order of $\mathbb{G}$ (i.e., ps) nor that of $\mathbb{G}^p$ (i.e. s) is known. Therefore, we instead use the knowledge of the upper-bound $\bar{s}$ of s to instantiate the distributions $\mathcal{D}$ and $\mathcal{D}_p$ respectively. This choice of choosing from the distributions $\mathcal{D}$ and $\mathcal{D}_p$ respectively allows for flexibility of various proofs.

A.4 Ideal Functionality and Privacy Definitiion

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, $\mathsf{Ideal}^{\delta}_{\{x_i^k\}_{i \in \mathcal{U} \setminus C, k}}$ 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:

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

[1] Recall that p and $\hat{s}$ are co-prime.

[2] This follows from the property that the exponent of a finite Abelian group is the least common multiple of its elements.

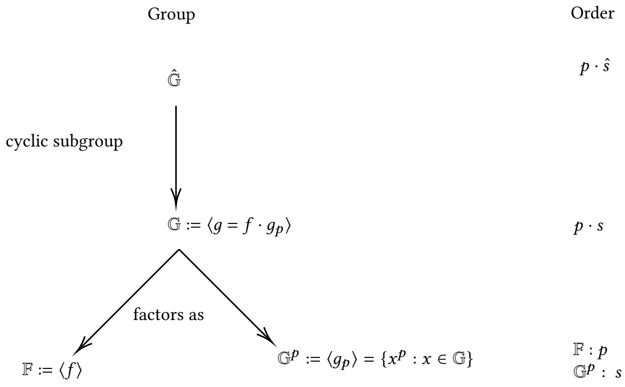
Group
Order
$\hat{\mathbb{G}}$
$p \cdot \hat{s}$
cyclic subgroup
$\mathbb{G} := \langle g = f \cdot g_p \rangle$
$p \cdot s$
factors as
$\mathbb{F} := \langle f \rangle$
$\mathbb{G}^p := \langle g_p \rangle = \{x^p : x \in \mathbb{G}\}$
$\mathbb{F} : p$
$\mathbb{G}^p : s$

Figure 6: A brief overview of the class group framework we employ. Here, $\hat{\mathbb{G}}$ is a finite multiplicative abelian group. $\hat{s}, p$ are co-prime and s divides $\hat{s}$. Further, $\hat{s}$ is unknown but an upperbound $\bar{s}$ is known. Finally, only encodings of $\hat{\mathbb{G}}$ are recognizable. Therefore, specifically the assumption we will rely on is that one cannot identify if a given group element belongs to $\mathbb{G}$ or $\mathbb{G}^p$. The last property we will rely on is that discrete logarithm is efficient in the subgroup $\mathbb{F}$.

Definition A.4 (Privacy against Semi-honest/Malicious Adversary). *Let K, n be integer parameters. Let Π 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 Π* guarantees privacy against γ fraction of semi-honest/malicious adversary with δ offline rate *if there exists a PPT simulator* 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 $C \subset \mathcal{U}$ with $|C| < \gamma n$ 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.,*

$$\mathsf{REAL}_C^{\mathcal{U},K}(M_C, \{x_i^k\}_{i\in\mathcal{U}\setminus C, k\in[K]}) \approx_c \mathsf{SIM}^{\mathcal{U},K,\mathsf{Ideal}^{\delta}_{\{x_i^k\}_{i\in\mathcal{U}\setminus C,k\in[K]}}}(M_C)$$

A.5 Hypergeometric Distribution

The hypergeometric distribution $X \sim \mathsf{HyperGeom}(N, m, n)$ 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 same tail bounds for $X \sim \mathsf{HyperGeom}(N, m, n)$ used in [10]:

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

B INTUITION OF THE TWO PROTOCOLS

We include the graphs high-levelly depicting the Setup phase and the Aggregation phase of MicroSecAgg$_{gDL}$ in this section.

C SECURITY PROOFS FOR MicroSecAgg$_{gDL}$

C.1 Proof of Group Properties

Proof. (**for Lemma 4.4**) We first prove that constraint 1 in Definition 4.3 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 $X \sim \mathsf{HyperGeom}(n-1, \gamma n, N)$. By the tail bound of the hypergeometric distribution,

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

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 $Y \sim \mathsf{HyperGeom}(n - 1, \delta n, N)$. 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.

□

C.2 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 $t - |G_d \cap C|$ 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, sk_i^{-1} and 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 sk_i^{-1} and sk_i^1 is hidden from the server.*

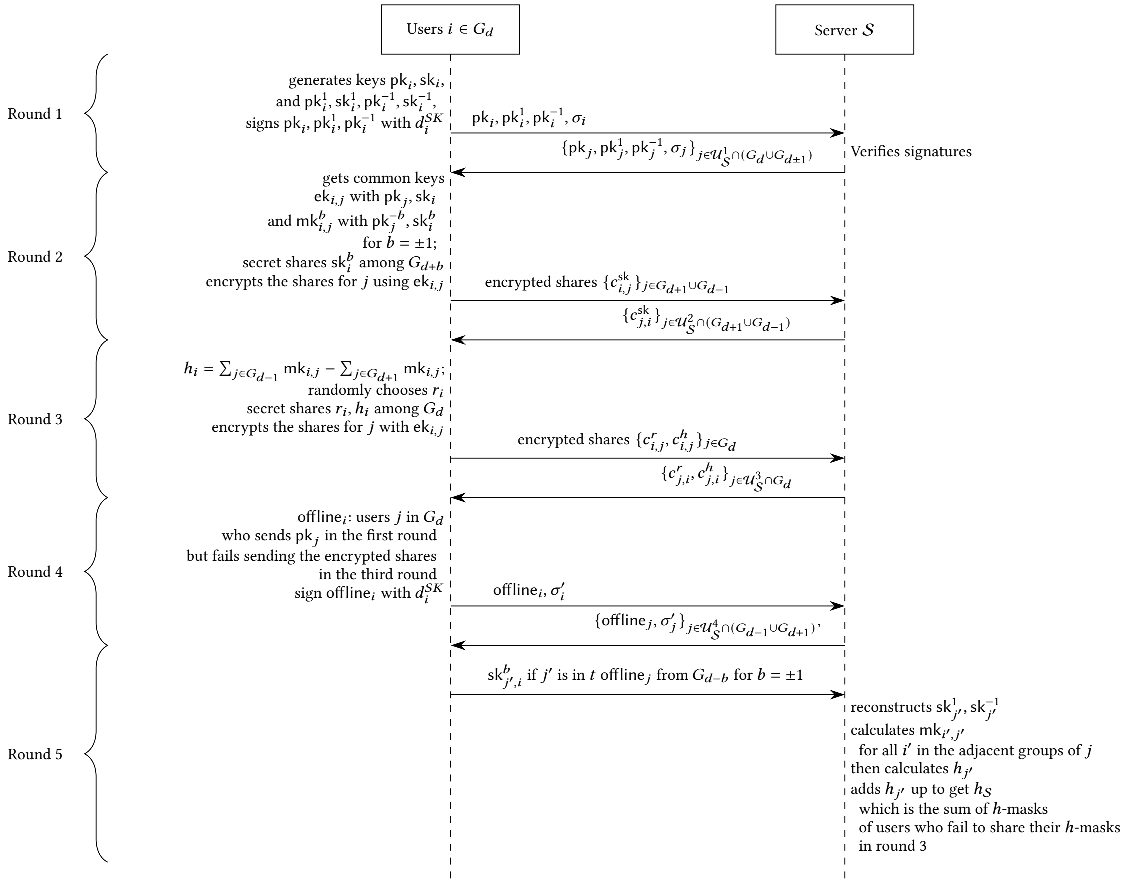
Users $i \in G_d$
Server $\mathcal{S}$
Round 1
generates keys $\mathsf{pk}_i, \mathsf{sk}_i$,
and $\mathsf{pk}_i^1, \mathsf{sk}_i^1, \mathsf{pk}_i^{-1}, \mathsf{sk}_i^{-1}$,
signs $\mathsf{pk}_i, \mathsf{pk}_i^1, \mathsf{pk}_i^{-1}$ with d_i^{SK}
$\mathsf{pk}_i, \mathsf{pk}_i^1, \mathsf{pk}_i^{-1}, \sigma_i$
$\{\mathsf{pk}_j, \mathsf{pk}_j^1, \mathsf{pk}_j^{-1}, \sigma_j\}_{j \in \mathcal{U}_S^1 \cap (G_d \cup G_{d\pm1})}$
Verifies signatures
Round 2
gets common keys
$\mathsf{ek}_{i,j}$ with $\mathsf{pk}_j, \mathsf{sk}_i$
and $\mathsf{mk}_{i,j}^b$ with $\mathsf{pk}_j^{-b}, \mathsf{sk}_i^b$
for $b = \pm 1$;
secret shares sk_i^b among G_{d+b}
encrypts the shares for j using $\mathsf{ek}_{i,j}$
encrypted shares $\{c_{i,j}^{\mathsf{sk}}\}_{j \in G_{d+1} \cup G_{d-1}}$
$\{c_{j,i}^{\mathsf{sk}}\}_{j \in \mathcal{U}_S^2 \cap (G_{d+1} \cup G_{d-1})}$
Round 3
$h_i = \sum_{j \in G_{d-1}} \mathsf{mk}_{i,j} - \sum_{j \in G_{d+1}} \mathsf{mk}_{i,j}$;
randomly chooses r_i
secret shares r_i, h_i among G_d
encrypts the shares for j with $\mathsf{ek}_{i,j}$
encrypted shares $\{c_{i,j}^r, c_{i,j}^h\}_{j \in G_d}$
$\{c_{j,i}^r, c_{j,i}^h\}_{j \in \mathcal{U}_S^3 \cap G_d}$
Round 4
$\mathsf{offline}_i$: users j in G_d
who sends pk_j in the first round
but fails sending the encrypted shares
in the third round
sign $\mathsf{offline}_i$ with d_i^{SK}
$\mathsf{offline}_i, \sigma_i'$
$\{\mathsf{offline}_j, \sigma_j'\}_{j \in \mathcal{U}_S^4 \cap (G_{d-1} \cup G_{d+1})}$,
Round 5
$\mathsf{sk}_{j',i}^b$ if j' is in t $\mathsf{offline}_j$ from G_{d-b} for $b = \pm 1$
reconstructs $\mathsf{sk}_{j'}^1, \mathsf{sk}_{j'}^{-1}$
calculates $\mathsf{mk}_{i',j'}$
for all i' in the adjacent groups of j
then calculates $h_{j'}$
adds $h_{j'}$ up to get h_S
which is the sum of h-masks
of users who fail to share their h-masks
in round 3

Figure 7: An overview of the Setup phase of MicroSecAgg$_{gDL}$

Proof. If 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 $t - |C \cap G_{d+1}|$ 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 $t - |C \cap G_d|$ 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. □

Common online set of a group. For some iteration k of the Aggregation phase, we say a user set $O_d \subseteq G_d$ is a *common online set of group d* if some honest user in G_d receives at least t valid signatures on 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 $2t > n/B + |C \cap G_d|$, there is at most one common online set O_d for each group d in every iteration.*

Proof. For the sake of contradiction, assume that there are two common online sets O_d and O_d' for group d in some iteration. In other words, in the joint view of the honest users in group d, there are at least t valid signatures on both sets. At most c signatures are from corrupt users. As $t > \frac{2}{3}n$ and $c < n - t < \frac{1}{3}n$, we have $2(t - c) > n - c$. In other words, there must be an honest user signing both on O_d and O_d', which is not possible. □

Now, we give the proof for Theorem 4.6.

Proof. **(of Theorem 4.6)** 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 (σ, η, 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 $(\mathsf{pk}_j, \sigma_j)$ from the server, and verifies the signatures as described in Algorithm 1, except that the simulator additionally aborts if some honest user i receives

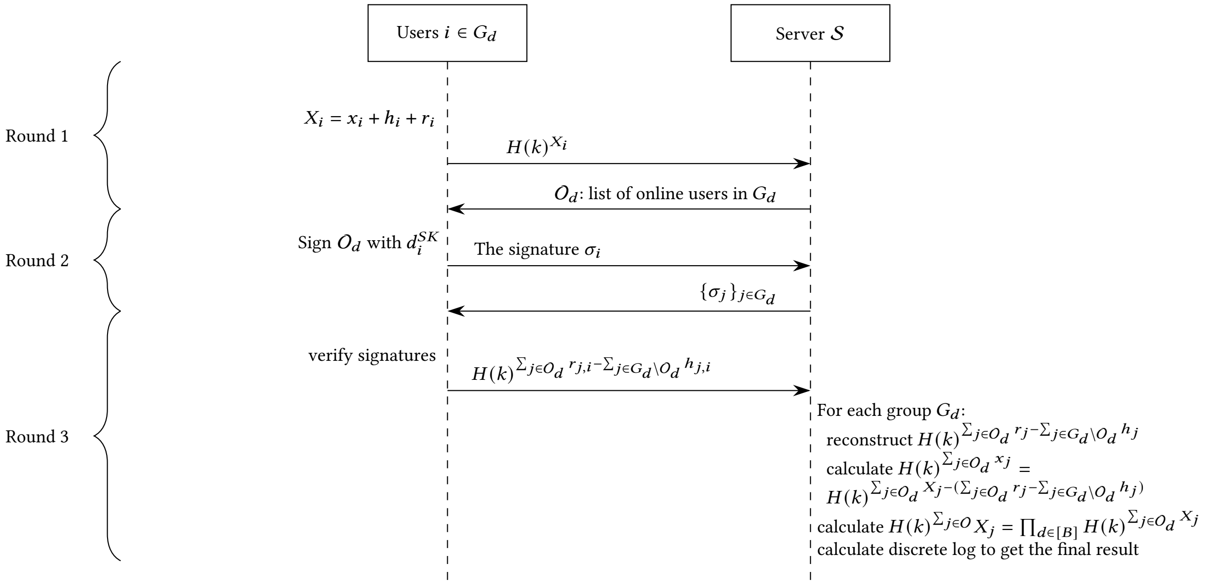
Users $i \in G_d$
Server $\mathcal{S}$
Round 1
Round 2
Round 3
$X_i = x_i + h_i + r_i$
$H(k)^{X_i}$
O_d: list of online users in G_d
Sign O_d with d_i^{SK}
The signature σ_i
$\{\sigma_j\}_{j \in G_d}$
verify signatures
$H(k)^{\sum_{j \in O_d} r_{j,i} - \sum_{j \in G_d \setminus O_d} h_{j,i}}$
For each group G_d:
reconstruct $H(k)^{\sum_{j \in O_d} r_j - \sum_{j \in G_d \setminus O_d} h_j}$
calculate $H(k)^{\sum_{j \in O_d} x_j} =$
$H(k)^{\sum_{j \in O_d} X_j - (\sum_{j \in O_d} r_j - \sum_{j \in G_d \setminus O_d} h_j)}$
calculate $H(k)^{\sum_{j \in O} X_j} = \prod_{d \in [B]} H(k)^{\sum_{j \in O_d} X_j}$
calculate discrete log to get the final result

Figure 8: An overview of the Aggregation phase of MicroSecAgg$_{gDL}$

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 $j \in \mathcal{U}_i^1 \cap C$, an honest user i stores $\mathsf{ek}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j, \mathsf{sk}_i)$. For each pair of honest users i, j, the simulator uniformly randomly chooses a symmetric encryption key $\mathsf{ek}^*_{i,j}$, and sets $\mathsf{ek}^*_{j,i} = \mathsf{ek}^*_{i,j}$. For each $j \in \mathcal{U}_i^1 \cap (G_{d-1} \cup G_{d+1})$, each honest user i also stores $\mathsf{mk}_{i,j}$. Then it follows the protocol, except that for honest user $j \in G_{d-1}$, instead of encrypting the share $\mathsf{sk}^{-1}_{i,j}$, it encrypts some dummy value by $c^{\mathsf{sk}^{-1*}}_{i,j} \leftarrow \mathsf{AE.enc}(0, \mathsf{ek}^*_{i,j})$, 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 $\mathsf{sk}^{-1}_{j,i}$ (or $\mathsf{sk}^1_{j,i}$) 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 $\mathsf{mk}^*_{i,j}$ for each pair of honest users $i \in G_d$ and $j \in G_{d+1}$, and let $\mathsf{mk}^*_{i,j} = \mathsf{mk}_{i,j}$ for each honest user $i \in G_d$ and each corrupt user $j \in G_{d\pm1}$. Then each honest user i calculates $h^*_i = \sum_{j \in \mathcal{U}_i^2 \cap G_{d-1}} \mathsf{mk}^*_{i,j} - \sum_{\mathcal{U}_i^2 \cap G_{d+1}} \mathsf{mk}^*_{i,j}$. 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 $\mathsf{ek}^*_{i,j}$.
- **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 $\mathsf{offline}_i$ and sends the list and the signature to the server.
- **Round 5**: On receiving $(\mathsf{offline}_j, \sigma_j)$ from the server, the user i aborts if any signature is invalid. The simulator additionally aborts if any honest user i receives $\mathsf{offline}'_j \neq \mathsf{offline}_j$ 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 $\mathsf{sk}^1_{i,j}$ (or $\mathsf{sk}^{-1}_{i,j}$) are sent to the server from honest users $j \in G_{d+1}$ (or æ $\in G_{d-1}$) and puts such users i in a user list $\mathsf{offline}_{\mathsf{SIM}}$.

- 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 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 O_d it receives in Round 2.
    - ∗ If there are such users in every group, then let $O = \cup_{d \in [B]} O_d$, the simulator queries the ideal functionality to get $w = \mathsf{Ideal}(O \backslash C, k)$. The simulator then uniformly randomly chooses w^*_i for $i \in O$ under the restriction that $\sum_{i \in O \backslash C} w^*_i = w$. For each iteration $k \in [K]$, it uniformly randomly picks h^*_i under the constraint that $\sum_{i \backslash \mathsf{offline}_{\mathsf{SIM}}} h^*_i = \sum_{i \backslash \mathsf{offline}_{\mathsf{SIM}}} h_i$. 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 $j \in G_d \backslash C$ based on $r_{i,j}$ for $j \in G_d \cap C$ that have already been sent to the corrupt users in the Setup phase. Let $r^*_{i,j} = r_{i,j}$ for $i \in C$. The simulator sends ζ^{r*}_i

and ζ_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 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 ζ_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 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.
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 $\mathsf{ek}^*{}_{i,j}$ instead of $\mathsf{ek}_{i,j}$ 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 as introduced in Appendix A.2.

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 $\mathsf{ek}^*_{i,j}$.
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 $j \in \mathcal{U} \backslash C$ making sure that $r_i^* = \mathsf{SS.recon}(\{r^*_{i,j}, j\}_{j \in \mathcal{U}_i \backslash C}, \{r_{i,j}, j\}_{j \in C})$. For corrupt users $j \in C$, let $r^*_{j,i} = r_{j,i}$. Then each honest user $i \in G_d$ who receives the common online set O_d with at least t valid signatures calculates $\zeta_i^* = H_c(k)^{\sum_{j \in O_d} r^*_{j,i} - \sum_{j \in \mathcal{U}_i \backslash O_d} h_{j,i}}$ and sends ζ_i^* to the server.

Hyb6 In this hybrid, in the third round of each iteration, for each honest user $i \in G_d$ not included in 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 atmost one unique O_d and if an honest user is not included in O_d, the share $r^*_{i,j}$ for honest user i not in O will not be included in ζ_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 $i \in O_d \backslash C$, instead of setting $r_i^* = X_i^* - x_i - h_i$, SIM randomly picks r_i^* under the constraint that $\sum_{i \in O_d \backslash C} r_i^* = \sum_{i \in O_d \backslash C} X_i^* - \sum_{i \in O_d \backslash C} (x_i + h_i)$.
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 $\mathsf{mk}^*_{i,j}$ for each pair of honest users $i, j \notin \mathsf{offline}_{\mathsf{SIM}}$ from two adjacent groups. For honest user $i \in G_d \backslash \mathsf{offline}_{\mathsf{SIM}}$ and user $j \in (C \cup \mathsf{offline}_{\mathsf{SIM}}) \cap G_{d \pm 1}$, let $\mathsf{mk}^*_{i,j} = \mathsf{mk}_{i,j}$ obtained in the Setup phase. The simulator then uses $\mathsf{mk}^*_{i,j}$ to calculates h_i^* for each honest user $i \notin \mathsf{offline}_{\mathsf{SIM}}$ 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 $\mathsf{sk}_i^{\pm 1}$ for any honest user $i \notin \mathsf{offline}_{\mathsf{SIM}}$ (guaranteed by the security of Shamir secret sharing). Thus, $\mathsf{mk}_{i,j}$ for honest users $i, j \notin \mathsf{offline}_{\mathsf{SIM}}$ is indistinguishable from $\mathsf{mk}^*_{i,j}$ chosen uniformly randomly.

Hyb9 Instead of choosing $\mathsf{mk}^*_{i,j}$ for each pair of honest users $i, j \notin \mathsf{offline}_{\mathsf{SIM}}$, the simulator just choose h_i^* for each honest user $i \notin \mathsf{offline}_{\mathsf{SIM}}$ uniformly at random under the constraint that $\sum_{i \notin \mathsf{offline}_{\mathsf{SIM}}} h_i^* = \sum_{i \notin \mathsf{offline}_{\mathsf{SIM}}} h_i$.
By Lemma C.7, 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 $i \notin \mathsf{offline}_{\mathsf{SIM}}$ under the same constraint $\sum_{i \notin \mathsf{offline}_{\mathsf{SIM}}} h_i^* = \sum_{i \notin \mathsf{offline}_{\mathsf{SIM}}} h_i$ at the beginning of each iteration.
This hybrid is indistinguishable from the previous one.

Hyb11 When O_d exists for each $d \in [B]$, instead using the constraint $\sum_{i \in O_d \backslash C} r_i^* = \sum_{i \in O_d \backslash C} X_i^* - \sum_{i \in O_d \backslash C} (x_i + h_i^*)$ for each $d \in [B]$ to randomly pick r_i^* for each $i \in O_d \backslash C$, the simulator uses the constraint $\sum_{i \in O \backslash C} r_i^* = \sum_{i \in O \backslash C} X_i^* - \sum_{i \in O \backslash C} (x_i + h_i^*)$.
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 $\sum_{i \notin \mathsf{offline}_{\mathsf{SIM}}} h_i^* = \sum_{i \notin \mathsf{offline}_{\mathsf{SIM}}} h_i$, then it uniformly randomly chooses h_i^{**} for each $i \in O_d \backslash C$ under the constraint that $\sum_{i \in O_d \backslash C} h_i^{**} = \sum_{i \in O_d \backslash C} h_i^*$. The h_i^{**} for other honest user $i \notin \mathsf{offline}_{\mathsf{SIM}}$ are randomly chosen

such that $\sum_{i \notin \mathsf{offline}_{\mathsf{SIM}}} h_i^{**} = \sum_{i \notin \mathsf{offline}_{\mathsf{SIM}}} h_i$. Then r_i^* is chosen under the constraint that $\sum_{i \in O_d \setminus C} r_i^* = \sum_{i \in O_d \setminus C} X_i^* - \sum_{i \in O_d \setminus C}(x_i + h_i^{**})$. 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 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 $O_{d'}$ 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 $\sum_{i \in O \setminus C} x_i$, the simulator queries the ideal functionality by $w = \mathsf{Ideal}(O, k)$ if there is a common online set 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_C in the real execution is computationally indistinguishable from the view in the simulated execution. □

Lemma C.3 (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 $a, b_1, \ldots, b_t, b'_1, \ldots b'_n \in \mathbb{Z}_q$, the following two distributions are computationally indistinguishable:*

$$(g^a, g^{b_1}, \ldots, g^{b_n}, g^{ab_1}, \ldots, g^{ab_n}) \tag{1}$$

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

Proof. We define hybrids $\mathsf{Hyb}_i = (g^a, g^{b_1}, \ldots, g^{b_n}, g^{ab_1}, \ldots, g^{ab_i}, g^{ab'_{i+1}}, \ldots, g^{ab'_n})$ 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 Hyb_{i-1} and 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 $b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_n$ and $b'_{i+1}, \ldots, b'_n$. Then it invokes $\mathcal{A}$ with the tuple

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

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 Hyb_i, and when $C = g^{ab'}$ for a uniformly random b', the distribution $\mathcal{D}$ feeds $\mathcal{A}$ is the distribution of 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 Hyb_{i-1} and Hyb_i are computationally indistinguishable.

As the distribution (1) is the same as Hyb_n, and the distribution (2) is the same as Hyb_0, the two distributions in the lemma are computationally indistinguishable. □

Lemma C.4. *Let n, t, n_C, K be integer parameters, $n_C \leq n - t < t$. 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., $\{w_i\}_{i \in [n]} \leftarrow \mathsf{SS.share}(w, [n], t)$.*

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

The following two distributions are computationally indistinguishable:

$$w, \{w_i\}_{i \in [n-n_C+1, n]}, \{H(k)^{w_i}\}_{i \in [n-n_C], k \in [K]} \tag{3}$$

$$w, \{w_i\}_{i \in [n-n_C+1, n]}, \{H(k)^{w_i^k}\}_{i \in [n-n_C], k \in [K]} \tag{4}$$

Proof. We define a hybrid Hyb_0 to be identical to the distribution (3), and a sequence of hybrids Hyb_k for $k \in [K]$ as following: Hyb_k is the same as Hyb_{k-1} except that in Hyb_k, $H(k)^{w_i}_{i \in [n-n_C]}$ are substituted with $H(k)^{w_i^k}_{i \in [n-n_C]}$. Thus, Hyb_K is identical to distribution (4). Then we prove that any two adjacent hybrids Hyb_{k_0-1} and Hyb_{k_0} for $k_0 \in [K]$ are computationally indistinguishable.

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

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

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

$$\mathcal{D}'(A, B_1, ..., B_{t-n_C-1}, C_1, ..., C_{t-n_C-1}):$$

$\mathcal{D}'$ uniformly randomly picks $\{w_i\}_{i \in [n-n_C+1, n]}$ as the second part of the input to $\mathcal{D}$, and calculates $W_i = \mathsf{SS.expoRecon}((g^w, 0), \{B_j, j\}_{j \in [t-n_C-1]}, \{g^{w_j}, j\}_{j \in [n-n_C+1, n]}, t, i)$ for $i \in [t - n_C, n - n_C]$. 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 $w = \mathsf{SS.recon}(\{w_i^k\}_{i \in [n-n_C]}, \{w_i\}_{i \in [n-n_C]}, t)$ and it sets $Z_i^k = g^{w_i^k s_k}$ for $i \in [n - n_C]$;
- For $k \in [k_0 + 1, K]$, it sets $Z_i^k = B_i^{s_k}$ for $i \in [t - n_C - 1]$ and $Z_i^k = W_i^{s_k}$ for $i \in [t - n_C, n - n_C]$;
- Then it sets $H(k_0) = A$, $Z_i^{k_0} = C_i$ for $i \in [t - n_C - 1]$, and runs

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

for $j \in [t - n_C, n - n_C]$.

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 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 Hyb_{k_0}. Thus, $\mathcal{D}'$ wins with the probability that $\mathcal{D}$ succeeds. By Lemma C.3, such a distinguisher $\mathcal{D}'$ does not exist. Thus, we have a contradiction. □

Lemma C.5. *Let n, t, n_C, K be integer parameters, $n_C \leq n - t < t$. Let $x_1, \ldots, x_{n-n_C}$ 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., $\{x_{i,j}\}_{j \in [n]} \leftarrow \mathsf{SS.share}(x_i, [n], t)$ for $i \in [n - n_C]$.*

For each $k = 1, \ldots, K$, let $y_1^k, \ldots, y_{n-n_C}^k$ 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 $y_i^k = \mathsf{SS.recon}(\{y_{i,j}^k, j\}_{j\in[n-n_C]}, \{x_{i,j}^k, j\}_{j\in[n-n_C+1,n]}, t)$. 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:

$$\{x_{i,j}\}_{i\in[n-n_C],j\in[n-n_C+1,n]}, \{\{H(k)^{x_i}\}_{i\in[n-n_C]}, \{H(k)^{x_{i,j}}\}_{i,j\in[n-n_C]}\}_{k\in[K]} \tag{5}$$

$$\{x_{i,j}\}_{i\in[n-n_C],j\in[n-n_C+1,n]}, \{\{H(k)^{y_i^k}\}_{i\in[n-n_C]}, \{H(k)^{y_{i,j}^k}\}_{i,j\in[n-n_C]}\}_{k\in[K]} \tag{6}$$

Proof. We prove the indistinguishability with a sequence of hybrids. Let Hyb_0 equal to the distribution (5). Then for each $k \in [K]$, Hyb_k is the same as 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 susbstituted 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, Hyb_K is identical to the distribution (6) in the lemma. Then between Hyb_k and Hyb_{k+1} for $k \in [0, K-1]$, we additionally define $\mathsf{Hyb}_{k,0} = \mathsf{Hyb}_k$ and a sequence of hybrids $\mathsf{Hyb}_{k,i}$ for $i \in [n - n_C]$: $\mathsf{Hyb}_{k,i}$ is the same as $\mathsf{Hyb}_{k,i-1}$ except that $H(k)^{x_i}, \{H(k)^{x_{i,j}}\}_{j\in[n-n_C]}$ in $\mathsf{Hyb}_{k,i-1}$ is substituted with $H(k)^{y_i^k}, \{H(k)^{y_{i,j}^k}\}_{j\in[n-n_C]}$. Note that $\mathsf{Hyb}_{k,n-n_C}$ is identical to Hyb_{k+1}.

Then we prove that the two adjacent hybrids Hyb_{k_0,i_0-1} and 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

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

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 $\{x_{i,j}\}_{j\in[n]} \leftarrow \mathsf{SS.share}(x_i, [n], t)$. For $k < k_0$, it also it unifromly 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 $Z_{k_0,i} = A^{y_i^k}$ and $Z_{k_0,i,j} = A^{y_{i,j}^k}$; 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

$$X_{i_0}, C_1, \ldots C_{t-1} = \mathsf{SS.expoRecon}((B_1, 1), \ldots (B_{t-n_C}, t - n_C), (g^{x_{i_0,n-n_C+1}}, n - n_C + 1), \ldots, (g^{x_{i_0,n}}, n), t),$$

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

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

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 $C_i = g^{ab_i}$ for $i \in [t - c]$, the distribution of the input for $\mathcal{D}$ is exactly the same as Hyb_{k_0,i_0-1}; when $C_=g^{c_i}$ for uniformly random c_i for $i \in [t - c]$, the distribution of the input for $\mathcal{D}$ is exactly the same as Hyb_{k_0,i_0}. Thus, $\mathcal{D}$ win the games with the same probability as $\mathcal{A}$ distinguishes between Hyb_{k_0,i_0-1} and Hyb_{k_0,i_0}. However, by Lemma C.3, there is no such a distinguisher $\mathcal{D}'$. Thus, we have a contradiction.

□

Lemma C.6. *Let n, t, n_C, K be integer parameters, $n_C \leq n - t < t$. Let $x_1, \ldots, x_{n-n_C}$ 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., $\{x_{i,j}\}_{j\in[n]} \leftarrow \mathsf{SS.share}(x_i, [n], t)$ for $i \in [n - n_C]$.*

For each $k = 1, \ldots, K$, let $y_1^k, \ldots, y_{n-n_C}^k$ 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

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

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:

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

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

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 calculates t-out-of-n shares of w by $\{w_j\}_{j\in[n]} \leftarrow \mathsf{SS.share}(w, [n], t)$ 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 calculates $x_{n-n_C,j} = w_j - \sum_{i\in[n-n_C-1]} x_{i,j}$ 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 calculates w_j^k for $j \in [n - n_C]$ such that $w = \mathsf{SS.recon}(\{w_j^k, j\}_{j\in[n-C]}, \{w_j, j\}_{j\in[n-n_C+1,n]}, t)$, and we calculates $y_{n-n_C,j}^k = w_{n-n_C}^k - \sum_{i\in[n-n_C-1]} x_{i,j}$. We substitutes $H(k)^{x_{n-n_C}}$ 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.4, 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, calculates $\{y_{i,j}^k\}_{j \in [n-n_C]}$ such that $y_i = \mathsf{SS.recon}(\{y_{i,j}^k, j\}_{j \in [n-n_C]}, \{x_{i,j}, j\}_{j \in [n-n_C+1,n]})$. Then we substitutes $H(k)^{x_i}$ 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 C.5, This one is indistinguishable from the previous one. This hybrid is also identical to distribution (8).

□

Lemma C.7. *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:*

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

This lemma can also be easily proved with induction.

D SECURITY PROOFS OF MICROSECAGG$_{CL}$

In this section, we discuss the privacy guarantee against semi-honest adversaries with a compromised server. The idea is similar to the security proofs of previous protocols.

We give a sketch of the proof with the simulator specification and hybrids which is very similar to those in the proof of Appendix C.2. For simplicity, here we consider the non-group version, i.e., instantiation of MicroSecAgg$_{DL}$ with class group rather than the generic cyclic group. Based on the security proof of this version, it is straightforward to apply the difference between the proofs of MicroSecAgg$_{DL}$ and MicroSecAgg$_{gDL}$ and prove the security of MicroSecAgg$_{CL}$.

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.

Fact D.1 (Unique Common Online Set). *When $t > \frac{2}{3}n$ and $|C| < \frac{1}{3}n$, 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, $|\mathcal{U}_1| \geq t - |C|$ and $|\mathcal{U}_2| \geq t - |C|$. Thus $|\mathcal{U}_1| + |\mathcal{U}_2| \geq 2t - 2|C| > \frac{4}{3}n - \frac{1}{3}n - |C| = n - |C|$, which is total number of honest users. Thus we have a contradiction as an honest user will only sign on one online set. □

We first define the behavior of a simulator SIM:

- In the Setup phase:
  - **Round 1**: For each honest user i, SIM generates a pair of encryption keys $(ek_{priv,i}, ek_{pub,i}) \leftarrow \mathsf{KA.gen}(pp)$, signs the public encryption key $ek_{pub,i}$ with sk_i, and sends $(ek_{pub,i}, \sigma_i)$ to the server, in which σ_i denotes the signature.
  - **Round 2**: Each honest user i receives $(\mathsf{pk}_j, \sigma_j)$ from the server, and verifies the signatures, 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 $j \in \mathcal{U}_i \cap C$, an honest user i stores $\mathsf{ek}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j, \mathsf{sk}_i)$. For each pair of honest users i, j, the simulator uniformly randomly chooses a symmetric encryption key $\mathsf{ek}_{i,j}^*$, and sets $\mathsf{ek}_{j,i}^* = \mathsf{ek}_{i,j}^*$.
  Then, for each corrupt user $j \in \mathcal{U}_i^1 \cap C$, user i uniformly randomly chooses $r_{i,j}$, encrypts it by $c_{i,j} \leftarrow \mathsf{AE.enc}(r_{i,j}, \mathsf{ek}_{i,j})$; for each honest user $j \in \mathcal{U}_i^1 \backslash C$, user i encrypts a dummy value by $c_{i,j} \leftarrow \mathsf{AE.enc}(\perp, \mathsf{ek}_{i,j}^*)$. 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 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 $w = \mathsf{Ideal}(\mathcal{O} \backslash C, k)$. Then for each $i \in \mathcal{O} \backslash C$, the simulator uniformly randomly samples w_i^* for $i \in \mathcal{O} \backslash C$ under the restriction $\sum_{i \in \mathcal{O} \backslash C} w_i^* = w$. Then for all honest users $i \in \mathcal{O}_i \backslash C$, the simulator uniformly randomly samples w_i^* for $i \in \mathcal{O}_i \backslash C$ under the restriction $\sum_{i \in \mathcal{O}_i \backslash C} w_i^* = w$. Moreover, the simulator SIM calculates the shares $R_{i,j}^*$ for all honest users in the online set $j \in \mathcal{O}_i \backslash C$ such that $\mathsf{SS.expoRecon}(\{R_{i,j}^*, j\}_{j \in \mathcal{O}_i \backslash C}, \{g_k^{r_{i,j}}\}_{j \in C}) = g_k^{X_i^*} / f^{w_i^*}$ where $r_{i,j}$ for $j \in C$ are the shares that have already been sent to the corrupt users in the Setup phase. Let $R_{j,i}^* = g^{r_{j,i}}$ for $j \in C$ 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 D.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 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 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 $\mathsf{ek}_{i,j}^*$ instead of $\mathsf{ek}_{i,j}$ 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 $c_{i,j}^r = \mathsf{AE.enc}(\mathsf{ek}_{i,j}^*, r_{i,j})$ sent between honest parties in the Setup phase in the previous hybrid with the encryption of a dummy value, i.e., ${c_{i,j}^r}^* = \mathsf{AE.enc}(\perp, \mathsf{ek}_{i,j}^*)$.

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 $j \in \mathcal{U}\backslash C$ making sure that $R_i^* = \mathsf{SS.expoRecon}(\{R_{i,j}^*, j\}_{j \in \mathcal{U}_i \backslash C}, \{g_k^{r_{i,j}}, j\}_{j \in C})$. For corrupt users $j \in C$, let $R_{j,i}^* = g_k^{r_{j,i}}$. Then each honest user i who receives O with at least t valid signatures calculates $\zeta_i^* = \prod_{j \in O_i} R_{j,i}^*$ and sends ζ_i^* to the server.

Hyb6 In this hybrid, in each iteration, if some honest user receives O with at least t valid signatures in the second round, then in the third round, for each user $i \in O\backslash C$, 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 $\prod_{i \in O_S \backslash C} R_i^* = \prod_{i \in O_S \backslash C} g_k^{X_i^*}/f^{x_i}$. The simulator then uses R_i^* to calculate the shares $\{R_{i,j}^*\}_{j \in \mathcal{U}\backslash C}$ for user $i \in O\backslash C$.

This hybrid is indistinguishable from the previous hybrid, as R_i^* are still uniformly random, and $\prod_{i \in O\backslash C} R_i^*$ that the server can reconstruct from the shares keeps the same.

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

This hybrid is identical to the previous one as there is only one unique O and if an honest user is not included in O, the share $R_{i,j}^*$ for $j \in \mathcal{U}\backslash C$ will not be included in any ζ_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 $\sum_{i \in O\backslash C} x_i$ to sample R_i^* for $i \in O\backslash C$, the simulator makes a query to the functionality Ideal with the user set $O\backslash C$ and iteration counter k and use the output value to sample random value R_i^* in every iteration with $|O\backslash C| \geq t - n_C$. 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.

E PERFORMANCE ANALYSIS

We include asymptotic performance analysis of MicroSecAgg$_{DL}$ (which is the same as MicroSecAgg$_{CL}$) and MicroSecAgg$_{gDL}$ in Appendix E.1 and Appendix E.2 respectively, and include more experiment results in Appendix E.4. In this section, we use n to denote the total number of users, L to denote the total length of input vectors, and use $R = 2^\ell$ to denote the size of the range of results, i.e., the sum of inputs the server learns in each iteration is assumed to be in the range $[R]$.

We include full comparison of asymptotic performance between different protocols in Table 2. One thing to notice here is that in practice different operations take different amounts of time. In BIK+17, BBG+20, and Flamingo, the multiplier L in the computation cost of both server and user sides come from extending the secrets to length L and adding up the vectors of length L, which is fast in practice; on the contrary, the L multiplier in the computation cost of MicroSecAgg$_{DL}$, MicroSecAgg$_{gDL}$, and MicroSecAgg$_{CL}$ comes from the fact that the server needs to do the reconstruction of shared secrets on the exponent and the user needs to do the exponentiation for every index of the input vector, which is more expensive. Besides, BIK+17, BBG+20, Flamingo, MicroSecAgg$_{gDL}$, and MicroSecAgg$_{CL}$ have part of the computation cost depending on the number of offline users. In Table 2, we assume that the offline rate is a constant, i.e., the number of offline users is $O(n)$. We include a more detailed analysis in Table 3.

E.1 Asymptotic Performance of MicroSecAgg$_{DL}$

E.1.1 Semi-Honest Protocol.

Communication. In the Setup phase, each user sends one public encryption key ($O(k)$) to the server and receives public encryption keys of all other users ($O(nk)$), 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 ($O(nk)$). This results in $O(nk)$ 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 $O(n^2k)$.

In the first round of the Aggregation phase, each user sends L group elements to the server ($O(kL)$) and receives the indicator of the online set O (n bits), which results in $O(kL) + n$ communication

	Computation cost			
Protocol	Server		User	
	Setup	Agg.	Setup	Agg.
BIK+17	–	$O(n^2 + (1-\delta)nL + \delta n^2 L)$	–	$O(n^2 + nL)$
BBG+20	–	$O(n \log^2 n + (1-\delta)nL + \delta nL \log n)$	–	$O(\log^2 n + L \log n)$
Flamingo	forwarding	$O(nL + \delta n \log^2 n + (1-\delta)n \log n)$	$O(\log^2 n)$	$O(L + n \log n)$
MicroSecAgg$_{DL}$	$O(n)$	$O(nL + 2^{\ell/2}L)$	$O(n^2)$	$O(nL)$
MicroSecAgg$_{gDL}$	$O(n + \delta n \log^2 n)$	$O(nL + 2^{\ell/2}L)$	$O(\log^2 n)$	$O(L \log n)$
MicroSecAgg$_{CL}$	$O(n + \delta n \log^2 n)$	$O(nL)$	$O(\log^2 n)$	$O(L \log n)$

Table 3: Total asymptotic computation cost for all rounds per aggregation with malicious security. n denotes the total number of users, L denotes the length of the input vector, and ℓ denotes the bit length of each element in the input vector. The overhead includes both received and sent messages. We assume the number of offline users is δn. In the line of Flamingo, "forwarding" means that the server only forwards the message for the users.

		Computation cost			
Protocol	rd	Server		User	
		Setup	Agg.	Setup	Agg.
BIK+17	– | 4	–	$O(n^2 L)$	–	$O(n^2 + nL)$
BBG+20	– | 4	–	$O(n \log^2 n + nL \log n)$	–	$O(\log^2 n + L \log n)$
Flamingo	2 | 2	forwarding	$O(nL + n \log^2 n)$	$O(\log^2 n)$	$O(L + n \log n)$
MicroSecAgg$_{DL}$	3 | 2	forwarding	$O(nL + 2^{\ell/2}L)$	$O(n^2)$	$O(nL)$
MicroSecAgg$_{gDL}$	4 | 2	$O(n \log^2 n)$	$O(nL + 2^{\ell/2}L)$	$O(\log^2 n)$	$O(L \log n)$
MicroSecAgg$_{CL}$	4 | 2	$O(n \log^2 n)$	$O(nL)$	$O(\log^2 n)$	$O(L \log n)$

Table 4: Total asymptotic computation cost for all rounds per aggregation with semi-honest security. n denotes the total number of users, L denotes the length of the input vector, and ℓ denotes the bit length of each element in the input vector. The "rd" column indicates the number of rounds in the setup phase (on the left, if applicable) and in each aggregation phase (on the right). We assume the number of offline users is $O(n)$ in every aggregation iteration in this table. "forwarding" means that the server only forwards the messages from the users.

	Communication cost			
Protocol	Server		User	
	Setup	Agg.	Setup	Agg.
BIK+17	–	$O(nL\ell + n^2\kappa)$	–	$O(L\ell + n\kappa)$
BBG+20	–	$O(nL\ell + n\kappa \log n)$	–	$O(L\ell + \kappa \log n)$
Flamingo	$O(\kappa \log n)$	$O(nL\ell + n\kappa \log^2 n)$	$O(\kappa \log n)$	$O(L\ell + n\kappa \log n)$
MicroSecAgg$_{DL}$	$O(n^2\kappa)$	$O(n\kappa L + n^2)$	$O(n\kappa)$	$O(\kappa L + n)$
MicroSecAgg$_{gDL}$	$O(n\kappa \log n)$	$O(n\kappa L + n \log n)$	$O(\kappa \log n)$	$O(\kappa L + \log n)$
MicroSecAgg$_{CL}$	$O(n\kappa \log n)$	$O(n\kappa L + n \log n)$	$O(\kappa \log n)$	$O(\kappa L + \log n)$

Table 5: Total received and sent asymptotic communication cost for all rounds per aggregation with semi-honest security. n denotes the total number of users, κ the security parameter, L the length of the input vector, and ℓ the bit length of each element.

cost. In the second round, each user also sends L element in $\mathbb{Z}_p^*$ to the server, which results in $O(kL)$ communication cost. Thus, the total communication cost of each user is $O(kL) + n$. As the size of messages between the server and each user is the same, the communication cost of the server is $O(nkL) + n^2$.

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 $\mathsf{ek}_{i,j}$ 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 $\mathsf{ek}_{i,j}$, 5) decrypt the cipher text $c_{j,i}$ for each other user j with $\mathsf{ek}_{i,j}$. Thus, the computation cost of each user in the Setup phase is $O(n^2)$. In the Aggregation phase, the computation cost of each user consists of calculating $H(k)^{x_i + r_i}$ and calculating $H(k)^{\sum_{j \in O} r_{j,i}}$ for each index of the input vector, which is $O(nL)$ 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 for each index of the input vector in the second round of the Aggregation phase. Thus, the computation cost of the server is $O(nL + 2^{\ell/2}L)$ in Aggregation phase of the semi-honest protocol, in which $2^{\ell/2}$ comes from discrete logarithm if the optimized algorithms, e.g., baby-step giant-step or Pollard's kangaroo algorithm are used.

E.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 $O(nk)$ 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, for each user, the asymptotic computation cost is the same as the cost of semi-honest protocol. For the server, the computation cost is $O(n)$ in the Setup phase and the same as the semi-honest protocol in the Aggregation phase.

E.2 Asymptotic Performance of MicroSecAgg$_{gDL}$

E.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 $O(k \log n)$. As messages between the server and each user is the same, the communication cost of the Setup phase for the server is $O(nk \log n)$.

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 phase is $O(kL + \log n)$ for each user and $O(nkL + n \log n)$ 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 $\mathsf{ek}_{i,j}$ for group members j of G_d and $\mathsf{mk}_{i,j}$ 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 $O(\log^2 n)$.

In each iteration of the Aggregation phase, each user i needs to compute the masked input $H(k)_i^X$ and the aggregated shares $H(k)^{\sum_{j \in O_d} r_{j,i} - \sum_{j \in G_d \setminus O_d} h_{j,i}}$ for every index of the input vector, which involves $O(L \log n)$ 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 δn offline users in which δ is a constant parameter, the server needs to do $O(\delta n \log^2 n)$ 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 for each index of the input vector. Assuming each group is of size $O(\log n)$, the computation cost of the server is $O(nL + 2^{\ell/2} L)$, in which $2^{\ell/2}$ comes from discrete logarithm if the optimized algorithms, e.g., baby-step giant-step or Pollard's kangaroo algorithm are used.

E.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 O_i by sending and receiving signatures on the set. These introduces $O(k \log n)$ 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.

E.3 Asymptotic Performance of MicroSecAgg$_{CL}$

As MicroSecAgg$_{CL}$ follows the same route as MicroSecAgg$_{gDL}$, the asymptotic performance of the two protocols is the same, except that the server-side computation cost of the Aggregation phase of MicroSecAgg$_{CL}$ does not have $2^{\ell/2}L$ which is from the inefficient discrete logarithm calculation in the other two protocols.

E.4 Additional Experimental Results

Different group/neighborhood sizes. Figure 9 shows the local computation time of each party of each iteration of MicroSecAgg$_{gDL}$ 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 [10] 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 over MicroSecAgg$_{DL}$ is not significant in MicroSecAgg$_{gDL}$ as the only 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.

Performance with Offline users. In Figure 10 we show the local computation time of online users (who stay online in the whole iteration) and the server in one iteration of MicroSecAgg$_{DL}$, MicroSecAgg$_{gDL}$, and BIK+17 with a different fraction of users dropping out. In each iteration, a δ-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 MicroSecAgg$_{DL}$, MicroSecAgg$_{gDL}$, and in the second round of BIK+17) and then stay silent for the rest rounds of the iteration. The size of the sum of inputs is fixed to 20 bits. 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 BIK+17 decreases as more users drop out. This is different from the experiment result reported in [11], as in [11] 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. 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 the impact of the dropout rate less severe. Moreover, we implement the reconstruction of Shamir's secret sharing naively using 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 the zoom-in graphs of Figure 10, we can see a higher value of δ offline parties 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.

Our protocols tolerate the same offline-rate as [10, 11]. The offline set can be different in each iteration without the need to reshare the masks and without efficiency loss as long as the number of offline users in every iteration is within the offline threshold. For practical applications, about 20% is a reasonable dropout rate. If too many users go offline, the protocol can halt and wait for enough users to come back, or start again with a smaller user set. As in [10] the offline parties are chosen at random, as shown in Lemma 4.4, with negligible probability a full group in MicroSecAgg$_{gDL}$ can go offline, in which case correctness is lost but privacy is maintained.

Setup phase performance. In Figure 11, we report the total computation time and bandwidth cost of each user and the server in the Setup phase of MicroSecAgg$_{DL}$ and MicroSecAgg$_{gDL}$ 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 MicroSecAgg$_{DL}$, 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 MicroSecAgg$_{gDL}$, 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.

Accuracy analysis with machine learning models. We also implement a federated learning protocol with MicroSecAgg$_{DL}$ on the adult census income dataset [27] 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. [16] which is a cleaned version with one constant intercept feature added based on a preprocessed version of the dataset from Jayaraman et al. [29] 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 MicroSecAgg$_{DL}$ 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) [39], 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.

F 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 [34]. However, it does not allow offline nodes to rejoin the training process later without sacrificing privacy. Several recent works also employ the idea of

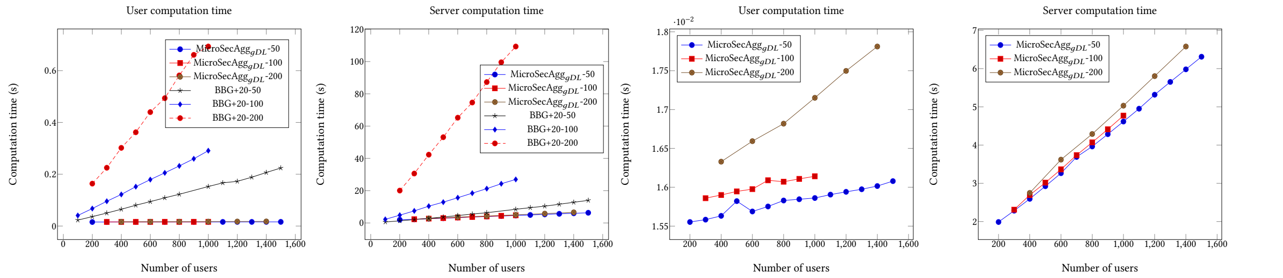

Figure 9: The left two graphs show the Wall-clock local computation time of one iteration of MicroSecAgg$_{gDL}$ and BBG+20 as the number of users increases. Different lines shows the running result with different group sizes. The length of the output is fixed to 20 bits. The right two graphs are zoom-in which only includes MicroSecAgg$_{gDL}$.

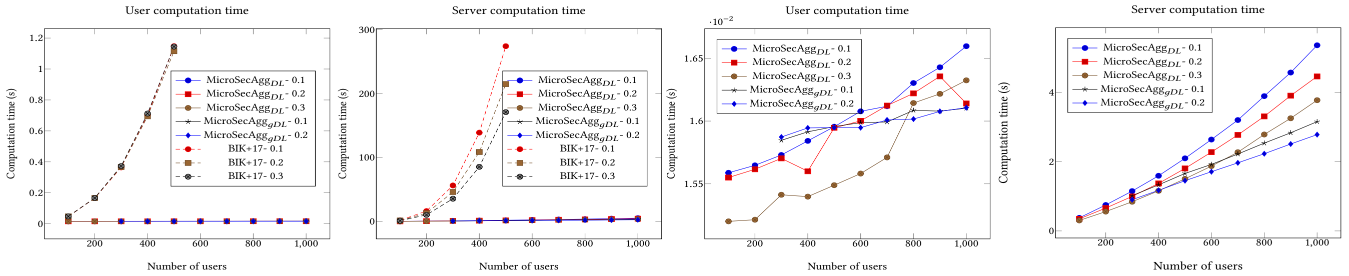

Figure 10: The left two graphs show wall-clock computation time of one iteration of the aggregation phase as the number of users increases. Different lines show the running time with different offline rates which is indicated in the legend, e.g., 0.1 means 10% of users are offline. The length of the output is fixed to 16 bits. The right two graphs are zoom-in with only MicroSecAgg$_{DL}$ and MicroSecAgg$_{gDL}$ included.

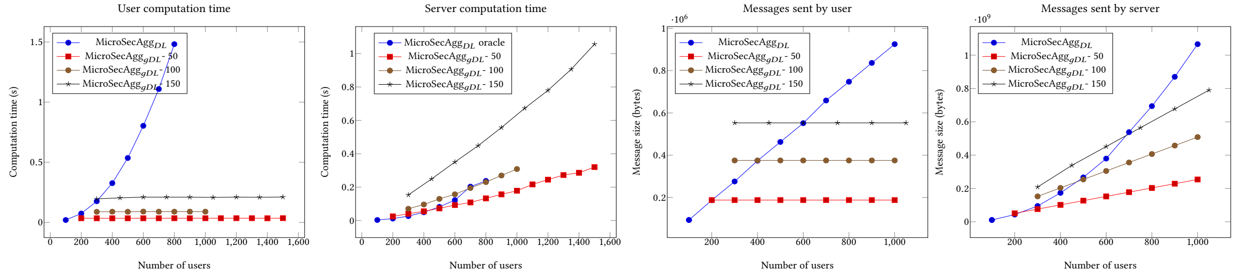

Figure 11: Wall-clock local computation time and outbound bandwidth cost (bytes) of the Setup phase as the number of users increases. Different lines shows the running results for MicroSecAgg$_{DL}$ and MicroSecAgg$_{gDL}$ with different group size.

reconstructing one layer of mask of online users. Yang et al. proposes a secure aggregation protocol LightSecAgg [48] 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 [28] proposed by Fereidooni et at., 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 calculates 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 [44] 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 users are online. HybridAlpha proposed by Xu et al. in [47] also adopts both differential privacy and functional encryption. It assumes honest but curious server and dishonest users.

G ABIDES FRAMEWORK

We use a discrete event simulation framework ABIDES [15] to implement the simulation of our protocol. In this section, we give an overview of this framework.

G.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.

G.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.

G.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.

G.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 are 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']`.

G.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.

G.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.