SECURE DISTRIBUTED COMPUTATION IN THE PRESENCE OF DYNAMIC PARTICIPATION

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Yue Guo
August 2022

SECURE DISTRIBUTED COMPUTATION IN THE PRESENCE OF DYNAMIC PARTICIPATION

Yue Guo, Ph.D.
Cornell University 2022

Distributed computing applications are booming in the Internet era as they enable mutually distrusting parties to collaborate and achieve a common goal. One of the major challenges in designing distributed computation protocols is to make progress and provide security guarantees while participants might not be always online. This work explores the solutions of secure distributed computation in the presence of dynamic participation and includes two main results. First, we propose a new timing model, weak synchrony, which combines the advantages of classical synchronous and partially synchronous model. Second, we provide a new secure aggregation protocol, MicroFedML, which improves on the performance of the state-of-the-art work [9, 11] while providing the same level of security guarantee and also supporting rejoining of temporarily offline participants.

The synchronous timing model is broadly applied in both theoretical designs and practical applications due to its robustness properties. In the synchronous model, it is assumed that messages between honest participants are delivered within a bounded delay Δ which is a parameter known to the protocol. This assumption allows protocols to be resilient against less than $1/2$ malicious participants. However, it also has a drawback that if an honest node ever experiences a short outage during which it is not able to receive honest messages within delay Δ, this node is considered to be corrupted and the protocol does not provide any consistency or liveness guarantee for it from that point

on. As an alternative, the partially synchronous model assumes the existence but not the knowledge of an upper bound of message delivery time between honest nodes, allowing the temporary offline period to be regarded as a long message delay. Unfortunately, protocols in the partially synchronous model cannot achieve the same level of malicious resiliency as in the synchronous model, as the well-known result shows that the resiliency upper bound of the partially synchronous model is only $1/3$.

In our work, we propose a new timing model, χ-weak-synchrony, which quantifies the network status more accurately and combines the advantages of both of the traditional timing models. It assumes a known upper bound Δ on message delivery time between honest and online participants. As long as there are more than χ-fraction of participants being honest and online in each round, a protocol in this model guarantees both liveness and consensus for all honest participants. We give constructions of a consensus protocol and an MPC protocol in the $1/2$-weakly synchronous model and a lower-bound result showing that the threshold $1/2$ is optimal.

Next, from a practical perspective, we discuss supporting dynamic participation in secure aggregation. Secure aggregation is a technique that allows a set of users to compute the aggregation of their private data with the aid of an entrusted server. It provides strong privacy guarantees and has been well-studied in the context of privacy-preserving federated learning. An important problem in privacy-preserving federated learning with constrained computation and wireless network resources is the computation and communication overhead which wastes bandwidth, increases training time, and can even affect model accuracy if many users drop out due to high cost. The seminal work of Bonawitz et al. [11] and the work of Bell et al. [9] have constructed secure aggregation protocols for a very large number of users which handle dropout users in a federated learning setting. However, these works suffer from high round complexity (defined as the number of times that users exchange messages with the server) and overhead in every training iteration.

In this work, we propose and implement MicroFedML, a new secure aggregation scheme with lower round complexity and computation overhead than existing works which achieve the same level of security. It uses a relaxation of the synchronous model which assumes a known upper bound on round time while allowing users to drop offline at any time point and come back online in later iterations. MicroFedML reduces the computational burden by at least 100 times for 500 users (or more depending on the number of users) and the message size by 50 times compared to prior work. Our system performs best when the input domain is not too large. Notable use cases include gradient sparsification, quantization, and weight regularization in federated learning.

BIOGRAPHICAL SKETCH

Yue Guo is a sixth-year Ph.D. student in Computer science department at Cornell University. Before coming to Cornell, she obtained her Bachelor's degree at Fudan University in 2016. Her research interests focus on applied cryptography and privacy. During her Ph.D. program, she did internships at VISA research, Thunder Research, and J.P. Morgan AI Research. She joined J.P. Morgan AI Research as a full-time employee in October, 2021.

To my family.

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to my advisor, Elaine Shi. She led me through each of our research projects and my whole PhD program, provided me with invaluable guidance and constant support in both of my research and personal life. Her passion for research always encouraged me to explore new problems and directions. I also want to thank my special committee members, Professor Rafael Pass and Professor Andrew Myers, for their valuable advice, guidance, and support.

I also would like to thank my coauthors during my PhD study, including Wei-kai Lin, Professor Hubert Chan, Professor Kai-Min Chung, Professor T-H. Hubert Chan, Antigoni Polychroniadou, David Byrd, Tucker Balch, Mustafa Safa ozdayi, Mahdi Zamani. I learned a lot from their novel questions and creative ideas, our fruitful discussions, and the experience of writing papers collaboratively.

I want to extend my sincere gratitude to Professor Yunlei Zhao, who was my research advisor when I was in undergraduate school. He inspired my interest in cryptography and encouraged me to pursue the Ph.D. degree.

I appreciate the help I got from the department, the administrative staff, and the international service office of Cornell. They were very responsive and supportive whenever I sought for help and always provided me with needful and timely information, advice, and assistance.

I am grateful for all the physical and mental support from my friends. A short list of them are: Yun Liu, Youer Pu, Tianze Shi, Yunhe Liu, Zikai Wen, Ted Yin, Xun Huang, Yawen Fang, Mengqi Xia, Xilun Chen, Mengjun Leng. They saw me through ups and downs in my PhD program and shared laughter and tears with me. I was very lucky to have their company in these years, especially during the pandemic.

Finally and most importantly, I want to thank my family. It was impossible for me to make it through without their encouragement and support.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vii
List of Tables . . . ix
List of Figures . . . x

1 Introduction 1
1.1 Weakly Synchronous Timing Model . . . 2
1.1.1 Definitional Contribution: A "Weak Synchronous" Network . . . 4
1.1.2 Results: Consensus in a Weakly Synchronous Network . . . 6
1.1.3 Results: MPC in a Weakly Synchronous Network . . . 10
1.2 MicroFedML: Secure Aggregation Protocol . . . 11
1.2.1 Problem Statement and Threat Model . . . 13
1.2.2 Our Results . . . 14

2 Preliminaries 17

3 Synchronous, with a Chance of Partition Tolerance 22
3.1 Technical Roadmap . . . 22
3.1.1 Reliable Broadcast (RBC) . . . 22
3.1.2 Verifiable Secret Sharing (VSS) . . . 24
3.1.3 Leader Election (LE) . . . 26
3.1.4 Byzantine Agreement (BA) . . . 28
3.1.5 Multi-Party Computation . . . 30
3.2 Defining a Weakly Synchronous Execution Model . . . 34
3.2.1 Modeling Corruption and Network Communication . . . 35
3.2.2 Modeling Setup Assumptions . . . 38
3.2.3 Weakly Synchronous Byzantine Agreement . . . 39
3.3 Lower Bounds . . . 41
3.3.1 Impossibility of Weakly-Synchronous Consensus for $\chi \leq 0.5$ . . . 41
3.3.2 Corrupt-Majority Protocols Sacrifice Partition Tolerance . . . 42
3.4 Reliable Broadcast (RBC) . . . 43
3.4.1 Definition . . . 44
3.4.2 Construction . . . 46
3.5 Verifiable Secret Sharing (VSS) . . . 47
3.5.1 Definitions . . . 47
3.5.2 A 0.5-Weakly-Synchronous VSS Scheme . . . 52
3.6 Leader Election (LE) . . . 55
3.6.1 Definition . . . 55
3.6.2 Construction . . . 57

3.7 Byzantine Agreement 60

4 MicroFedML: Secure Aggregation for Small Weights 64

4.1 Related Works 64
4.2 Preliminaries 66
4.2.1 Security definition 70
4.3 Secure aggregation with Random Oracle 72
4.3.1 The high-level construction 72
4.3.2 Protocol 74
4.3.3 Security 78
4.3.4 Correctness with dropouts 79
4.3.5 Privacy Against Semi-Honest Adversary 80
4.3.6 Privacy against malicious adversary 84
4.4 MicroFedML$_2$: Improvement with user grouping 93
4.4.1 Protocol with user grouping 96
4.4.2 Group properties 102
4.4.3 Correctness 104
4.4.4 Privacy 105
4.5 Performance Analysis 117
4.5.1 Asymptotic performance of MicroFedML$_1$ 118
4.5.2 Asymptotic performance of MicroFedML$_2$ 120
4.5.3 Implementation and concrete performance 122
4.5.4 Discussion about the size of the inputs 132
4.6 Conclusion and Future Work 135
4.7 ABIDES Framework 135
4.7.1 Creating Parties for a Protocol 135
4.7.2 Connecting Parties in a Protocol 140
4.7.3 Realistic Computation Delays within a Protocol 141

LIST OF TABLES

1.1 Communication overhead per user of each training iteration of our protocols guaranteeing privacy against semi-honest/malicious adversaries (the extra round required for privacy in the malicious setting is marked as red and underlined in the table). n denotes the total number of users and R denotes the size of the range of the aggregation output. An element in BIK+17 and BBG+20 is of size $O(\log R)$ while an element in MicroFedML$_1$ and MicroFedML$_2$ is of size $O(R)$. The overhead includes both received and sent messages. 15

4.1 Communication overhead of the Setup phase of aggregation of MicroFedML$_1$ and MicroFedML$_2$ guaranteeing privacy against malicious adversaries in which n denotes the total number of users. Note that BIK+17 and BBG+20 have $O(n^2)$ and $O(n \log n)$ communication cost, respectively, on the server side but they incur this cost in every iteration. 118

LIST OF FIGURES

4.1 An overview of the Setup phase of MicroFedML$_2$ 95

4.2 An overview of the Aggregation phase of MicroFedML$_2$ 96

4.3 Wall-clock local computation time of one iteration of the Aggregation phase of a user and the server as the number of user increases. The length of the sum of inputs is fixed to $\ell = 20$ bits in different lines, i.e., the input of each user is in the range $[2^\ell/n]$ when the total number of users is n. For the protocol MicroFedML$_2$ and BBG+20, the group size / neighbor size is set to 200. 124

4.4 Outbound bandwidth cost (bytes) on the user and the server side of each iteration of different protocols when the total number of users grows. The size of the result is fixed to 20 bits. The neighbor size in protocol MicroFedML$_2$ and BBG+20 is fixed to 100. 125

4.5 Zoom-in results of Figure 4.7 for the MicroFedML$_1$ protocol. The lines show wall-clock computation time of one iteration of the Aggregation phase of a user and the server, as the number of users increases. The length of the sum of inputs varies in different lines as shown in the legend. 125

4.6 Wall-clock local computation time of MicroFedML$_2$ for a user and the server with different group size (as shown in the legend), as the total number of users increases. The maximum size of the aggregation result is fixed to 20 bits. 126

4.7 Wall-clock local computation time of one iteration of the Aggregation phase of a user and the server as the number of user increases. The length of the sum of inputs is set to $\ell = 16, 20, 24$ bits in different lines, i.e., the input of each user is in the range $[2^\ell/n]$ when the total number of users is n. 127

4.8 Wall-clock computation time of one iteration of the Aggregation phase of a user and the server, as the number of users increases. Different lines show the running time when the fraction of offline users are different. The length of the sum of inputs is fixed to 16 bits. In other words, the input of each client is in the range $[2^{16}/n]$ when the total number of users is n. . . . 129

4.9 Zoom-in of Figure 4.8 with only MicroFedML$_1$ included. Wall-clock running time for a user and the server, as the number of users increases. Different lines show the running time when the fraction of offline users are different. 129

4.10 Wall-clock local computation time (y-axis) of one iteration of MicroFedML$_2$ and BBG+20 for a user (left) and the server (right) as the number of users (x-axis) increases. Different lines shows the running result with different group size (shown in the legend). The length of the sum of all inputs is fixed to 20 bits. 131

4.11 Wall-clock local computation time of the **Setup phase** for a user and the server, as the number of users increases. The length of the sum of inputs is fixed to 20 bits. Different lines shows the running results for MicroFedML$_1$ and MicroFedML$_2$ with different group size. 133

4.12 Outbound bandwidth cost (bytes) on the user and the server side of **the Setup phase** of MicroFedML$_1$ and MicroFedML$_2$ when the total number of users grows. The size of the result is fixed to 20 bits. Different lines shows the running results for MicroFedML$_1$ and MicroFedML$_2$ with different group size. 133

CHAPTER 1
INTRODUCTION

Distributed computation is persistently gaining popularity in the modern online world. It allows multiple participants to collaboratively perform a computation task without trusting any third party or risking their data privacy. Regarding the participants' behavior with respect to time, one widely used assumption known as the synchronous model is that all the messages between honest participants are always delivered within a known bounded period of time. This model enables simple protocol design in reliable and predictable networks, and provides robustness that would otherwise be impossible. However, this assumption is criticized for being too restrictive in use cases involving a large number of participants or running for a long time. In the synchronous model, if some participants fail to send or receive messages within the time bound, they are considered corrupted which means the protocol does not provide guarantee of output, consensus, or even data privacy to them. However, in the tasks which are run by thousands of participants scattered all over the world or run continuously for several years, e.g., federated learning or blockchain, some benign participants experiencing unstable network connection or power outages is almost unavoidable. It is important to handle the honest but temporarily offline participants more gracefully in those cases.

This work explores the solutions of distributed computation in the presence of dynamic participation. We presents two main results in timing models and secure aggregation respectively. In the next two sections, we briefly introduce the background of the problems and our main contributions. More thorough presentation of the works is included in Chapter 3 and Chapter 4.

1.1 Weakly Synchronous Timing Model

The “synchronous” model is one of the most commonly studied models in the past 30 years of distributed computing and cryptography literature. In the synchronous model, it is assumed that whenever an honest node sends a message, an honest recipient is guaranteed to have received it within a bounded delay Δ, and the protocol is aware of the maximum delay Δ.

The synchronous model allows protocols to achieve robustness properties that would otherwise be impossible. For example, assuming synchrony, we can achieve distributed consensus even when arbitrarily many nodes may be malicious [20]. In comparison, it is well known that if message delays can be arbitrarily long [22], consensus is impossible in the presence of $\frac{1}{3}$ fraction of corrupt nodes. On the other hand, the synchrony assumption has been criticized for being too strong [7, 43]: if an honest node ever experiences even a short outage (e.g., due to network jitter) during which it is not able to receive honest messages within Δ delay, this node is now considered as corrupt. From this point on, a consensus protocol proven secure under a synchronous model is not obliged to provide consistency and liveness to that node any more, even if the node may wake up shortly afterwards and wish to continue participating in the protocol. Similarly, as soon as P has even a short-term outage, a multi-party computation (MPC) protocol proven secure under a synchronous model is not obliged to provide privacy for party P’s inputs — for example, some protocols that aim to achieve fairness and guaranteed output would now have the remaining online parties reconstruct P’s secret-shared input and thus P loses its privacy entirely.

We stress that this is not just a theoretical concern. Our work is in fact directly motivated by conversations with real-world blockchain engineers who were building and deploying a fast cryptocurrency and pointed out what seems to be a fatal flaw in a blockchain protocol [44] that was proven secure in the classical synchronous model: even when all nodes are benign and only a few crash in a specific timing pattern, transactions that were "confirmed" can be "undone" from the perspective of an honest node who just experienced short-term jitter possibly unknowingly (see the online full version [28] for a detailed description of this real-world example).

Not only so, in fact to the best of our knowledge, all known classical-style, *synchronous* consensus protocols [2, 32, 40] are *underspecified* and *unimplementable* in practice: if a node ever experiences even a short-term outage and receives messages out of sync, these protocols [2, 32, 40] provide no explicit instructions for such nodes to join back and continue to enjoy consistency and liveness!

Of course, one known solution to this problem is to simply adopt a partially synchronous [22] or asynchronous [15] model. In a partially synchronous or asynchronous model, a short-term outage would be treated in the same way as a long network delay, and a node that is transiently offline will not be penalized. For this reason, partially synchronous (or asynchronous) protocols are known to be arbitrarily *partition tolerant*; while synchronous protocols are *not*. Unfortunately, as mentioned, partially synchronous or asynchronous protocols can tolerate only $1/3$ fraction of corruptions!

Can we achieve the best of both worlds, i.e., design distributed computing protocols that resist more than $1/3$ corruption and meanwhile achieve a practical notion of partition tolerance?

1.1.1 Definitional Contribution: A "Weak Synchronous" Network

At a very high level, we show that synchrony and partition tolerance are not binary attributes, and that we can guarantee a notion called "best-possible partition tolerance" under a quantifiable measure of synchrony. To this end, we propose a new model called a χ-*weakly-synchronous* network.

A natural but overly restrictive notion. One natural way to quantify the degree of synchrony is to count the fraction of nodes that always respect the synchrony assumption. For example, we may want a distributed computing protocol to satisfy desired security properties (e.g., consistency, liveness, privacy), as long as more than χ fraction of nodes are not only honest but also always have good connectivity (i.e., bounded Δ delay) among themselves. This model, however, is overly restrictive especially in long-running distributed computing tasks such as a blockchain: after all, no node can guarantee 100% up-time [1], and over a few years duration, it could be that every node was at some point, offline.

χ**-weak-synchrony.** We thus consider a more general model that allows us to capture network churn. We now require only the following:

> [χ-weakly-synchronous assumption:] In every round, more than χ fraction nodes are not only honest but also *online*; however, the set of honest and online nodes in adjacent rounds need not be the same.

Throughout the paper we use the notation $\mathcal{O}_r$ to denote a set of at least $\lfloor \chi n \rfloor + 1$ honest nodes who are online in round r — henceforth $\mathcal{O}_r$ is also called the "honest and online set of round r". Note that the remaining set $[n]\backslash\mathcal{O}_r$ may contain a combination of honest or corrupt nodes and an honest node in $[n]\backslash\mathcal{O}_r$ is said to be offline in round r.

We assume that the network delivery respects the following property where multicast means "send to everyone":

[network assumption:] when a node in $\mathcal{O}_r$ multicasts a message in round r, every node in $\mathcal{O}_t$ where $t \geq r + \Delta$ must have received the message in round t.

We allow the adversary to choose the honest and online set of each round (even after observing the messages that honest nodes want to send in the present round), and delay or erase honest messages, as long as the above χ-weak-synchrony and network delivery constraints are respected. For example, the adversary may choose to delay an honest but offline node's messages (even to online nodes) for as long as the node remains offline. The adversary can also selectively reveal an arbitrary subset of honest messages to an honest and offline node.

Therefore, our weak synchrony notion can be viewed as a generalization of the classical synchronous notion (henceforth also called strong synchrony). In a strongly synchronous network, it is required that the honest and online set of every round must contain all honest nodes.

We ask whether we can achieve secure distributed computing tasks under such a χ-weakly-synchronous network. With the exception of liveness (or guaranteed output) which we shall discuss shortly, we would like to guarantee all security properties, including consistency and privacy, for *all* honest nodes, regardless of whether or when they are online/offline. Defining liveness (or guaranteed output) in the χ-weakly-synchronous model, however, is more subtle. Clearly we cannot hope to guarantee liveness for an honest but offline node for as long as it remains offline. Therefore, we aim to achieve a "best-effort" notion of liveness: a protocol has T-liveness iff for any honest node that becomes online in some round $r \geq T$, it must have produced output by the end of round r.

The challenges. We are faced with a few apparent challenges when designing distributed protocols secure under χ-weak-synchrony. First, the online nodes may change rapidly in adjacent rounds. For example, if $\chi = 0.5$ and everyone is honest, the honest and online sets belonging to adjacent rounds can be almost disjoint. Second, we stress that *offline nodes may not be aware they are offline*, e.g., a DoS attack against a victim's egress router clearly will not announce itself in advance. Further, the adversary can selectively reveal a subset of messages to offline nodes such that they cannot detect they are offline from the protocol messages they receive either. Because of these facts, designing protocols in our χ-weakly-synchronous model is significantly more challenging than the classical synchronous model (or even the above restrictive model where we may assume a sufficiently large set of honest and persistently online nodes).

1.1.2 Results: Consensus in a Weakly Synchronous Network

We consider the feasibility and infeasibility of achieving Byzantine Agreement (BA) in a χ-weakly-synchronous network. In a BA protocol, a designated sender has an input bit that it wants to convey to all other nodes. We would like to achieve the following guarantess for all but a negligible fraction of executions: 1) *consistency*, i.e., all honest nodes must output the same bit; 2) *validity*, i.e., if the designated sender is honest and online in the starting round (i.e., round 0) of the protocol, every honest node's output (if any) must agree with the sender's input bit; and 3) T*-liveness*, i.e., every node in $\mathcal{O}_r$ where $r \geq T$ must have produced an output by the end of round r. Note that if the designated sender is honest but offline initially, the protocol cannot make up for the time lost when the sender is offline — thus validity requires that the sender not only be honest but also online in the starting round.

As mentioned, we are primarily interested in protocols that tolerate more than $1/3$ corruptions since otherwise one could adopt a partially synchronous or asynchronous model and achieve arbitrary partition tolerance. To avoid a well-known lower bound by Lamport et al. [35], throughout the paper we assume the existence of a public-key infrastructure (PKI).

Impossibility when minority are honest and online. Unfortunately, we show that it is impossible to have a χ-weakly-synchronous consensus protocol for $\chi < 0.5 - 1/n$, i.e., if the honest and online set of each round contains only minority number of nodes (and this lower bound holds even assuming any reasonable setup assumption such as PKI, random oracle, common reference string (CRS), or the ability of honest nodes to erase secrets from memory). The intuition for the lower bound is simple: there can be two honest well-connected components that are partitioned from each other, i.e., the minority honest nodes inside each component can deliver messages to each other within a single round; however messages in between incur very long delay. In this case, by liveness of the consensus protocol, each honest well-connected component will reach agreement independently of each other. We formalize this intuition later in Section 3.3.

Best-possible partition tolerance. Due to the above impossibility, a consensus protocol that achieves consistency, validity, and liveness under 0.5-weak-synchrony is said to be *best-possible partition tolerant*.

A refinement of synchronous consensus. First, it is not hard to see that any best-possible partition tolerant Byzantine Agreement (BA) protocol (i.e., secure under 0.5-weak-synchrony) must also be secure under honest majority in the classical, strong synchronous model. On the other hand, the converse is not true. Interestingly, we examined several classical, honest-majority BA protocols [2, 32, 40, 44] and found none of them to satisfy best-possible partition tolerance. In this sense, our notion of best-possible partition tolerance can also be viewed as a refinement of classical honest-majority BA, i.e., we can tease out a proper subset of honest-majority BA protocols that satisfy good-enough partition tolerance in practice — and we strongly recommend this robust subset for practical applications.

Round-efficient, best-possible partition tolerant BA. Of course, to show that our notion is useful, we must show existence of a best-possible partition tolerant BA that is efficient; and this turns out to be non-trivial.

Theorem 1 (Informal)**.** *Assume the existence of a PKI and enhanced trapdoor permutations. Then, there exists an expected constant-round* BA *protocol secure under* 0.5*-weak-synchrony.*

Note that here, expected constant-round means that there is a random variable T whose expectation is constant, such that if an honest node becomes online in round $r \geq T$, it must have produced an output in round r.

We additionally show how to extend the above result and construct a best-possible partition tolerant BA protocol that is optimistically responsive [44]: specifically, under the following optimistic conditions, the honest and online nodes in $\mathcal{O}$ will produce an output in $O(\delta)$ amount of time where δ is the actual maximum network delay (rather than the a-priori upper bound Δ):

> $\mathbf{O}$:= "there exists a set $\mathcal{O}$ containing at least $3n/4$ honest and persistently online nodes, and moreover, the designated sender is not only honest but also online in the starting round"

Corollary 1.1.1 (Informal)**.** *Assume the existence of a PKI and enhanced trapdoor permutations. Then, there exists an expected constant-round* BA *protocol secure under χ-weak-synchrony; moreover, if the optimistic conditions* $\mathbf{O}$ *specified above also holds, then the honest and online nodes in $\mathcal{O}$ would produce output in $O(\delta)$ time where δ is the actual maximum network delay.*

Classical, corrupt-majority BA **protocols inherently sacrifice partition tolerance.** As is well-known, in the classical, strongly synchronous model, we can achieve BA even when arbitrarily many nodes can be corrupt. We show, however, the set of corrupt-majority protocols are disjoint from the set of best-possible partition tolerant protocols. Not only so, we can show that the more corruptions one hopes to tolerate, the less partition tolerant the protocol becomes. Intuitively, the lower bound is simple because in a corrupt majority protocol, a minority honest well-connected component must independently reach agreement among themselves in a bounded amount of time; and obviously there can be two such components that are disconnected from each other and thus consistency among the two components is violated (with constant probability).

This simple observation, however, raises another philosophical point: if we adopted the classical synchronous model, it would be tempting to draw the conclusion that corrupt-majority BA is strictly more robust than honest-majority BA. However, we show that one must fundamentally sacrifice partition tolerance to trade for the ability to resist majority corruption and this tradeoff is, unfortunately, inherent.

1.1.3 Results: MPC in a Weakly Synchronous Network

We next consider the feasibility of realizing multi-party computation in a χ-weakly-synchronous network. Imagine that n parties would like to jointly evaluate the function $f(x_1, \ldots, x_n)$ over their respectively inputs $x_1, x_2, \ldots, x_n$ such that only the outcome is revealed and nothing else. Again, a couple of subtleties arise in formulating the definition. First, one cannot hope to incorporate the inputs of offline nodes if one would like online nodes to obtain outputs quickly. Thus, we require that at least $\lfloor \chi n \rfloor + 1$ number of honest nodes' inputs be included and moreover, every honest node who has always been online throughout the protocol should get their inputs incorporated. Concretely, we require that the ideal-world adversary submit a subset $\mathfrak{I} \subseteq [n]$ to the ideal functionality, such that $\mathfrak{I} \cap \mathsf{Honest} \geq \lfloor \chi n \rfloor + 1$ where Honest denotes the set of honest nodes, and moreover $\mathfrak{I}$ must include every honest node who has been online throughout the protocol. Henceforth, the subset $\mathfrak{I}$ is referred to as the "effective input set":

- for every $i \in \mathfrak{I}$ that is honest, the computation should use node i's true inputs;
- for every $i \in \mathfrak{I}$ that is corrupt, we allow the ideal-world adversary to replace the input to any value of its choice; and
- for every $i \notin \mathfrak{I}$, the computation simply uses a canonical input $\perp$ as its input.

Second, the notion of guaranteed output must be treated in the same manner as liveness for BA since we cannot hope that honest but offline nodes can obtain output for as long as they remain offline. We say that an execution of the multi-party protocol completes in T rounds, iff for any honest node in $\mathcal{O}_t$ where $t \geq T$, it must have produced an output by the end of round t.

Under the above definition, we prove the following theorem (informally stated):

Theorem 2 (Informal)**.** *Assume the existence of a PKI, enhanced trapdoor permutations, and that the Learning with Errors (LWE) assumption holds. Then, there is an expected constant-round protocol that allows multiple parties to securely evaluate any function f under* 0.5*-weak-synchrony.*

We further extend our results in a non-trivial manner and and achieve optimistically responsive MPC in the online full version [28].

Additional related work. We provide comparison with additional related work in our online full version [28].

1.2 MicroFedML: Secure Aggregation Protocol

Federated learning allows a large number of users with limited resources, e.g., mobile phones, to collaboratively train a global learning model with the assistance of a central server without sharing the raw data with any other party. In each learning iteration of federated learning, a central server sends the global model to all users who then train the model with their local data to obtain an updated model. The server aggregates the updates from all the users and updates the global model. The new global model is sent to all users, and the process repeats. In many cases, individual user privacy can still be compromised by using the trained model and the local updates to infer certain details of the training data set [41] . To address this problem, the seminal work of Bonawitz et al. proposed the first secure aggregation protocol run among the server and the users without revealing any individual's update. The protocol allow each user to modify its local updates by masking/encrypting it such that no information about the local updates is revealed to the server and the other users apart from the final aggregated update.

In the federated learning setting, we require secure aggregation protocols whose efficiency and communication requirements scale practically even when the number of parties is large. Bonawitz et al. [11] proposed the first secure aggregation protocol which is efficient for deep-network-sized problems and real-world connectivity constraints. The protocol also allows users to drop offline and come back online later, as in many scenarios the users are not expected to be always online in the whole training process. The subsequent work of Bell et al. [9] reduces the bandwidth cost at the price of higher round complexity and probabilistic correctness, i.e., when there are enough number of honest users online, the server learns the correct aggregated result with overwhelming probability. The work of Bonawitz et al. [11] provides a protocol with perfect correctness. Instead of requiring each user to exchange information with all other users in the network, Bell et al. [9] works with sparse neighborhood graphs. The intuition is that each user has a small group of users as its neighbors and the communication only happens between the user and its neighbors.

The works of [9, 11] suffer from some inefficiencies. More specifically, in every learning iteration the protocols in [9, 11] reveal part of the masks used to protect the local updates from the server. That said, the masks cannot be reused in later training iterations and fresh masks are generated at each iteration introducing more communication. Furthermore, every user needs to exchange information about the freshly generated masks with a number of other users (either all other users in [11] or a subset of users in [9]) 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 [9]. A lower round complexity can prevent the multiple exchange of messages between the server and the many users and can increase the accuracy given that users have less chances to drop out in every round. Therefore, a natural yet fundamental question to ask is the following: can we have a secure aggregation protocol which is more tailored to the multi-iteration nature of the federated learning setting with improvements in the round complexity, computational complexity and bandwidth costs?

In this work, we answer the above question in the affirmative by designing secure aggregation protocols MicroFedML which considerably reduce the communication and computation overheads of [9, 11]. In a nutshell, our protocols generate the masks in a one-time setup phase which is executed only once and most importantly the masks can be reused in every training iteration while achieving the same provable security guarantees provided by [9, 11]. Notable improvements of MicroFedML over [9, 11] include the following. MicroFedML reduces the round complexity from 5 in [11] and 6 in [9] to just 3 rounds. The computation complexity of each iteration for each user increases linearly with the number of users in MicroFedML while that of [9, 11] both grow quadratically. We present our contributions with more details in the next section.

1.2.1 Problem Statement and Threat Model

We consider a star network topology network in which each user only communicates with the central server and the messages between the online users and the server are delivered within bounded time Δ. We propose multi-iteration secure aggregation protocols in which n users P_i for $i \in [n]$ holds a private value x_i, and they wish to learn the sum $\sum_i x_i$ with the aid of a single untrusted server without leaking any information about the individual x_i. In the federated learning setting, the server and the users interact several times (multi-iteration) to compute the summation of model weights x_i. We model an adversary which can launch two kinds of attacks: (1) honest users that disconnect/drop out or are too slow to respond as a result of unstable network conditions, power loss, etc. User 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 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 dropping out in every iteration. We assume that $\gamma + 2\delta < 1$. An adversary in a semi-honest protocol corrupts parties but follows the protocol's specification and tries to learn information from the received messages. An adversary in a malicious protocol is allowed to deviate from the protocol's specification in arbitrary ways, changing the messages in an effort to learn the private information of the honest parties.

1.2.2 Our Results

We propose two new multi-iteration secure aggregation protocols MicroFedML$_1$ and MicroFedML$_2$ both in the semi-honest and malicious adversary settings. Both protocols consist of two phases, the Setup phase and the Aggregation phase. The Setup phase which is independent of the user private inputs consists of 3 rounds of interaction between the server and the users in MicroFedML$_1$ or 5 rounds in MicroFedML$_2$. The setup phase runs only once at the beginning of the execution of federated learning. The aggregation phase runs repetitively for multiple learning iterations and consists of 2 rounds of interaction in the semi-honest scenario in both protocols, while an extra round is needed to protect privacy against a malicious adversary.

In Table 1.1 we list the communication complexity per user per round. As we can see

Round	Communication cost per user			
	BIK+17	BBG+20	MicroFedML$_1$	MicroFedML$_2$
1	$O(n)$ elements	$O(1)$ elements	1 element	1 element
2	$O(n)$ elements	$O(\log n)$ elements	1 element + n bits	1 element + $\log n$ bits
3	1 elements	$O(\log n)$ elements	$O(n)$ elements	$O(\log n)$ elements
4	$O(n)$ elements	1 element		
5	$O(n)$ elements	$O(\log n)$ elements		
6		$O(\log n)$ elements		

Table 1.1: Communication overhead per user of each training iteration of our protocols guaranteeing privacy against semi-honest/malicious adversaries (the extra round required for privacy in the malicious setting is marked as red and underlined in the table). n denotes the total number of users and R denotes the size of the range of the aggregation output. An element in BIK+17 and BBG+20 is of size $O(\log R)$ while an element in MicroFedML$_1$ and MicroFedML$_2$ is of size $O(R)$. The overhead includes both received and sent messages.

our protocols offer a significant advantage per round. The one-time setup phase communication complexity is $O(n)$ and $O(\log n)$ for each user in MicroFedML$_1$ and MicroFedML$_2$ respectively, which can be found in Table 4.1 in Section 4.5. We refer the reader to Section 4.5.1 and 4.5.2 for the detailed analysis of the asymptotic performance. The group version of our protocol is more suitable for the use cases where weaker security guarantees are sufficient. Such as, the adversary cannot adaptively corrupt all parties in a single group neighborhood. If such an event happens privacy is lost.

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

Theorem 3. *The aggregation protocol MicroFedML$_1$ 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.*

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 $k \in [K]$. Our protocols do not introduce any noise and thus do not drop the accuracy of the models. We test our protocol by running logarithmic regression algorithm on the Census adult dataset [21] 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).

Theorem 4 (Group version)**.** *Let γ, δ be two parameters such that $\gamma + 2\delta < 1$. The aggregation protocol MicroFedML$_2$ 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.*

We also provide the asymptotic and concrete performance analysis in Section 4.5. Our algorithms work best with small input domains, which is applicable in gradient sparsification, quantization and weight regularization areas in federated learning. See Section 4.1 for more information about these areas.

Implementation and evaluation. The system we report here is implemented, and we report running times in Section 4.5. Our protocol MicroFedML$_1$ outperforms BIK+17 by 100 times in computation time with 500 total participants, while MicroFedML$_2$ runs more than 5 times faster than BBG+20 when the connectivity of the user communication graph is 100 and the total number of clients is 500. For 1000 participants, MicroFedML$_2$ is 5 times faster than MicroFedML$_1$.

CHAPTER 2

PRELIMINARIES

In this chapter, we introduce the notations and several cryptographic primitives that will be used in this dissertation.

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

Cryptographic Primitives 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 all but negligible probability.

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

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

Shamir's Secret Sharing We use Shamir's t-out-of-n secret sharing in [48] to tolerate offline users. Informally speaking, it allows the secret holder dividing the secret into n shares such that anyone who knows t of them can reconstruct the secret, while anyone who knows less than t shares cannot learn anything about the secret.

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:

- $\mathsf{SS.share}(s, \{x_1, x_2, \ldots, x_n\}, t) \to \{(s_1, x_1), \ldots, (s_n, x_n)\}$, 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.
- $\mathsf{SS.recon}(\{(s_1, x_1), \ldots, (s_n, x_n)\}, t) = s$, in which each pair (s_i, x_i) denotes the share s_i on index x_i. This function returns the original secret s.

The first function can be implemented by uniformly randomly choosing $t-1$ coefficients $a_1, \ldots, a_{t-1}$ from $\mathbb{Z}_q$, and calculates $s_i = f(x_i)$ for $f(x) = s + a_1 x + \ldots + a_{t-1}x^{t-1}$. The function f can be reconstructed from the shares with the Lagrange basis polynomials. More specifically, let $\ell_i(x) = \Pi_{j \neq i, j \in [n]} \frac{x - x_j}{x_i - x_j}$, then $f(x) = \sum_{i \in [n]} s_i \cdot \ell_i(x)$. In this way, we can obtain $s = f(0)$.

Additionally, we define the following two functions as an extension of Shamir's secret sharing. Let p, q be primes such that $p = 2q + 1$. Let $g \in \mathbb{Z}_p$ be a generator of $\mathbb{Z}_p$, and 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:

- $\mathsf{SS.exponentRecon}((g^{s_1}, x_1), \ldots, (g^{s_n}, x_n), t) = \{g^s, g^{a_1}, \ldots, g^{a_{t-1}}\}$: With the shares $g^{s_1}, \ldots, g^{s_t}$, it returns the secret and the polynomial coefficients of the Shamir's 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 $i \in \{i_1, \ldots, i_t\}$.

  This function can be implemented without knowing $s_{i_1}, \ldots, s_{i_n}$ by performing all the linear operations in the exponent.
- $\mathsf{SS.exponentShare}(x, g^s, g^{a_1}, \ldots, g^{a_{t-1}}) = g^{s_x}$: with the coefficients of the polynomial in exponent, it returns a new share for player i. 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 value of $s, a_1, \ldots, a_{t-1}$.

Decisional Diffie-Hellman (DDH) Assumption

Definition 2.0.1 (Decisional Diffie-Hellman (DDH) Assumption). *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 $\mathbb{Z}_q$:*

$$g^a, g^b, g^{ab} \textit{ and } g^a, g^b, g^c.$$

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 discrete log problem is computationally hard. It consists of three algorithms,

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

Random Oracle We assume the existence of random oracle which answers each unique query with a uniformly random response in its output domain. We use the random oracle to guarantee that all users and the server can access the same fresh randomness for each iteration.

Pseudorandom Generator Pseudorandom generator (PRG) is an algorithm which extends a short uniformly random seed to a longer sequence of bits which is computationally indistinguishable from a truly random sequence of the same length as long as the random seed is hidden from the distinguisher. We use the algorithm $\mathsf{PRG}(r) \to s$ which takes a uniformly random seed $r \in \{0,1\}^\lambda$ and generates a sequence $s \in \{0,1\}^T$. Here, T is the total number of iterations the protocol will run.

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 following tail bounds for $X \sim \mathsf{HyperGeom}(N, m, n)$:

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

Authenticated Encryption Authenticated encryption (AE) is a class of encryption algorithms that guarantees message integrity, confidentiality, and authenticity. We assume that the scheme we use in this dissertation satisfies IND-CCA2 (adaptive chosen ciphertext attack) security. In this kind of attacks, the adversary is given access to both the encryption and decryption oracles which encrypts and decrypts arbitrary plaintext or ciphertext at the adversary's request. It receives two messages $m_0, m1$, and a challenge ciphertext which is an encryption of message m_b for a uniformly random bit b, and needs to give a guess of b. The adversary tries to gain a non-negligible advantage in this game by querying the two oracles both before and after it receives the challenge ciphertext (without querying the decryption oracle with the challenge ciphertext). If an encryption scheme is secure against such an attack, it guarantees that an incorrectly generated ciphertext will be refused by the decryption algorithm.

Public Key Infrastructure In cryptography, a public key infrastructure (PKI) is an arrangement that binds public keys with respective identities of entities (like people and organizations). The binding is established through a process of registration and issuance of certificates at and by a certificate authority (CA). Depending on the assurance level of the binding, this may be carried out by an automated process or under human supervision. When done over a network, this requires using a secure certificate enrollment or certificate management protocol such as CMP.

PKI provides sender authentication for messages and also prevents the adversary from forging large number of identities in a protocol.

Indistinguishability We say a function $f : \mathbb{N} \to [0,1]$ is negligible if for every $c \in \mathbb{N}$, there exists some $n_0 \in \mathbb{N}$ such that for all $n > n_0, f(n) \leq 1/n^c$; symmetrically, we say a function $g : \mathbb{N} \to [0,1]$ is overwhelming if for every $c \in N$, there exists some $n_0 \in \mathbb{N}$ such that for all $n > n_0$, $g(n) \geq 1 - 1/n^c$. For random variables X and Y taking values in U, their statistical difference is

$$SD(X,Y) := \max_{T \in U} |\Pr[X \in T] - \Pr[Y \in T]|,$$

CHAPTER 3

SYNCHRONOUS, WITH A CHANCE OF PARTITION TOLERANCE

3.1 Technical Roadmap

The most technically non-trivial part of our result is how to realize Byzantine Agreement (BA) under 0.5-weak-synchrony. Existing synchronous, honest-majority protocols [32, 40] completely fail in our model. Since the honest and online set can change rapidly in every round, it could be that by the end of the protocol, very few or even no honest nodes have been persistently online, and everyone honest was offline at some point. In other words, it could be that from the view of every honest node, message delivery was asynchronous at some point in the protocol. Indeed, interestingly many of our core techniques are in fact reminiscent of asynchronous consensus rather than synchronous approaches.

Although at a very high level, we follow a well-known recipe that constructs BA from a series of building blocks:

$$\begin{aligned} & \text{Reliable Broadcast } (\mathsf{RBC}) \Rightarrow \text{Verifiable Secret Sharing } (\mathsf{VSS}) \\ \Rightarrow\ & \text{Leader Election } (\mathsf{LE}) \Rightarrow \text{Byzantine Agreement } (\mathsf{BA}) \end{aligned}$$

as it turns out, for all these building blocks, even how to define them was non-trivial: the definitional subtleties arise partly due to the new χ-weakly-synchronous model, and partly due to compositional issues.

3.1.1 Reliable Broadcast (RBC)

Definition. Reliable broadcast (RBC) allows a designated sender to convey a message to other nodes. The primitive can be viewed as a relaxed version of BA: assuming 0.5-

weak-synchrony, RBC always guarantees the following for all but a negligible fraction of executions:

1. *Consistency:* if two honest nodes output x and x' respectively, it must be that $x = x'$. For technical reasons that will become clear later, we actually need a strengthening of the standard consistency notion, requiring that an efficient extractor can extract the value that honest nodes can possibly output, given honest nodes' transcript in the initial T rounds of the protocol.

2. *Validity:* if the sender is honest, then honest nodes' output must be equal to the honest sender's input;

3. *T-liveness (under an honest and initially online sender):* if the sender is not only honest but also online in the starting round, then every node in $\mathcal{O}_t$ where $t \geq T$ must have produced an output by the end of round t;

4. *Close termination:* if any honest node (even if offline) produces and output in round r, then anyone in $\mathcal{O}_t$ where $t \geq r + 2\Delta$ must have produced an output by the end of round t too.

Interestingly, note that the T-liveness property is reminiscent of classical synchronous definitions whereas the close termination property is reminiscent of asynchronous definitions.

Construction. At a very high level, our RBC construction combines techniques from classical synchronous "gradecast" [23, 32] and asynchronous "reliable broadcast" [14, 15]. We defer the concrete construction to Section 3.4; the construction is constant round, i.e., achieves T-liveness where $T = O(1)$.

3.1.2 Verifiable Secret Sharing (VSS)

Definition. Verifiable secret sharing (VSS) allows a dealer to share a secret among all nodes and later reconstruct it. We propose a new notion of (a computationally secure) VSS that is composable and suitable for a 0.5-weakly-synchronous network. Somewhat imprecisely, we require the following properties:

- *Binding (formally referred to as Validity in Section 3.5.2).* Standard notions of VSS [15] require that the honest transcript of the sharing phase binds to the honestly reconstructed secret. For technical reasons needed later in the proof of the Leader Election (LE), we require a stronger notion: an efficient extractor $\mathcal{E}$, knowing honest nodes' public and secret keys, must be able to extract this secret from the honest transcript during the sharing phase, and the honestly reconstructed secret must agree with the extractor's output.
- *Secrecy and non-malleability.* If the dealer is honest, then the shared value must remain secret from the adversary before reconstruction starts. Not only so, we also need a non-malleablity: an adversary, after interacting in VSS instances each with an honest dealer, cannot act as a dealer in another VSS instance and share a secret that is related to the honest secrets.
- *Liveness.* For liveness, we require that if the dealer is honest and online in the initial round of the sharing phase, for $t \geq T$, everyone in $\mathcal{O}_t$ must have output "`sharing-succeeded`". Even when the dealer is corrupt or initially offline, if any honest node (even if offline) ever outputs "`sharing-succeeded`" in some round r, then everyone in $\mathcal{O}_t$ where $t \geq r + 2\Delta$ must have output "`sharing-succeeded`" by the end of round t. If some honest node has output "`sharing-succeeded`", then reconstruction must be successful and will terminate in T rounds for honest and

online nodes.

Just like the RBC definition, our VSS definition also has both synchronous and asynchronous characteristics.

Construction. Informally our construction works as follows:

- Share. In the starting round of the sharing phase, the dealer secret splits its input s into n shares denoted $s_1, s_2, \ldots, s_n$ using a $(\lfloor n/2 \rfloor + 1)$-out-of-$n$ secret-sharing scheme. It then encrypts the share s_j to node i's public key pk_j using a public-key encryption scheme — let CT_j be the resulting ciphertext. Now, the node proves in zero-knowledge, non-interactively, that the ciphertexts $\mathsf{CT}_1, \ldots, \mathsf{CT}_n$ are correct encryptions of an internally consistent sharing of some secret — let π denote the resulting proof. Assuming PKI and honest majority, we can realize a Non-interactive Zero-Knowledge Proof (NIZK) system (without CRS) using a technique called multi-string honest majority NIZK proposed by Groth and Ostrovsky [27] (see online full version [28]). Finally, the dealer invokes an RBC instance (henceforth denoted RBC_0) to reliably broadcast the tuple $(sid, \{\mathsf{CT}_j\}_{j \in [n]}, \pi)$ to everyone — here sid denotes the current instance's unique identifier and this term is needed here and also included in the NIZK statement for compositional reasons.

  Suppose that the RBC scheme employed satisfies T_{rbc}-liveness. Now in round T_{rbc} (assuming that the starting round is renamed to round 0), if a tuple has been output from the RBC_0 instance with a valid NIZK proof, then reliably broadcast the message "`ok`"; otherwise reliably broadcast the message $\perp$ — note that here n instances of RBC are spawned and each node i will act as the designated sender in the i-th instance. Finally, output "`sharing-succeeded`" iff not only RBC_0 has output a tuple with a valid NIZK proof but also at least $\lfloor n/2 \rfloor + 1$ RBC instances have output "ok" — note that at this moment, the node (denoted i) can decrypt its own share s'_i from the corresponding ciphertext component contained in the output of RBC_0.

- Reconstruct: If the sharing phase has output "sharing-succeeded" and moreover the reconstruction phase has been invoked, then node i multicasts the decrypted share s'_i as well as a NIZK proof that the decryption was done correctly (in a way that is consistent with its public key). Finally, as soon as $\lfloor n/2 \rfloor + 1$ decryption shares with valid NIZK proofs are received, one can reconstruct the secret.

3.1.3 Leader Election (LE)

Definition. Leader Election (LE) is an inputless protocol that allow nodes to elect a leader denoted $L \in [n]$ among the n nodes. For the outcome of LE to be considered "good", we want that not only every honest node must agree on the leader, but also that this leader belongs to $\mathcal{O}_r$ for some a-priori known round r — jumping ahead, later in our BA protocol, everyone would attempt to propose a value during this round r and the proposal of the elected leader will be chosen.

Intuitively, we would like that the LE achieves a good outcome with $O(1)$ probability. Our actual definition turns out to be tricky due to compositional issues that arise due to multiple LE instances sharing the same PKI. We would like that even when multiple LE instances share the same PKI, roughly speaking, almost surely there is still *independent* constant probability that each individual instance's outcome is good. In formal definition (see Section 3.6), we will precisely specify which subset of honest coins that are freshly chosen in each LE instance allow us to capture this desired independence. Note that this independence property is desired because later in our BA protocol, we need to argue that after a bounded number of trials, an honest leader must be elected except with negligible probability.

Construction. Our LE protocol is in fact inspired by the *asynchronous* leader election protocol by Canetti and Rabin [15]. Since our LE construction is rather technical, we explain a high-level intuition here while deferring the full protocol to Section 3.6. The idea is for everyone i to choose n coins denoted $c_{i,1}, \ldots, c_{i,n} \in \mathbb{F}$, one for each person. All these coins will be committed to using a VSS protocol such that corrupt nodes cannot choose their coins after seeing honest coins. Each person j's *charisma* is the product of the coins chosen for him by at least $\lfloor n/2 \rfloor + 1$ others, i.e., $\prod_{i \in D_j} c_{i,j}$ where $D_j \subseteq [n]$ and $|D_j| \geq \lfloor n/2 \rfloor + 1$ — in this way, at least one in D_j is honest and has chosen a random coin. In our protocol, every person j will announce this set D_j itself through an RBC protocol. Ideally we would like nodes to agree on a set of candidates that contain many nodes in $\mathcal{O}_r$ for some r, and elect the candidate with the maximum charisma (lexicographically) from this set — unfortunately at this moment we do not have Byzantine Agreement yet. Thus we must accomplish this task without reaching agreement. Our idea is for each node to *independently calculate a sufficiently large set of candidates; and although honest nodes may not agree on this candidate set, honest nodes' candidate sets must all contain every node in $\mathcal{O}_r$*. We stress that the challenge is that honest offline nodes' candidate sets must also satisfy this property even though they are receiving only an arbitrary subset of messages chosen by the adversary — note that these nodes basically have "asynchronous" networks. Perhaps more challengingly, it could be that every honest node may be offline in some round, and thus everyone's network may be asynchronous at some point.

Towards this end, we adapt Canetti and Rabin's leader election idea [15] to our weakly synchronous setting: specifically, everyone first reliably broadcasts a *tentative* candidate set S, but they keep maintaining and growing a local candidate set denoted $S^* \supseteq S$. They would keep adding nodes that they newly deem eligible to their local set S^*, until at some point, they decide that their local set S^* is sufficiently inclusive based on sufficiently many tentative candidate sets that have been reliably broadcast. At this moment, the node stops growing its local candidate set and outputs the candidate with maximum charisma from its current local set. We refer the reader to Section 3.6 for a detailed description.

3.1.4 Byzantine Agreement (BA)

The next question is how to construct BA given weakly synchronous LE. This step turns out to be non-trivial too. In particular, we stress that existing synchronous BA protocols [2, 32, 40] are broken under 0.5-weak-synchrony, not only because they lack a good leader election (or common coin) algorithm — in fact even if we replaced the leader election in existing schemes with an ideal version (e.g., our own leader election scheme in Section 3.6), the resulting BA schemes would still be broken under 0.5-weak-synchrony. All existing synchronous BA schemes make use of synchrony in a *strong* manner: they rely on the fact that if an honest node i sees some message m in round t, then i is surely able to propagate the message to *every* honest node by the end of round $t+\Delta$. This assumption is not true in our model since our model does not provide any message delivery guarantees for offline honest nodes. Instead, our protocol makes use of only weak synchrony and specifically the following observation (and variants of it): if $\lfloor n/2 \rfloor + 1$ number of nodes declare they have observed a message m by the end of round t, then at least one of them must be in $\mathcal{O}_t$ and if all of these nodes try to propagate the message m to others in round t, then everyone in $\mathcal{O}_{t^*}$ where $t^* \geq t + \Delta$ must have observed m by the end of round t^*.

At a very high level, our protocol proceeds in epochs. We make the following simplifying assumptions for the time being: 1) $\Delta = 1$, and 2) every node keeps echoing every message they have seen in every round (in our later technical sections we will remove the need for infinite echoing):

- *Propose:* For the first epoch, the designated sender's signature on a bit is considered a valid proposal. For all other epochs, at epoch start a leader election protocol is invoked to elect a leader. Recall that with constant probability, the leader election algorithm guarantees the following "good" event G: 1) the LE protocol guarantees that the elected leader is in $\mathcal{O}_r$ for some pre-determined round r; and 2) no two honest nodes output inconsistent leaders. Now imagine that in precisely round r of this epoch, everyone tentatively proposes a random bit b — and if the node indeed gets elected as a leader the proposed bit will be recognized as a valid proposal[1].
- *Vote (formally called "Prepare" later):* Let T_{le} be the liveness parameter of the LE scheme. In round T_{le} of the epoch e, a node votes on the elected leader's proposal if in epoch $e - 1$ majority nodes complained of not having received majority votes for either bit — in this case no honest node can have made a decision yet. Otherwise if the node has observed majority votes for some bit b' from the previous epoch $e - 1$, it votes for b' — in this case some honest node might have made a decision on b' and thus we might need to carry on the decision. Henceforth the set of majority votes for b' from epoch $e - 1$ is said to be an epoch-e pseudo-proposal for b'.
- *Commit:* In round $T_{\text{le}} + 1$ of the epoch e, a node sends an epoch-e commit message for a bit b, iff it has observed majority epoch-e votes on b, and no epoch-e proposal or pseudo-proposal for $1 - b$ has been seen.
- *Complain:* In round $T_{\text{le}} + 2$ of the epoch e, send a complaint if neither bit gained

[1] This is necessary because if a single proposer made a proposal *after* being elected, the adversary could make the proposer offline in that precise round.

majority votes in this epoch.

At any time, if $\lfloor n/2 \rfloor + 1$ number of commits from the same epoch and for the same bit b have been observed, output b and continue participating in the protocol (we describe a termination technique in the online full version [28]).

Remark 3.1.1. *We point out that although our* BA *protocol might somewhat resemble the recent work by Abraham et al. [2], their protocol is in fact broken under* 0.5*-weak-synchrony (even if they adopted an ideal leader election protocol) for a couple of reasons. In their protocol, in essence a node makes a decision if the node itself has seen majority votes and no conflicting proposal. To ensure consistency under weak synchrony, our protocol makes a decision when majority votes have been collected and moreover, majority nodes have declared that they have not seen a conflicting proposal (or pseudo-proposal). Finally, we introduce a "complain" round, and technically this (and together with the whole package) allows us to achieve liveness under* 0.5*-weak-synchrony — in comparison, Abraham et al.'s protocol [2] appears to lack liveness under weak synchrony.*

3.1.5 Multi-Party Computation

We now consider multi-party computation in a weakly synchronous network. Specifically, we will consider the task of secure function evaluation (SFE). Imagine that n nodes each has an input where node i's input is denoted x_i. The nodes would like to jointly compute a function $f(x_1, \ldots, x_n)$ over their respective inputs. The privacy requirement is that besides learning the outcome, each node must learn nothing else (possibly in a computational sense). Recall that earlier in our Byzantine Agreement (BA) protocols, there is no privacy requirement, and therefore our goal was to ensure that honest nodes who drop offline do not risk inconsistency with the rest of the network. With SFE, we would like to protect not only the consistency but also the *input-privacy* of those who are benign but drop offline or have unstable network connection.

Of course, in a weakly synchronous environment, if we would like online nodes to obtain outputs in a bounded amount of time, we cannot wait forever for offline honest nodes to come online. Thus, in our definition, we require that 1) *at least $n/2$ honest nodes' inputs be included in the computation*; and 2) every honest node that remains online during the protocol must get their inputs incorporated. Note that the second requirement ensures that our notion is strictly stronger (i.e., more robust) than classical synchronous MPC under honest majority.

Construction. Our goal is to construct an expected constant-round SFE protocol secure under 0.5-weak-synchrony. The naïve approach of taking *any* existing MPC and replacing the "broadcast" with our weakly synchrounous BA (see earlier subsections of this section) may not solve the problem. Specifically, we need to additionally address the following challenges:

1. Classical synchronous MPC protocols are not required to provide secrecy for honest nodes who even temporarily drop offline. Once offline, an honest node's input may be reconstructed and exposed by honest nodes who still remain online.
2. Many standard MPC protocols [10, 25] require many pairs of nodes to have finished several rounds of pairwise interactions to make progress. Even if such protocols required only constant number of rounds in the classical synchronous model, they may suffer from bad round complexity in our model — recall that in a weakly synchronous network, nodes do not have persistent online presence; thus it can take (super-)linear number of rounds for sufficiently many pairs of nodes to have had an

opportunity to rendezvous.

To tackle these challenges we rely on a Threshold Multi-Key Fully Homomorphic Encryption (TMFHE) scheme [7, 26]. In a TMFHE scheme [7],

1. Each node i can independently generate a public key denoted pk_i and register it with a PKI.
2. Now, each node i can encrypt its input x_i resulting in a ciphertext CT_i.
3. After collecting a set of ciphertexts $\{\mathsf{CT}_i\}_{i\in S}$ corresponding to the nodes $S \subseteq [n]$, any node can independently perform homomorphic evaluation (for the function f) on the ciphertext-set $\{\mathsf{CT}_i\}_{i\in S}$ and obtain an encryption (denoted $\widetilde{\mathsf{CT}}$) of $f(\{x_i\}_{i\in S})$.
4. Now, each node i can evaluate a partial decryption share of $\widetilde{\mathsf{CT}}$ such that if sufficiently many partial decryption shares are combined, one can reconstruct the plaintext evaluation outcome $f(\{x_i\}_{i\in S})$.

In our protocol, in round 0, every node i will compute an TMFHE ciphertext (denoted CT_i) that encrypts its own input and compute a NIZK proof (denoted π_i) attesting to well-formedness of the ciphertext. The pair (CT_i, π_i) will be broadcast by invoking an instance of our BA protocol described in Section 3.7. Let T_{ba} be the liveness parameter of BA. Now, every honest node in $\mathcal{O}_{T_{\text{ba}}}$ will have obtained outputs from all BA instances at the beginning of round T_{ba}. From the outputs of these BA instances, nodes in $\mathcal{O}_{T_{\text{ba}}}$ can determine the effective-input set $\mathfrak{I}$ — specifically if any BA instance that has produced a well-formed output with a valid NIZK proof, the corresponding sender will be included in the effective-input set. Observe that everyone in $\mathcal{O}_0$ will be included in $\mathfrak{I}$. Now, in round T_{ba}, any node who has produced outputs from all n BA instances will perform homomorphic evaluation independently over the collection of ciphertexts $\{\mathsf{CT}_i\}_{i\in\mathfrak{I}}$. They will then compute and multicast a partial decryption share and a NIZK proof vouching for the correctness of the partial decryption share. Now, everyone in $\mathcal{O}_t$ for $t \geq T_{\mathrm{ba}}$ will have received sufficiently many decryption shares in round t to reconstruct the evaluation outcome.

Comparison with "lazy MPC". Interestingly, the recent work by Badrinarayanan et al. [7] propose a related notion called "lazy MPC"; and their goal is also to safeguard the inputs of those who are benign but drop out in the middle of the protocol. Their model, however, is overly restrictive:

1. first, Badrinarayanan et al. [7] require that a set of majority number of honest nodes to be online *forever*;
2. not only so, they also make the strong assumption that nodes who drop offline never come back (and thus we need not guarantee liveness for nodes who ever drop offline).

As mentioned, in long-running distributed computation environments (e.g., decentralized blockchains where a secure computation task may be repeated many times over the course of years), most likely no single node can guarantee 100% up-time (let alone majority). From a technical perspective, the existence of a majority "honest and persistent online" set also makes the problem significantly easier. For example, for BA, there is in fact a simple compiler that compiles any existing honest-majority, strongly synchronous BA to a setting in which the existence of majority "honest and persistent online" set is guaranteed: basically, simply run an honest-majority, strongly synchronous BA protocol denoted BA_0. If BA_0 outputs a value v, multicast a signed tuple ($\texttt{finalize}$, v). Output v iff $\lfloor n/2 \rfloor + 1$ number of ($\texttt{finalize}$, v) messages have been received with valid signatures from distinct nodes. In fact, this simple protocol also ensures liveness for drop-outs who come back online.

Under our definition of weak synchrony, realizing BA is highly non-trivial (see earlier subsections of this section). Once we realize BA, our approach for realizing MPC is reminiscent of Badrinarayanan et al. [7]. There is, in fact, a notable difference in a low-level subtlety: in Badrinarayanan et al. [7]'s lazy MPC model, they can afford to have sufficiently many pairs of nodes engage in several rounds of pairwise interaction, whereas in our model, it can take (super-)linear number of rounds for sufficiently many pairs of nodes to have had an opportunity to rendezvous. For this reason, we need to use a strengthened notion of Threshold Multi-Key Fully Homomorphic Encryption (TMFHE) in comparison with Badrinarayanan et al. [7]. More detailed discussion of these technicalities are inclued in the online full version [28].

3.2 Defining a Weakly Synchronous Execution Model

A protocol execution is formally modeled as a set of Interactive Turing Machines (ITMs). The execution proceeds in *rounds*, and is directed by a non-uniform probabilistic polynomial-time (p.p.t.) environment denoted $\mathcal{Z}(1^\kappa)$ parametrized by a security parameter $\kappa \in \mathbb{N}$. Henceforth we refer to ITMs participating in the protocol as *nodes* and we number the nodes from 1 to $n(\kappa)$ where n is chosen by $\mathcal{Z}$ and may be a polynomial function in κ.

3.2.1 Modeling Corruption and Network Communication

We assume that there is a non-uniform p.p.t. adversary $\mathcal{A}(1^\kappa)$ that may communicate with $\mathcal{Z}$ freely at any time during the execution. $\mathcal{A}$ controls a subset of nodes that are said to be *corrupt*. All corrupt nodes are fully within the control of $\mathcal{A}$: $\mathcal{A}$ observes a node's internal state the moment it becomes corrupt and henceforth all messages received by the corrupt node are forwarded to $\mathcal{A}$; further, $\mathcal{A}$ decides what messages corrupt nodes send in each round. In this paper, we assume that corruption is *static*, i.e., the adversary $\mathcal{A}$ decides which nodes to corrupt prior to the start of the protocol execution.

Nodes that are not corrupt are said to be *honest*, and honest nodes faithfully follow the prescribed protocol for as long as they remain honest. In each round, an honest node can either be *online* or *offline*.

Definition 3.2.1 (Honest and online nodes)**.** *Throughout the paper, we shall use the notation $\mathcal{O}_r$ to denote the set of honest nodes that are online in round r. The set $\mathcal{O}_r$ is also called the "honest and online set" of round r. For $i \in \mathcal{O}_r$, we often say that i is honest and online in round r.*

We make the following assumption about network communication — note that our protocol is in the *multicast* model, i.e., every protocol message is sent to the set of all nodes:

Assumption 1 (Message delivery assumption)**.** *We assume that if someone in $\mathcal{O}_r$ multicasts a message* m *in round r, then everyone in $\mathcal{O}_t$ where $t \geq r + \Delta$ will have received* m *at the beginning of round t.*

In other words, an honest and online node is guaranteed to be able to deliver messages to the honest and online set of nodes Δ or more rounds later. The adversary $\mathcal{A}$ may delay or erase honest messages arbitrarily as long as Assumption 1 is respected.

Remark 3.2.2 (Offline nodes' network communication)**.** *Note that the above message delivery assumption implies that messages sent by honest but offline nodes can be arbitrarily delayed or even completely erased by the adversary. Further, the adversary can control which subset of honest messages each offline node receives in every round; it can omit an arbitrary subset of messages or even all of them from the view of honest offline nodes for as long as they remain offline.*

Remark 3.2.3. *We stress that a node is not aware whether it is online or offline. This makes protocol design in this model more challenging since the adversary can carefully choose a subset of messages for an offline (honest) node to receive, such that the offline node's view can appear perfectly "normal" such that it is unable to infer that it is offline. Jumping ahead, a consensus protocol secure in our model should guarantee that should an offline node make a decision while it is offline, such decisions would nonetheless be safe and would not risk inconsistency with the rest of the network.*

Our protocol needs to be aware of the parameters Δ and n. Throughout we shall assume that Δ and n are polynomial functions in κ. Formally, we can imagine that $\mathcal{Z}$ inputs Δ and n to all honest nodes at the start of the execution. Throughout the paper, we assume that $(\mathcal{A}, \mathcal{Z})$ respects the following constraints:

> $\mathcal{Z}$ always provides the parameters n and Δ to honest nodes at the start of the execution such that n is the total number of nodes spawned in the execution, and moreover, the adversary $\mathcal{A}$ respects Assumption 1.

Schedule within a round. More precisely, in each round r, the following happens:

1. First, each honest node receives inputs from $\mathcal{Z}$ and receives incoming messages from the network; note that at this moment, $\mathcal{A}$'s decision on which set of incoming

messages an honest node receives will have bearings on whether this honest node can be included in $\mathcal{O}_r$;

2. Each honest node then performs polynomially bounded computation and decides what messages to send to other nodes — these messages are immediately revealed to $\mathcal{A}$. Further, after the computation each honest node may optionally send outputs to $\mathcal{Z}$.

3. At this moment, $\mathcal{A}$ decides which nodes will belong to $\mathcal{O}_r$ where r denotes the current round. Note that $\mathcal{A}$ can decide the honest and online set $\mathcal{O}_r$ of the present round after seeing what messages honest nodes intend to send in this round.

4. $\mathcal{A}$ now decides what messages each corrupt node will send to each honest node. Note also that $\mathcal{A}$ is *rushing* since it can see all the honest messages before deciding the corrupt nodes' messages.

5. Honest nodes send messages over the network to other nodes (which may be delayed or erased by $\mathcal{A}$ as long as Assumption 1 is satisfied).

Definition 3.2.4 (χ-weak-synchrony)**.** *We say that* $(\mathcal{A}, \mathcal{Z})$ *respects* χ*-weak-synchrony (or that* $\mathcal{A}$ *respects* χ*-weak-synchrony), iff in every round* r, $|\mathcal{O}_r| \geq \lfloor \chi \cdot n \rfloor + 1$.

To aid understanding, we make a couple of remarks regarding this definition. First, observe that the set of honest and online nodes need not be the same in every round. This allows us to model churns in the network: nodes go offline and come online; and we wish to achieve consistency for *all* honest nodes, regardless of whether they are online or offline, as long as sufficiently many nodes are online in each round. Second, the requirement of χ-weak-synchrony also imposes a corruption budget. As an example, consider the special case when $\chi = 0.5$ and n is an even integer: if $(\mathcal{A}, \mathcal{Z})$ respects 0.5-weak-synchrony, it means that the adversary controls at most $n/2 - 1$ nodes. It could be that the adversary in fact controls fewer, say, $n/3$ number of nodes. In this case, up to $n/2 - 1 - n/3$ honest nodes may be offline in each round, and jumping ahead, in a consensus protocol we will require that consistency hold for these honest but offline nodes as well.

Finally, note also that our weakly-synchronous model is a generalization of the classical synchronous model: in the classical synchronous model, it is additionally required that for every r, $\mathcal{O}_r$ must be equal to the set of all nodes that remain honest till the end of round r (or later).

3.2.2 Modeling Setup Assumptions

In the plain model without any setup assumptions, Lamport et al. [35] showed that no consensus protocol could tolerate $1/3$ or more corruptions; however for $< 1/3$ corruptions, one can construct protocols that tolerate arbitrary network partitions by adopting the partially synchronous model [16, 22, 34]. It is also known that assuming a public-key infrastructure (PKI) and computationally bounded adversaries, one can construct consensus protocols that tolerate arbitrarily many corruptions in the classical fully synchronous model. Thus the interesting open question is whether, assuming the existence of a PKI and computationally bounded adversaries, we can construct protocols that tolerate more than $1/3$ corruptions and yet provide some quantifiable degree of partition tolerance. Therefore, throughout this paper we shall assume the existence of a PKI and computationally bounded adversaries. We assume that the adversary chooses which nodes to corrupt before the PKI is established.

3.2.3 Weakly Synchronous Byzantine Agreement

We now define Byzantine Agreement (BA) in a weakly synchronous network. The consistency definition is standard except that now we require consistency for honest nodes regardless of whether they are online or offline. For validity, if the sender is honest but offline initially, we cannot hope that the protocol will somehow make up for the time lost waiting for the sender to come online, such that honest and online nodes would output by the same deadline. Thus we require validity to hold only if the sender is not only honest but also online in the starting round. For liveness, we cannot hope that honest but offline nodes obtain outputs quickly without the risk of being inconsistent with the rest of the network. Thus, we require that as soon as an honest node is online at time T or greater (where T is also called the liveness parameter), it must produce an output if it has not done so already.

Syntax. A Byzantine Agreement (BA) protocol must satisfy the following syntax. Without loss of generality, we assume that node 1 is the designated sender. Before protocol start, the sender receives an input bit b from $\mathcal{Z}$; and all other nodes receive no input. The nodes then run a protocol, and during the protocol every node may output a bit.

Security. Let $T(\kappa, n, \Delta)$ be a polynomial function in the stated parameters. For $P \in \{$consistency, validity, T-liveness$\}$, a BA protocol is said to satisfy property P w.r.t. some non-uniform p.p.t. $(\mathcal{A}, \mathcal{Z})$ that is allowed to spawn multiple possibly concurrent BA instances sharing the same PKI, iff there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\kappa \in \mathbb{N}$, except with $\mathsf{negl}(\kappa)$ probability over the choice of protocol execution, the corresponding property as explained below is respected in all BA instances spawned —

henceforth we rename the starting round of each BA instance to be round 0 and count rounds within the same instance accordingly afterwards:

- *Consistency.* If honest node i outputs b_i and honest node j outputs b_j, it must be that $b_i = b_j$.
- *Validity.* If the sender is in $\mathcal{O}_0$, any honest node's output must be equal to the sender's input.
- *T-liveness.* Any node in $\mathcal{O}_r$ for $r \geq T$ must have output a bit by the end of round r.

We say that a BA protocol satisfies property $P \in \{$consistency, validity, and T-liveness$\}$ under χ-weak-synchrony if it satisfies the property P w.r.t. any non-uniform p.p.t. $(\mathcal{A}, \mathcal{Z})$ that respects χ-weak-synchrony and is allowed to spawn multiple possibly concurrent BA instances sharing the same PKI. Henceforth, if a BA protocol satisfies consistency, validity, and T-liveness under χ-weak-synchrony, we also say that the protocol is a "χ-weakly-synchronous BA protocol".

Remark 3.2.5 (Worst-case vs expected notions of liveness)**.** *We note that T-liveness defines a worst-case notion of liveness. In the remainder of the paper, we sometimes use an expected round complexity notion. We say that our* BA *protocol is expected constant round, iff there is a random variable R whose expectation is constant such that everyone in $\mathcal{O}_r$ where $r \geq R$ should have produced an output by the end of round r.*

Multi-valued agreement. The above definition can be extended to multi-valued agreement where nodes agree on a value from the domain $\{0,1\}^{\ell(\kappa)}$ rather than a single bit. Multi-valued agreement can be obtained by parallel composition of ℓ instances of BA. In this paper, we will refer to the multi-valued version as Byzantine Agreement (BA) too.

3.3 Lower Bounds

3.3.1 Impossibility of Weakly-Synchronous Consensus for $\chi \leq 0.5$

First, we show that for any $\chi \leq 0.5 - \frac{1}{n}$, it is impossible to achieve BA under χ-weak-synchrony. The intuition for this lower bound is simple: if a BA protocol allows a minority set of online nodes to reach agreement without hearing from the offline nodes, then two minority camps could independently reach agreement thus risking consistency. We formalize this intuition in the following theorem.

Theorem 5. *For any $\chi \leq 0.5 - \frac{1}{n}$, for any polynomial function T, no BA protocol Π can simultaneously achieve consistency, validity, and T-liveness under χ-weak-synchrony.*

Proof. Please refer to the online full version [28]. □

We point out that the above the lower bound holds even if $\mathcal{A}$ is restricted to scheduling the same honest and online set throughout, i.e., $\mathcal{O}_0 = \mathcal{O}_1 = \ldots$, has to decide the message delivery schedule in advance, and even when no node is corrupt. Moreover, the lower bound holds even for randomized protocols, allowing computational assumptions, and allowing additional setup assumptions (e.g., PKI, random oracle, or the erasure model).

Best-possible partition tolerance. In light of Theorem 5, a BA protocol secure under 0.5-weak-synchrony is also said to be best-possible partition tolerant.

3.3.2 Corrupt-Majority Protocols Sacrifice Partition Tolerance

It is well-known that there exist Byzantine Agreement protocols that tolerate arbitrarily many byzantine faults [20] under the classical synchronous model henceforth referred to as *strong* synchrony. If we adopted the classical strong synchrony model we might be misled to think that protocols that tolerate corrupt majority are strictly more robust than those that tolerate only corrupt minority. In this section, however, we show that corrupt-majority protocols (under strong synchrony) in fact sacrifice partition tolerance in exchange for tolerating corrupt majority, and this is inherent. As explained earlier, in real-world scenarios such as decentralized cryptocurrencies, partition tolerance seems to be a more important robustness property.

It is not too difficult to see that any corrupt-majority, strongly-synchronous protocol cannot be secure under 0.5-weak-synchrony. Specifically, with a corrupt-majority strongly-synchronous protocol, if the network partitions into multiple minority connected components, each component will end up reaching its own independent decision. We can generalize this intuition and prove an even stronger statement: any strongly-synchronous protocol that tolerates more than $\nu \geq 0.5$ fraction of corruptions cannot be secure under ν-weak-synchrony, i.e., such a protocol cannot guarantee consistency for all honest nodes (including offline ones) even if we make the strong assumption that at least ν fraction of honest nodes are online. In other words, *the more corruptions the protocol tolerates under strong synchrony, the less partition tolerant it becomes*. To state our theorem precisely, we introduce the following notation:

- We say that $(\mathcal{A}, \mathcal{Z})$ respects μ-strongly-synchronous iff at least $\lfloor \mu n \rfloor + 1$ nodes are honest and moreover all honest nodes are forever online. We say that a BA protocol satisfies property $P \in \{\text{consistency, validity, and } T\text{-liveness}\}$ under μ-strong-synchrony iff it satisfies property P w.r.t. any non-uniform p.p.t. $(\mathcal{A}, \mathcal{Z})$ that respects μ-strong-synchrony.

- Let $\mathcal{BA}\{\mu\}$ be the family that contains every protocol Π satisfying the following: $\exists$ a polynomial function $T(\cdot,\cdot,\cdot)$ s.t. Π that satisfy consistency, validity, and $T(\kappa, n, \Delta)$-liveness under μ-strong-synchrony.
- Let $\mathcal{BA}^{+}\{\chi\}$ be the family that contains every protocol Π satisfying the following: $\exists$ a polynomial function $T(\cdot,\cdot,\cdot)$ s.t. Π that satisfy consistency, validity, and T-liveness under χ-weak-synchrony.

Theorem 6. $\forall 0 < \mu < 0.5, \chi \leq 1 - \mu - 2/n,\ \mathcal{BA}\{\mu\} \cap \mathcal{BA}^{+}\{\chi\} = \emptyset$.

Proof. Please refer to the online full version [28]. □

3.4 Reliable Broadcast (RBC)

In our upper bound sections (Sections 3.4, 3.5.2, 3.6, 3.7, and the MPC upper bound in the online full version [28]) for convenience, we will make a slightly stronger assumption on the underlying network — but in fact this stronger assumption can be realized from Assumption 1 described earlier.

Assumption 2 (Strong message delivery assumption). *If $i \in \mathcal{O}_r$ and i has multicast or received a message* m *before the end of round r, then everyone in $\mathcal{O}_t$ where $t \geq r + \Delta$ will have received* m *at the beginning of round t.*

In the online full version [28], we describe how to realize Assumption 2 through a simple echo mechanism: roughly speaking, nodes echo and retry sending messages they have seen until they believe that the message has become part of the honest and online nodes' view.

3.4.1 Definition

We define a primitive called reliable broadcast (RBC) that allows a designated sender to broadcast a message, guaranteeing consistency regardless of whether the sender is honest or online, and additionally guaranteeing liveness when the sender is not only honest but also online in the starting round. We also require a "close termination" property: even when the designated sender is corrupt, we require that if some honest node outputs in round r, then everyone in $\mathcal{O}_t$ where $t \geq r + 2\Delta$ must have output by the end of round t too. The liveness notion is defined in a similar fashion as in Section 3.2.3: since under weak synchrony we cannot guarantee progress for offline nodes, we require that any honest node who comes back online in some time T or greater will have received output (assuming an honest and initially online sender). For technical reasons that will be useful later in the proof of our Leader Election (LE) protocol, we need a stronger version of the standard consistency property: not only must honest nodes' outputs agree, there must be an efficient extractor that outputs either a bit $b \in \{0, 1\}$ or $\perp$ when given the PKI and the honest nodes' transcript in the initial T rounds as input. If any honest node indeed makes an output, the output must be consistent with the extractor's output b.

Syntax. An RBC protocol consists of the following algorithms/protocols:

- **PKI setup:** at the very beginning every node i registers a public key pk_i with the PKI;
- RBC **protocol:** all instances of RBC share the same PKI. In each RBC instance, a designated sender (whose identifier is pre-determined and publicly-known) receives a value x from the environment $\mathcal{Z}$ whereas all other nodes receive nothing. Whenever a node terminates, it outputs a value y. Henceforth we shall assume that an

admissible $\mathcal{Z}$ must instruct all nodes to start protocol execution in the same round[2];

- **Extractor** $\mathcal{E}$: a polynomial-time deterministic extractor $\mathcal{E}$ that is needed only in our security definitions and proofs, not in the real-world protocol.

Security. Let $T(n, \Delta, \kappa)$ be a polynomial function in the stated parameters. For $P \in \{T\text{-consistency, validity, } T\text{-liveness, close termination}\}$, we say that an RBC protocol Π satisfies property P under χ-weak-synchrony iff for any non-uniform p.p.t. $(\mathcal{A}, \mathcal{Z})$ that respects χ-weak-synchrony and can spawn multiple instances of RBC sharing the same PKI, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\kappa \in \mathbb{N}$, except for $\mathsf{negl}(\kappa)$ fraction of the executions in the experiment $\mathsf{EXEC}^{\Pi}(\mathcal{A}, \mathcal{Z}, \kappa)$, the following properties hold for every RBC instance:

- *T-consistency.* Let $y := \mathcal{E}(\{\mathsf{pk}_i\}_{i \in [n]}, \mathsf{Tr})$ where Tr denotes the transcript of all honest nodes in the initial T rounds of the RBC instance. Then, if any honest node ever outputs y', it must be that $y' = y$.
- *Validity.* If the sender is honest and its input is x, then if any honest node outputs x', it must be that $x' = x$.
- *T-liveness (under an honest and initially online sender).* If the sender is not only honest and but also online in the starting round of this RBC instance (henceforth the starting round is renamed to be round 0 for convenience), then every node that is honest and online in round $r \geq T$ will have produced an output by the end of round r.
- *Close termination.* If an honest node outputs in some round r, then every node that is honest and online in round $r' \geq r + 2\Delta$ will have output by the end of round r'.

[2] Later in our VSS and LE protocols that invoke RBC, the fact that the RBC's environment $\mathcal{Z}$ is admissible is guaranteed by construction.

Remark 3.4.1. *Although in general, consistency and liveness can be parametrized by different delay functions, without loss of generality we may assume that two parameters are the same T (since we can always take the maximum of the two).*

3.4.2 Construction

During the PKI setup phase (shared across all subsequent RBC instances), every node calls $(\mathsf{vk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{K}(1^\kappa)$ and registers the vk with the PKI. The portion ssk is kept secret and henceforth the node will use ssk to sign protocol messages in all future RBC instances. Henceforth, although not explicitly noted, we assume that every message is by default tagged with the current session's identifier denoted sid. Every signature computation and verification will include the sid. We also assume that each message is tagged with the purported sender such that a recipient knows under which public key to verify the signature.

1. **Propose (round 0):** In round 0, the sender multicasts $(\texttt{propose}, x)$ where x is its input, attached with a signature on the tuple.
2. **ACK (round Δ):** At the beginning of round Δ, if a tuple $(\texttt{propose}, y)$ with a valid signature has been received from the sender, multicast $(\texttt{ack}, y)$ along with a signature on the tuple.
3. **Commit (round 2Δ):** At the beginning of round 2Δ, if the node has observed $\lfloor n/2 \rfloor + 1$ number of $(\texttt{ack}, y)$ messages for the same y and with valid signatures from distinct nodes, and moreover, it has not received any conflicting $(\texttt{propose}, y')$ message (with a valid signature from the sender) for $y' \neq y$, then multicast $(\texttt{commit}, y)$ along with a signature on the tuple.

4. **Finalize (any time):** At any time, if the node has received $\lfloor n/2 \rfloor + 1$ valid $(\texttt{commit}, y)$ messages for the same y and from distinct nodes, multicast $(\texttt{finalize}, y)$ along with a signature on the tuple. At any time, if a collection of $\lfloor n/2 \rfloor + 1$ $(\texttt{finalize}, y)$ messages with valid signatures from distinct nodes have been observed, output y.

We defer the constructor of the extractor $\mathcal{E}$ to the proofs since it is needed only in the security definitions and proofs and not in the real-world protocol.

Theorem 7. *Suppose that the signature scheme employed is secure, then the above* RBC *protocol satisfies* 2Δ*-consistency, validity,* 4Δ*-liveness, and close termination under* 0.5*-weak-synchrony.*

Proof. Please refer to the online full version [28]. □

3.5 Verifiable Secret Sharing (VSS)

3.5.1 Definitions

A Verifiable Secret Sharing (VSS) allows a dealer to share a secret among all nodes and later reconstruct the secret. Standard notions of VSS [15] require that the honest transcript of the sharing phase binds to the honestly reconstructed secret. For technical reasons needed later in the proof of the Leader Election (LE), we require a stronger notion, i.e., an efficient extractor $\mathcal{E}$, knowing honest nodes' public and secret keys, must be able to extract this secret from the honest transcript during the sharing phase (and later the honestly reconstructed secret must agree with the extractor's output). We need a composable notion of secrecy which we call non-malleability — note that composition was a non-issue in previous works that achieve security against unbounded adversaries [15]. Finally, for liveness, we require that if the dealer is honest and online in the initial round, for $t \geq T$, everyone in $\mathcal{O}_t$ must have output "`sharing succeeded`". Even when the dealer is corrupt or offline, if any honest node ever outputs "`sharing succeeded`" in some round r, then everyone in $\mathcal{O}_t$ where $t \geq r + 2\Delta$ must have output "`sharing succeeded`" by the end of round t. If some honest node has output "`sharing succeeded`", then reconstruction must be successful and will terminate in T rounds for honest and online nodes.

Syntax

A Verifiable Secret Sharing (VSS) scheme for a finite field $\mathbb{F}$ consists of a setup algorithm K that is run once upfront and henceforth shared among all protocol instances where each protocol instance contains two sub-protocols called Share and $\mathsf{Reconstruct}$:

1. $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{K}(1^\kappa)$: every node i calls this algorithm to generate a public and secret key pair denoted pk_i and sk_i; and pk_i is registered with the PKI.
2. Share: A designated node called the dealer receives an input $s \in \mathbb{F}$ from $\mathcal{Z}$ and all other nodes receive no input. Now all nodes execute the Share sub-protocol for the dealer to secret-share its input. We assume that for the same VSS instance, an admissible $\mathcal{Z}$ always instructs all honest nodes to start executing Share in the same round. Should execution of Share successfully terminate, a node would output a canonical output "`sharing succeeded`".
3. $\mathsf{Reconstruct}$: All nodes execute the $\mathsf{Reconstruct}$ sub-protocol to reconstruct a secret that is shared earlier in the Share sub-protocol. We assume that an admissible $\mathcal{Z}$ always instructs all honest nodes to start executing $\mathsf{Reconstruct}$ in the same round. Should execution of $\mathsf{Reconstruct}$ successfully terminate, a node would output a reconstructed secret $s' \in \mathbb{F}$.

Besides these real-world algorithms, a VSS scheme additionally has a polynomial-time extractor algorithm $\mathcal{E}$ that is needed later in the security definitions (including the definitions of validity and non-malleability). We shall explain the extractor $\mathcal{E}$ later when we define security.

T-Liveness

Consider a pair $(\mathcal{A}, \mathcal{Z})$ that may spawn multiple (concurrent or sequential) VSS instances all of which share the same n, PKI setup, and the same Δ. Let $T(n, \Delta, \kappa)$ be a polynomial function in n, Δ, κ. We say that a VSS protocol satisfies T-liveness under χ-weak-synchrony iff for any non-uniform p.p.t. $(\mathcal{A}, \mathcal{Z})$ that respects χ-weak-synchrony (and may spawn multiple instances sharing the same PKI), there exists $\mathsf{negl}(\cdot)$ such that for any $\kappa \in \mathbb{N}$, such that except with $\mathsf{negl}(\kappa)$ probability, the following holds for every VSS instance spawned:

1. *Termination of* Share *under honest and initially online dealer:* suppose that the Share sub-protocol is spawned in round r_0, and moreover the dealer is in $\mathcal{O}_{r_0}$, then any node in $\mathcal{O}_r$ for $r \geq r_0 + T$ must have output "`sharing succeeded`" by the end of round r;
2. *Close termination of* Share*:* if an honest node i has terminated the Share sub-protocol outputting "`sharing succeeded`" in round r, then for every $r' \geq r + 2\Delta$, every node in $\mathcal{O}_{r'}$ must have terminated the Share sub-protocol outputting "`sharing succeeded`" by the end of round r';
3. *Termination of* Reconstruct*:* if by the end of some round r, some honest node has terminated the Share sub-protocol outputting "`sharing succeeded`", and moreover honest nodes have been instructed to start Reconstruct, then, anyone in $\mathcal{O}_t$ for $t \geq r + T$ must

have terminated the Reconstruct sub-protocol outputting some reconstructed value in $\mathbb{F}$ by the end of round t.

T-Validity

As before, we consider an $(\mathcal{A}, \mathcal{Z})$ pair that is allowed to spawn multiple (concurrent or sequential) VSS instances, all of which share the same n, PKI setup, and Δ. Let $T(n, \Delta, \kappa)$ be a polynomial function in its parameters. Henceforth let $\mathsf{Honest} \subseteq [n]$ denote the set of honest nodes. We say that a VSS protocol satisfies T-validity under χ-weak-synchrony, iff for every non-uniform p.p.t. $(\mathcal{A}, \mathcal{Z})$ that respects χ-weak-synchrony (and may spawn multiple VSS instances sharing the same PKI where each instance has a unique sid), there exists a negligible function $\mathsf{negl}(\cdot)$ such that except with $\mathsf{negl}(\kappa)$ probability, the following holds for every VSS instance spawned: let $s' := \mathcal{E}(\{\mathsf{pk}_i\}_{i \in [n]}, \{\mathsf{sk}_i\}_{i \in \mathsf{Honest}}, \mathsf{Tr})$ where Tr denotes the transcript observed by all honest nodes in the initial T rounds of the Share sub-protocol; it must be that

(a) if an honest node ever outputs a reconstructed secret, the value must agree with s';

(b) if $\mathcal{E}$ outputs $\perp$, then no honest node ever outputs "`sharing succeeded`"[3];

(c) if the dealer is honest and online in the round in which the Share sub-protocol was invoked, and moreover it received the input s from $\mathcal{Z}$, then $s' = s$.

Non-Malleability

Consider the following experiment $\mathsf{Expt}^{\mathcal{A}}(1^\kappa, s)$ involving an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$, as well as a challenge input $s \in \mathbb{F}$. We assume that throughout the experiment,

[3] Note that (a) implies that if $\mathcal{E}$ outputs $\perp$, then no honest node will ever output a reconstructed secret.

if an honest node outputs a string in any VSS *instance, the adversary* $\mathcal{A}$ *is notified of the node's identifier, the identifier of the* VSS *instance, as well as the corresponding output.*

1. **Setup.** First, $\mathcal{A}$ chooses which set of nodes to corrupt. Henceforth the challenger $\mathcal{C}$ acts on behalf of all honest nodes and interact with $\mathcal{A}$. The honest nodes run the honest key generation algorithm such that each picks a public/secret-key pair. The public keys are given to $\mathcal{A}$. $\mathcal{A}$ now chooses corrupt nodes' public keys arbitrarily and sends them to $\mathcal{C}$.

2. **Queries.** The adversary $\mathcal{A}$ is now allowed to (adaptively) instruct $\mathcal{C}$ to spawn as many VSS instances as it wishes. The queries can be issued at any time, including before, during, or after the challenge phase (see the **Challenge** paragraph later).

   - Whenever $\mathcal{A}$ sends $\mathcal{C}$ a tuple $(sid, \mathsf{Share}, u, x)$ where $sid \in \{0,1\}^*$ and $u \in [n]$, $\mathcal{C}$ spawns instance sid with node u as the dealer. If u is honest, $\mathcal{A}$ must additionally specify the honest dealer u's input x in this instance (otherwise the field x is ignored). Now, $\mathcal{C}$ invokes the instance's Share sub-protocol (if this has not been done already);

   - Whenever $\mathcal{A}$ sends $\mathcal{C}$ a tuple $(sid, \mathsf{Reconstruct})$ where $sid \in \{0,1\}^*$, $\mathcal{C}$ does the following: if the instance sid has been spawned, then invoke the $\mathsf{Reconstruct}$ sub-protocol for that instance (if this has not been done).

   - Whenever $\mathcal{A}$ sends $\mathcal{C}$ a tuple $(sid, \mathsf{Extract})$ and instance sid has executed for at least T rounds, then $\mathcal{C}$ computes $\mathcal{E}(\{\mathsf{pk}_i\}_{i\in[n]}, \{\mathsf{sk}_i\}_{i\in\mathsf{Honest}}, \mathsf{Tr})$ where Tr is the transcript of honest nodes in the initial T rounds of the Share sub-protocol; $\mathcal{C}$ returns the result to $\mathcal{A}$.

3. **Challenge.** At any time, $\mathcal{A}$ may send the tuple $(\texttt{challenge}, sid, u)$ to $\mathcal{C}$ where u must be an honest node and the challenge sid must not be specified in any Extract or

Reconstruct query throughout the experiment (in the past or future). $\mathcal{C}$ then spawns a challenge VSS instance identified by sid where u is the designated dealer and receives the input s; further $\mathcal{C}$ invokes the challenge instance's Share sub-protocol.

4. **Output.** Whenever the adversary $\mathcal{A}$ outputs a bit $b \in \{0, 1\}$, this bit is defined as the experiment's output.

We assume that an admissible $\mathcal{A}$ never attempts to create two VSS instances with the same sid, i.e., $\mathcal{A}$ chooses distinct session identifiers for all instances. Further, throughout the experiment, $\mathcal{A}$ is allowed to decide which honest nodes are online/offline in each round (after seeing the messages honest nodes want to send in that round). $\mathcal{A}$ also controls the message delivery schedule[4].

Definition 3.5.1 (Non-malleability for VSS)**.** *We say that a* VSS *scheme satisfies non-malleability under χ-weak-synchrony iff for any non-uniform p.p.t. $\mathcal{A}$ that respects χ-weak-synchrony, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for any $s, s' \in \mathbb{F}$,* $\left|\Pr[\mathsf{Expt}^{\mathcal{A}}(1^\kappa, s) = 1] - \Pr[\mathsf{Expt}^{\mathcal{A}}(1^\kappa, s') = 1]\right| \leq \mathsf{negl}(\kappa)$.

3.5.2 A 0.5-Weakly-Synchronous VSS Scheme

We show how to construct a 0.5-weakly synchronous VSS scheme. We will rely on the following cryptographic primitives:

1. let $\mathsf{NIZK} := (\mathsf{K}, \widetilde{\mathsf{K}}, \mathsf{P}, \mathsf{V})$ denote multi-CRS NIZK scheme that satisfies completeness, zero-knowledge, and simulation soundness (see the online full version [28]);

[4] Specifically, when honest nodes running inside $\mathcal{C}$ want to send messages, the messages are forwarded to $\mathcal{A}$, and $\mathcal{A}$ tells $\mathcal{C}$ when each honest node receives what message.

2. let $\mathsf{PKE} := (\mathsf{K}, \mathsf{Enc}, \mathsf{Dec})$ denote a perfectly correct public-key encryption scheme that preserves IND-CCA security; and

3. let RBC denote a reliable broadcast scheme that satisfies T_{rbc}-consistency, T_{rbc}-liveness, validity, and close termination under 0.5-weak-synchrony for some polynomial function T_{rbc}.

PKI setup (shared across all VSS instances): During the PKI setup phase, every node i performs the following:

- let $(\mathsf{epk}_i, \mathsf{esk}_i) \leftarrow \mathsf{PKE.K}(1^\kappa)$; $(\mathsf{vk}_i, \mathsf{ssk}_i) := \Sigma.\mathsf{K}(1^\kappa)$; $\mathsf{crs}_i \leftarrow \mathsf{NIZK.K}(1^\kappa)$; and let $(\mathsf{rpk}_i, \mathsf{rsk}_i) \leftarrow \mathsf{RBC.K}(1^\kappa)$;
- node i registers its public key $\mathsf{pk}_i := (\mathsf{epk}_i, \mathsf{crs}_i, \mathsf{vk}_i, \mathsf{rpk}_i)$ with the PKI; and it retains its secret key comprised of $\mathsf{sk}_i := (\mathsf{esk}_i, \mathsf{ssk}_i, \mathsf{rsk}_i)$.

Share (executed by the dealer): Let s be the input received from the environment, the dealer does the following:

- it splits s into n shares using a $(\lfloor n/2 \rfloor + 1)$-out-of-$n$ Shamir Secret Sharing scheme, where the i-th share is henceforth denoted s_i;
- for $i \in [n]$, it computes $\mathsf{CT}_i := \mathsf{PKE.Enc}_{\mathsf{epk}_i}(sid, s_i)$ where sid is the identifier of the current instance;
- it calls $\mathsf{NIZK.P}(\{\mathsf{crs}_i\}_{i \in [n]}, x, w)$ to compute a proof π where x and w are defined as below: $x := (sid, \{\mathsf{pk}_i, \mathsf{CT}_i\}_{i \in [n]})$ is the statement declaring that there is a witness $w := (s, \{s_i\}_{i \in [n]})$ such that for each $i \in [n]$, CT_i is a valid encryption[5] of (sid, s_i) under epk_i (which is part of pk_i); and moreover, the set of shares $\{s_i\}_{i \in [n]}$ is a valid sharing of the secret s.

[5] For simplicity, we omit writing the randomness consumed by PKE.Enc which is also part of the witness.

- finally, the dealer relies on RBC to reliably broadcast the tuple $(sid, \{\mathsf{CT}_i\}_{i\in[n]}, \pi)$ — henceforth this RBC instance is denoted RBC_0.

Share (executed by everyone): Every node i does the following (where the starting round of Share is renamed round 0):

- **Any time:** whenever the RBC_0 instance outputs a tuple of the form $(sid, \{\mathsf{CT}_j\}_{j\in[n]}, \pi)$, call NIZK.V to verify the proof π w.r.t. the statement $(sid, \{\mathsf{pk}_i, \mathsf{CT}_i\}_{i\in[n]})$; and if the check succeeds, set $\mathsf{flag} := 1$ (we assume that flag was initially 0).
- **Round T_{rbc}:** if $\mathsf{flag} = 1$, reliably broadcast the message "ok"; else reliably broadcast the message "$\perp$";
- **Any time:** whenever more than $\lfloor n/2 \rfloor + 1$ RBC instances have output "ok" and RBC_0 has output a tuple; decrypt CT_i contained in the tuple output by RBC_0 using secret key esk_i; let $(\_, s_i)$ be the decrypted outcome; now record the share s_i and output "sharing-succeeded";

Reconstruct (executed by everyone): when the Reconstruct sub-protocol has been invoked, every node i waits till the instance's Share sub-protocol has output "sharing-succeeded" and then performs the following where the set $\mathbb{S}$ is initially empty:

- let s_i be the share recorded at the end of the Share sub-protocol;
- call $\mathsf{NIZK.P}(\{\mathsf{crs}_i\}_{i\in[n]}, x, w)$ to compute a proof (henceforth denoted π_i) for the following statement $x := (sid, i, s_i, \mathsf{CT}_i)$ declaring that there is random string that causes PKE.K to output the tuple $(\mathsf{epk}_i, \mathsf{esk}_i)$ where $\mathsf{epk}_i \in \mathsf{pk}_i$; and moreover, (sid, s_i) is a correct decryption of CT_i using esk_i — the witness w includes the randomness used in PKE.K, esk_i, and the randomness of PKE.Dec.

- multicast the tuple (sid, i, s_i, π_i);
- upon receiving a tuple (sid, j, s_j, π_j) such that π_j verifies w.r.t. the statement $(sid, j, s_j, \mathsf{CT}_j)$ where CT_j was the output of RBC_0 during the Share sub-protocol, add s_j to the set $\mathbb{S}$.
- whenever the set $\mathbb{S}$'s size is at least $\lfloor n/2 \rfloor + 1$, call the reconstruction algorithm of Shamir Secret Sharing to reconstruct a secret s, and if reconstruction is successful, output the result.

Since the extractor algorithm $\mathcal{E}$ is only needed in the proofs, we defer its presentation to the online full version [28].

Theorem 8. *Without loss of generality, assume that $T_{\mathrm{rbc}} \geq 3\Delta$ (if not, we can simply define $T_{\mathrm{rbc}} := 3\Delta$); and moreover assume that the* RBC *scheme employed satisfies T_{rbc}-liveness, validity, T_{rbc}-consistency, and close termination under 0.5-weak-synchrony; the* NIZK *scheme employed satisfies zero-knowledge and simulation soundness; and the* PKE *scheme satisfies IND-CCA security and is perfectly correct. Then, the above* VSS *protocol satisfies $2T_{\mathrm{rbc}}$-liveness, T_{rbc}-validity, and non-malleability under 0.5-weak-synchrony.*

Proof. Please refer to the online full version [28]. □

3.6 Leader Election (LE)

3.6.1 Definition

A leader election (LE) protocol is an inputless protocol such that when a node terminates, it outputs an elected leader $L \in [n]$. For the outcome of LE to be considered good, we want that not only every honest node must agree on the leader, but also that this leader belongs to $\mathcal{O}_r$ for some a-priori known round r. We would like that the LE achieves a good outcome with $O(1)$ probability. Our actual definition below is somewhat tricky due to compositional issues that arise due to multiple LE instances sharing the same PKI. We would like that even when multiple LE instances share the same PKI, roughly speaking, almost surely there is still *independent* constant probability that each individual instance's outcome is good. In our formal definition below, we will precisely specify which subset of honest coins that are freshly chosen in each LE instance allow us to capture this desired independence. Note that this independence property is desired because later in our BA protocol, we need to argue that after super-logarithmically many trials, an honest leader must be elected except with negligible probability. We formalize the definitions below.

T**-liveness.** Consider an $(\mathcal{A}, \mathcal{Z})$ pair that is allowed to spawn multiple concurrent or sequential LE instances all of which share the same n, PKI setup, and Δ.

Let $T(n, \Delta, \kappa)$ be a polynomial function in its parameters. We say that an LE protocol denoted Π satisfies T-liveness under χ-weak-synchrony if for every non-uniform p.p.t. $(\mathcal{A}, \mathcal{Z})$ that respects χ-weak-synchrony and may spawn multiple LE instances sharing the same PKI, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\kappa \in \mathbb{N}$, except with $\mathsf{negl}(\kappa)$ probability, the following holds for every LE instance spawned (for the LE instance of interest, we rename its starting round to round 0):

every node in $\mathcal{O}_r$ for $r \geq T$ must have output by the end of round r.

(T^*, q)**-quality.** We consider an $(\mathcal{A}, \mathcal{Z})$ pair who can spawn $m(\kappa)$ LE instances possibly running concurrently. Henceforth let $\vec{\rho}^{\,*}_{\ell}$ denote the collection of the following randomness:

> for each node honest and online in the starting round (i.e., round 0) of the ℓ-th instance: the first $d(\kappa, n)$ bits of randomness consumed by this node in this round,

where $d(\kappa, n)$ is an appropriate polynomial function that depends on the construction. Let $\vec{\rho}$ be all randomness consumed by the entire experiment (including by $(\mathcal{A}, \mathcal{Z})$ and by honest nodes and the randomness of the PKI), and let $\vec{\rho}\backslash\vec{\rho}^*_\ell$ denote all other randomness besides $\vec{\rho}^*_\ell$.

We say that a leader election (LE) protocol satisfies (T^*, q)-quality under χ-weak-synchrony, iff for any polynomial function $m(\kappa)$, for any non-uniform p.p.t. $(\mathcal{A}, \mathcal{Z})$ that respects χ-weak-synchrony and spawns $m(\kappa)$ LE instances possibly executing concurrently, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\kappa \in \mathbb{N}$, for every $1 \leq \ell \leq m(\kappa)$, except for a $\mathsf{negl}(\kappa)$ fraction of choices for $\vec{\rho}\backslash\vec{\rho}^*_\ell$, there exist at least q fraction of choices for $\vec{\rho}^*_\ell$, such that the experiment (determined by the joint randomness choice above) would guarantee the following good events for the ℓ-th instance:

1. *Consistency:* if an honest node outputs L and another honest node outputs L', it holds that $L = L'$; and
2. *Fairness:* let L be the leader output by an honest node, we have that $L \in \mathcal{O}_{T^*}$ (assuming that the start round of the ℓ-th instance is renamed round 0).

3.6.2 Construction

The construction is a bit involved and thus we refer the reader to Section 3.1.3 for an intuitive explanation of our protocol. Below we focus on a formal description.

Let VSS denote a verifiable secret sharing scheme for inputs over the finite field $\mathbb{F}$. (see Section 3.5.2) and let T_{vss} be its liveness parameter. We now show how to construct leader election from verifiable secret sharing. In our protocol below, there are n^2 instances of VSS. Henceforth we use $\mathsf{VSS}[i, j]$ to denote the j-th instance where node i is the designated dealer. Additionally, let RBC denote a reliable broadcast protocol (see Section 3.4) whose liveness parameter is denoted T_{rbc}. Let $\Sigma := (\mathsf{K}, \mathsf{Sign}, \mathsf{Ver})$ denote a digital signature scheme.

The following protocol is executed by every node, below we describe the actions taken by node $i \in [n]$ — for simplicity we implicitly assume that every message is tagged with its purported sender:

- **PKI setup** (shared across all LE instances): each node i calls $(\mathsf{rpk}_i, \mathsf{rsk}_i) \leftarrow \mathsf{RBC}.\mathsf{K}(1^\kappa)$; $(\mathsf{vpk}_i, \mathsf{vsk}_i) \leftarrow \mathsf{VSS}.\mathsf{K}(1^\kappa)$; and $(\mathsf{vk}_i, \mathsf{ssk}_i) \leftarrow \Sigma.\mathsf{K}(1^\kappa)$. Now its public key is $(\mathsf{rpk}_i, \mathsf{vpk}_i, \mathsf{vk}_i)$ and its secret key is $(\mathsf{rsk}_i, \mathsf{vsk}_i, \mathsf{ssk}_i)$.

  In the following, we describe the leader election (LE) protocol. We assume that all LE protocols share the same PKI. Moreover, whenever a node i uses ssk_i to sign messages, the message to be signed is always tagged with the session identifier sid of the current instance and signature verification also verifies the signature to the same sid.

- **Round 0**: Node i chooses n random coins $c_{i,1}, \ldots, c_{i,n} \in \mathbb{F}$. For instances $\mathsf{VSS}[i, 1]$, ..., $\mathsf{VSS}[i, n]$ where node i is the dealer, node i provides the inputs $c_{i,1}, \ldots, c_{i,n}$ respectively to each instance. Then, node i invokes the Share sub-protocol of all n^2 instances of VSS.

- **Any round**: At any time during the protocol, if in node i's view, all n VSS instances where node j is the dealer has terminated outputting "`sharing succeeded`", we say that node i now considers j as a *qualified dealer*.

- **Round** T_{vss}: If in round T_{vss}, at least $\lfloor n/2 \rfloor + 1$ qualified dealers have been identified so far: let D be the current set of all qualified dealers; reliably broadcast the message $($`qualified-set`$, D)$ using RBC. Henceforth, we use RBC$[j]$ to denote the RBC instance where j is the sender. If not enough qualified dealers have been identified, reliably broadcast the message $\perp$.

- **Any round**: In any round during the protocol, if RBC$[j]$ has output (`qualified-set`, D_j) such that D_j is a subset of $[n]$ containing at least $\lfloor n/2 \rfloor + 1$ nodes, and moreover every node in D_j has become qualified w.r.t. node i's view so far, then node i considers j as a *candidate*, and node i records the tuple (j, D_j).

- **Round** $T_{\text{vss}} + T_{\text{rbc}}$**:** In round $T_{\text{vss}} + T_{\text{rbc}}$, do the following:

  - invoke the Reconstruct sub-protocol of all VSS instances;

  - if at least $\lfloor n/2 \rfloor + 1$ nodes are now considered candidates: let S be the set of all candidates so far; now multicast (`candidate-set`, S) along with a signature on the message.

- **Any round**: At any time, if a node i has observed a (`candidate-set`, S_j) message with a valid signature from the purported sender j where $S_j \subseteq [n]$ is at least $\lfloor n/2 \rfloor + 1$ in size, and moreover, every node in S_j is now considered a candidate by node i too, we say that node i becomes *happy* with j.

- **As soon as** node i becomes happy with at least $\lfloor n/2 \rfloor + 1$ nodes, let S_i^* be the current set of nodes that are considered candidates;

- **As soon as** the relevant VSS instances (needed in the following computation) have terminated the reconstruction phase outputting a reconstructed secret — henceforth let $c'_{u,v}$ be the secret reconstructed from instance VSS$[u, v]$:

- For every $u \in S_i^*$: let (u, D_u) be a previously recorded tuple when u first became a candidate; compute node u's *charisma* as $C_u := \prod_{v \in D_u} c'_{v,u}$.
- Output the node $u^* \in S_i^*$ with maximum charisma (where ordering between elements in $\mathbb{F}$ is determined using lexicographical comparisons).

Theorem 9. *Suppose that the* VSS *scheme satisfies* T_{vss}*-liveness,* T_{vss}*-validity, and non-malleability under* 0.5*-weak-synchrony; the* RBC *scheme satisfies* T_{rbc}*-consistency,* T_{rbc}*-liveness, validity, and close termination under* 0.5*-weak-synchrony, and the signature scheme satisfies existential unforgeability under chosen-message attack. Then, the above* LE *scheme satisfies* $(2T_{\text{vss}} + T_{\text{rbc}})$*-liveness and* $(T_{\text{vss}}, 1/2)$*-quality under* 0.5*-weak-synchrony.*

Proof. Please refer to the online full version [28]. □

3.7 Byzantine Agreement

Let LE be a leader election scheme that satisfies T_{le}-liveness and $(T'_{\text{le}}, 1/2)$-quality under 0.5-weak-synchrony where $T_{\text{le}} > T'_{\text{le}}$.

PKI setup. Upfront, every node performs PKI setup as follows: every node calls $(\mathsf{LE.pk}, \mathsf{LE.sk}) \leftarrow \mathsf{LE.K}(1^\kappa)$; further, it calls $(\mathsf{vk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{K}(1^\kappa)$. The tuple $(\mathsf{LE.pk}, \mathsf{vk})$ is the node's public key and registered with the PKI, and the tuple $(\mathsf{LE.sk}, \mathsf{ssk})$ is the node's secret key.

As before we assume that all messages, excluding the ones within the LE instance[6], are signed (using each node's ssk) and tagged with the purported sender, and honest recipi-

[6] Recall that the LE instance deals with its own message signing internally.

ents verify the signature (using the purported sender's vk) upon receiving any message. To allow multiple BA instances to share the same PKI, we assume that a message is always tagged with the current instance's session identifier sid before it is signed and the verification algorithm checks the sid accordingly. Messages with invalid signatures are discarded immediately.

Protocol. In the following, an epoch-e commit evidence for $b \in \{0, 1\}$ is a set of signatures from $\lfloor n/2 \rfloor + 1$ number of distinct nodes on the message $(\texttt{prepare}, e, b)$. Our protocol works as follows. For each epoch $e = 1, 2, \ldots$, do the following (henceforth the initial round of each epoch is renamed round 0 of this epoch):

- **Propose.** For the initial T_{le} rounds in each epoch, do the following:

  1. If the current epoch is $e = 1$, then in round 0 of epoch 1, the sender multicasts a signed tuple $(\texttt{propose}, b)$ where b is its input bit.
  2. Round 0 of every epoch: invoke an instance of the LE protocol.
  3. Round T'_{le} of every epoch: every node $i \in [n]$ flips a random coin $b_i \leftarrow_{\$} \{0, 1\}$, and multicasts a signed tuple $(\texttt{propose}, b_i)$

- **Prepare (round $T_{\text{le}} + \Delta$ of each epoch).** If $e = 1$ and a node has heard an epoch-1 proposal for b from the sender, then it multicasts the signed tuple $(\texttt{prepare}, e, b)$. Else if $e > 1$, every node performs the following:

  1. if an epoch-e proposal of the form $(\texttt{propose}, e, b)$ has been heard from an eligible epoch-e proposer which is defined by the output of LE and moreover, either an epoch-$(e-1)$ commit evidence vouching for b or $\lfloor n/2 \rfloor + 1$ epoch-$(e-1)$ complaints from distinct nodes have been observed, multicast the signed tuple $(\texttt{prepare}, e, b)$.

If LE has not produced an output in the range $[n]$ at the beginning of this round, act as if no valid proposal has been received.

2. else multicast the signed tuple $(\texttt{prepare}, e, b)$ if the node has seen an epoch-$(e-1)$ commit evidence vouching for the bit b (if both bits satisfy this then send a prepare message for each bit).

- **Commit (round $T_{\text{le}} + 2\Delta$ of each epoch).** If by the beginning of the commit round of the current epoch e, a node

  1. has heard an epoch-e commit evidence for the bit b;
  2. has not observed a valid epoch-e proposal for $1-b$ (from an eligible proposer); and
  3. has not observed any epoch-$(e-1)$ commit evidence for $1-b$;

  then multicast the signed tuple $(\texttt{commit}, e, b)$.
- **Complain (round $T_{\text{le}} + 3\Delta$ of each epoch).** If no epoch-e commit evidence has been seen, multicast the signed tuple $(\texttt{complain}, e)$.
- End of this epoch and beginning of next epoch (round $T_{\text{le}} + 4\Delta$).

Finalization. At any time during the protocol, if a node has collected $\lfloor n/2 \rfloor + 1$ commit messages (from distinct nodes) for the same epoch and vouching for the same bit b, then output b if no bit has been output yet and continue participating in the protocol (we devise a termination technique in the online full version [28]).

Theorem 10. *Suppose that the* LE *scheme satisfies* T_{le}*-liveness and* $({T_{\text{le}}}', 1/2)$*-quality under* 0.5*-weak-synchrony, the digital signature scheme employed is secure, and let* λ *be any super-logarithmic function in the security parameter* κ*. Then, the* BA *scheme above satisfies consistency, validity, and* $\lambda \cdot (T_{\text{le}} + 4\Delta)$*-liveness under* 0.5*-weak-synchrony.*

Proof. Please refer to the online full version [28]. □

CHAPTER 4

MICROFEDML: SECURE AGGREGATION FOR SMALL WEIGHTS

4.1 Related Works

Our work is inspired by the line of works on secure aggregation protocols [9, 11]. Both protocols consider a single iteration of aggregation, thus they automatically supports offline users rejoining the protocol in later iterations as every iteration the protocol starts from the scratch.

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 [37]. However, it does not allow offline nodes to rejoin the training process later without sacrificing privacy. Several recent works also employ the idea of reconstructing one layer of mask of online users. Yang et al. proposes a secure aggregation protocol LightSecAgg [56] 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 [24] 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 [50] which protects the privacy during learning process with secure multiparty computation and prevents inference over the outputs of learning with differential privacy. This work assumes all clients are online. HybridAlpha proposed by Xu et al. in [55] also adopts both differential privacy and functional encryption. It assumes honest but curious server and dishonest users.

Quantization, gradient sparsification, and weight regularization Both *quantization* and *gradient sparsification* are methods commonly used to reduce the cost of communicating gradients between nodes in the scenario of data-parallel Stochastic Gradient Descent (SGD). There is a collection of works [3, 4, 5, 6, 8, 12, 17, 18, 19, 29, 30, 36, 38, 42, 46, 47, 52, 53, 54, 57] proposing methods for quantization, gradient compression and sparsification as well as clustering leading to smaller weights. Our secure aggregation protocols, suitable to smaller weights, can be used to execute the above methods in a privacy-preserving way for federated learning setting. Moreover, weight regularization [33, 51] is a widely used technique to reduce overfitting by keeping the weights of the model small.

4.2 Preliminaries

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 1, i.e., $[n]$ denotes the set $\{1, \ldots, n\}$.

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 all 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 the power and discrete log computation happening in group $\mathbb{G}$.

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

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

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

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

Random Oracle We assume the existence of a (sequence of) random oracle(s) which answers each unique query with a uniformly random response in its output domain. We use the random oracle to guarantee that all users and the server can access the same fresh randomness for each iteration.

Shamir's Secret Sharing We use Shamir's t-out-of-n secret sharing in [48] 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:

- $\mathsf{SS.share}(s, \{x_1, x_2, \ldots, x_n\}, t) \to \{(s_1, x_1), \ldots, (s_n, x_n)\}$, 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.
- $\mathsf{SS.recon}(\{(s_1, x_1), \ldots, (s_n, x_n)\}, t) = s$, in which each pair (s_i, x_i) denotes the share s_i on index x_i. This function returns the original secret s.

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

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

- SS.exponentRecon$((g^{s_1}, x_1), \ldots, (g^{s_n}, x_n), t) = \{g^s, g^{a_1}, \ldots, g^{a_{t-1}}\}$: 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 $\mathsf{SS.recon}$ in the exponent.
- SS.exponentShare$(g^s, g^{a_1}, \ldots, g^{a_{t-1}}, x) = g^{s_x}$: with the coefficients of the polynomial in exponent, it returns a new share in 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 $s, a_1, \ldots, a_{t-1}$.

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

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

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 following tail bounds for $X \sim \mathsf{HyperGeom}(N, m, n)$:

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

4.2.1 Security definition

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

Definition 4.2.1 (Aggregation Protocol)**.** *An aggregation protocol $\Pi(\mathcal{U}, \mathcal{S}, K)$ with a set of users $\mathcal{U}$, a server $\mathcal{S}$, integers K as parameters consists of two phases: the Setup phase and the*

Aggregation phase. The Setup phase runs once at the beginning of the execution, then the Aggregation phase runs for K iterations. At the beginning of each iteration $k \in [K]$ of the Aggregation phase, each user $i \in \mathcal{U}$ holds a input x_i^k, and at the end of each iteration k, the server $\mathcal{S}$ outputs a value $w^k = \sum_{i \in \mathcal{U}} x_i^k$.

We define the correctness and privacy property of the protocol below. By saying a user i is offline during a time period, we mean user i fails to both send messages to and receive messages from the server from the start point of the period till the end point of the period. We assume that the users dropping offline in the Setup phase do not come back online later, and the users dropping offline in some iteration k of the Aggregation phase only come back online at the beginning of some later iteration $k' > k$. As we assume a star network topology in which all users only communicate with the server, we assume that the server is always online. Moreover, both the server and the users do not accept incomplete messages which can be guaranteed with a message authentication code. By semi-honest adversary, we mean that the parties controlled by the adversary follow the description of the protocol while trying to extract information from their joint view. By malicious adversary, we mean that the parties controlled by the adversary can deviate arbitrarily from the description of the protocol.

Definition 4.2.2 (Correctness with Dropouts). *Let $n = |\mathcal{U}|$. An aggregation protocol* Π guarantees correctness with δ offline rate *if for every iteration $1 \leq k \leq K$ 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.*

Ideal Functionality To define privacy property, we first describe an ideal functionality which allows the adversary to learn the sum of secrets of all honest and online users chosen by the adversarial server in every iteration. More formally, $\mathsf{Ideal}^{\delta}_{\{x_i^k\}_{i\in\mathcal{U}\setminus\mathcal{C},k}}$ 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:

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

Definition 4.2.3 (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 $\mathcal{U} = \{1,\ldots,n\}$. 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 $k = 1,\ldots,K$, all inputs vectors $X^k = \{x_1^k,\ldots,x_n^k\}$ for each iteration $1 \le k \le K$, and all sets of corrupted users $\mathcal{C} \subset \mathcal{U}$ with $|\mathcal{C}| < \gamma n$ controlled by an honest-but-curious/malicious adversary $M_{\mathcal{C}}$ 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.,*

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

4.3 Secure aggregation with Random Oracle

4.3.1 The high-level construction

In this section, we explain the high level idea of our constructions. 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 $x_1, \ldots, x_n$ while hiding each individual input x_i, each individual user i adds a mask h_i to its secret input x_i which is hidden from the server and all other users and can be cancelled out when all the masks are added up, i.e., $\sum_{i\in[n]} h_i = 0$, and sends $X_i = h_i + x_i$ to the server. By adding all X_i up, the server obtains the sum of all x_i. More concretely, assuming $i > j$ without loss of generality, each pair of users i, j first agree on a random symmetric secret mask $\mathsf{mk}_{i,j}$, then they mask their inputs by user i adding $\mathsf{mk}_{i,j}$ to x_i while user j subtracting $\mathsf{mk}_{i,j}$ 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 some masked input X_i of user i is missing, the sum of h_j of online users j will not cancel out in the final sum. To tolerate the fail-stop failure, the protocol adopts t-out-of-n Shamir's secret sharing scheme, which allows a secret value to be divided into n shares and to be reconstructed with any t shares of them while guaranteeing that anyone with less than t shares cannot obtain any information about the secret. More specifically, each user i shares its masks $\mathsf{mk}_{i,j}$ 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 their masks as long as there are at least t users are still online. Also, to prevent the server from directly reconstruct the secret when it forwards the shares for the users, each pair of users i, j first agree on a symmetric encryption key $\mathsf{ek}_{i,j}$ (with a key exchange algorithm which is introduced in Section 4.2) 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 $\mathcal{O}$, the server needs to remove $\sum_{i \in \mathcal{O}} r_i$ of the online users from $\sum_{i \in \mathcal{O}} X_i$ and cancel $\sum_{i \in \mathcal{O}} h_i$ with $\sum_{i \notin \mathcal{O}} h_i$. Thus, if user i is online in the view of at least t honest users, then r_i is reconstructed and can be removed from its masked input, and h_i is kept hidden and can be canceled 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 set of users to help it reconstruct the masks of different subset of users. By appropriately setting the threshold t, for each user the server can recover at most one mask while the other mask is kept hidden so that the input is covered. We describe the protocol in Section 4.3.2 and analyze the security in Section 4.3.3.

4.3.2 Protocol

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

Algorithm 1 Setup ($\mathsf{MicroFedML}_1$)

This protocol uses the following algorithms defined in Section 4.2: a Public key infrastructure, a Diffie-Hellman key exchange scheme $(\mathsf{KA.setup}, \mathsf{gen}, \mathsf{KA.agree})$; a CCA2-secure authenticated encryption scheme $(\mathsf{AE.enc}, \mathsf{AE.dec})$; a Shamir's secret sharing scheme $(\mathsf{SS.share}, \mathsf{SS.recon}, \mathsf{SS.exponentRecon})$. It proceeds as follows:

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

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

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

Round 1: Encryption Key Exchange

1: Each user $i \in \mathcal{U}$: generates a pair of encryption keys $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{gen}(pp)$, then signs pk_i with d_i^{SK} and sends $(\mathsf{pk}_i, \sigma_i)$ to the server , in which σ_i denotes the signature .

2: Server $\mathcal{S}$: On receiving $(\mathsf{pk}_i, \sigma_i)$ from user j, the server verifies the signature σ_j with d_j^{PK}. If the signature verification fails, ignore the message from user j. Otherwise, add j to a user list $\mathcal{U}_\mathcal{S}^1$. If $|\mathcal{U}_\mathcal{S}^1| < t$ after processing all messages from users, $\mathcal{S}$ aborts.

Otherwise, the server sends all public keys and signatures it receives from users $j \in \mathcal{U}_{\mathcal{S}}^1$ to each user in $\mathcal{U}_{\mathcal{S}}^1$.

Round 2: Mask Sharing

3: Each user i: On receiving $(\mathsf{pk}_i, \sigma_i)$ for a user $j \in \mathcal{U}$ from the server, each user i verifies the signatures σ_j with d_j^{PK}. It aborts if any signature verification fails as that indicates the server is corrupt. Otherwise, it puts j into a user list $\mathcal{U}_i^1$ and stores $\mathsf{ek}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j, \mathsf{sk}_i)$. It aborts if $|\mathcal{U}_i^1| < t$ after processing all received messages. Otherwise, user i uniformly randomly chooses r_i, and calculates the secret shares of r_i by $\{r_{i,j}\}_{j \in \mathcal{U}} \leftarrow \mathsf{SS.share}(r_i, \mathcal{U}_i^1, t)$. Then it encrypts each share $r_{i,j}$ by $c_{i,j} \leftarrow \mathsf{AE.enc}(r_{i,j}, \mathsf{ek}_{i,j})$ and sends all encrypted shares $\{c_{i,j}\}_{j \in \mathcal{U}_i^1}$ to the server.

4: Server $\mathcal{S}$: If it receives messages from less than t users, abort. Otherwise, it denotes this set of users with $\mathcal{U}_{\mathcal{S}}$. It sends each $c_{i,j}$ to the corresponding receiver j for each $i \in \mathcal{U}_{\mathcal{S}}$. Then it outputs the client set $\mathcal{U}_{\mathcal{S}}$.

Round 3: User Receiving Shares

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

Algorithm 2 Aggregation ($\mathsf{MicroFedML}_1$)

This protocol uses the following algorithms defined in Section 4.2: a Public key infrastructure, a Shamir's secret sharing scheme ($\mathsf{SS.share}$, $\mathsf{SS.recon}$, SS.exponentShare, SS.exponentReco

a hash function $H(\cdot)$. It proceeds as follows:

Input: Every user i holds its own signing key d_i^{SK} and all users' verification key d_j^{PK} for $j \in [n]$, r_i, a list of users $\mathcal{U}_i$, and $r_{j,i}$ for every $j \in \mathcal{U}_i$ it obtains in the Setup phase.

Moreover, it also holds a secret input x_i^k for every iteration k. The server $\mathcal{S}$ holds all users' verification keys, all public parameters it receives in the Setup phase, and a list of users $\mathcal{U}_\mathcal{S}$ which is its output of the Setup phase.

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

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

```
for Iteration $k = 1, 2, \dots$ do
  Round 1: Secret Sharing:
    User $i$: It calculates $X_i = x_i + r_i$ and sends $H(k)^{X_i}$ to the server.
    Server $\mathcal{S}$: Denote the set of users it receives messages from with $\mathcal{O}$. If $|\mathcal{O}| < t$, abort. Otherwise, it sends $\mathcal{O}$ to all users $i \in \mathcal{O}$.
  Round 2: Online Set Checking (Only needed in Malicious setting):
    User $i$: On receiving $\mathcal{O}$ from the server, it first checks that $\mathcal{O} \subseteq \mathcal{U}_i$ and $|\mathcal{O}| \geq t$, then signs the set $\mathcal{O}$ and sends the signature $\sigma_i$ to the server.
    Server $\mathcal{S}$: If it receives less than $t$ valid signatures on $\mathcal{O}$, abort. Otherwise, it forwards all valid signatures to all users in $\mathcal{O}$.
  Round 3: Mask Reconstruction on the Exponent:
    User $i$: On receiving signatures from the server, it first verifies the signatures with $\mathcal{O}$ and the verification keys of the other users. If there are less than $t$ valid signatures, abort. Otherwise, it calculates $\zeta_i = H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}}$. It sends $\zeta_i$ to the server.
    Server $\mathcal{S}$: If it receives $\zeta_i$ from less than $t$ users, abort. Otherwise, let $\mathcal{O}'$ denote the set of users $i$ successfully sends $\zeta_i$ to the server. The server reconstructs $R_\mathcal{O} = \mathsf{SS.exponentRecon}(\{\zeta_j, j\}_{j \in \mathcal{O}'}, t)$ and calculates the discrete log of $H(k)^{\sum_{i \in \mathcal{O}} X_i} / R_\mathcal{O}$ to get $\sum_{i \in \mathcal{O}} x_i$.
```

```
8: end for
```

4.3.3 Security

In this section, we first recap the idea of the protocol MicroFedML$_1$ which guarantees privacy against malicious adversary as a whole picture, then we discuss the security properties of the protocol MicroFedML$_1$ in different adversarial settings.

As described in Section **??**, in the Setup phase, each user i first agree with every other users j on the symmetric encryption key $\mathsf{ek}_{i,j}$, by picking a pair of secret key sk_i and public key pk_i and sends the public key pk_i to all other users j. On receiving the public key pk_j from other users, the user i combines it with its own secret key sk_i to generate $\mathsf{ek}_{i,j}$, which should be the same generated in the same way on user j's side (see Section 4.2 for more details). Then it uniformly randomly picks a random mask r_i, and calculate the secret shares of it for all other users. Before sending the shares, user i encrypts the share for user j with $\mathsf{ek}_{i,j}$ so that the server cannot learn the share from the messages it forwards for them, while they can decrypt the messages between them with $\mathsf{ek}_{i,j}$. Moreover, in the key agreement process mentioned above, a server controlled by a malicious adversary might sending users malicious public keys pk' rather than forwarding the public keys from honest users so that the user i is actually agreeing on a corrupt key with the adversary and all its encrypted messages can easily be decrypted by the adversary then. Thus, a public-key infrastructure (PKI) (see Section 4.2 for more details) which allows users to verify the source of messages is required. More specifically, in PKI, each user holds a secret signing key which is only known to itself and a public verification key which is known to all parties. User i signs the public key pk_i with its signing key generating a signature which cannot be forged by anyone not knowing its signing key and can be verified by anyone holding it public verification key that the message is from user i. It sends the signature with pk_i to the server and verifies all the signatures on pk_j's from other users j with user j's verification key.

After the Setup phase, each user should hold its own random mask r_i and one share of r_j for each of other users j. Then, in each iteration, the user i first sends the masked input in the exponent $g^{X_i} = g^{x_i+r_i}$ to the server. When user i receives the online set $\mathcal{O}$ from the server, it needs to check if all other honest users receive the same online set.

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

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

4.3.4 Correctness with dropouts

We first discuss the correctness guarantee of the protocol when all users and the server follow the protocol except that less than δ fraction of users are offline in each iteration.

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

Theorem 11. *The protocol Π instantiated with Algorithm 1 and Algorithm 2 with parameter $t = \lfloor (1-\delta)n \rfloor + 1$ guarantees correctness with δ offline rate.*

4.3.5 Privacy Against Semi-Honest Adversary

In this section, we discuss the privacy guarantee of the protocol when the semi-honest adversary controls both a subset of users and the server.

We can also reduce the round complexity by removing the second round in the Aggregation phase, as the server is assumed to follow the protocol so that it sends the same online list to each user.

Theorem 12. *The protocol Π instantiated with Algorithm 1 and Algorithm 2 running with a server $\mathcal{S}$ and n users with parameter $t = \lfloor n/2 \rfloor + 1$ guarantees privacy against $\frac{1}{2}$-fraction of semi-honest adversary with $\frac{1}{2}$ offline rate. The protocol takes 3 rounds in Setup phase and 2 rounds for each iteration of Aggregation phase.*

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

- In the Setup phase:
  - **Round 1**: Each honest user i follows the protocol description in Algorithm 1.

- **Round 2**: For each corrupt user $j \in \mathcal{U}_i \cap \mathcal{C}$, an honest user i stores $\mathsf{ek}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j, \mathsf{sk}_i)$. 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 $j \in \mathcal{U}^1_i \cap \mathcal{C}$, 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 $j \in \mathcal{U}^1_i \backslash \mathcal{C}$, 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 $H(k)^{X^*_i}$ to the server.
  - **Round 3**: If the online set $\mathcal{O}$ the honest users receive from the server is of size at least t, the simulator queries the ideal functionality to get $w = \mathsf{Ideal}(\mathcal{O} \backslash \mathcal{C}, k)$. Then for all honest users $i \in \mathcal{O} \backslash \mathcal{C}$, the simulator uniformly randomly samples w^*_i for $i \in \mathcal{O} \backslash \mathcal{C}$ under the restriction $\sum_{i \in \mathcal{O} \backslash \mathcal{C}} w^*_i = w$. For each $i \in \mathcal{O} \backslash \mathcal{C}$, the simulator SIM calculates $r^*_i = X^*_i - w^*_i$, and calculates the shares $r^*_{i,j}$ for all $j \in \mathcal{U}_i \backslash \mathcal{C}$ such that $r^*_i = \mathsf{SS.recon}(\{r^*_{i,j}, j\}_{j \in \mathcal{U}_i \backslash \mathcal{C}}, \{r_{i,j}, j\}_{j \in \mathcal{C}}, t)$, where $r_{i,j}$ for $j \in \mathcal{C}$ are the shares that have already been sent to the corrupt users in the Setup phase. Let $r^*_{j,i} = r_{j,i}$ for $j \in \mathcal{C}$ and honest user i.

    Then for the honest users i who receives $\mathcal{O}$, the simulator sends $\zeta^*_i = H(k)^{\sum_{j \in \mathcal{O}} r^*_{j,i}}$ to the server on behalf of user i.

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

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

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

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

Hyb2 In this hybrid, for any pair of two honest users i, j, the encryption of shares $c_{i,j}$ and $c_{j,i}$ they send between each other are encrypted and decrypted using a uniformly random key $\mathsf{ek}^*_{i,j}$ 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.

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

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

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

By Lemma 4.3.4, this hybrid is indistinguishable from the previous one.

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

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

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

This hybrid is identical to the previous one as there is only one unique $\mathcal{O}$ and if an honest user is not included in $\mathcal{O}$, the share $r^*_{i,j}$ for $j \in \mathcal{U}\backslash\mathcal{C}$ 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.

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

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

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

□

4.3.6 Privacy against malicious adversary

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

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

Fact 4.3.1 (Unique Common Online Set)**.** *When $t > \frac{2}{3}n$ and $|\mathcal{C}| < \frac{1}{3}n$, There is at most one common online set $\mathcal{O}$ in every iteration.*

Proof. For the sake of contradiction, assume there exists two different common online set

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

The following theorem shows that the joint view of any subset of less than t users and the server can be simulated without knowing the secret input of any other users. In other words, the adversary controlling the server and less than t users cannot learn anything other than the output of the computation.

Theorem 13. *The protocol Π instantiated with Algorithm 1 and Algorithm 2 (with underlined parts) guarantees privacy against $\frac{1}{3}$-fraction of malicious adversary with $\frac{1}{3}$ offline rate. The Setup phase of the protocol runs in 3 rounds with $O(n)$ communication complexity per user and the Aggregation phase runs in 3 rounds with $O(n)$ communication complexity per user.*

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

- In the Setup phase:
  - **Round 1**: Each honest user i follows the protocol description in Algorithm 1.
  - **Round 2**: Each honest user i receives $(\mathsf{pk}_j, \sigma_j)$ from the server, and verifies the signatures as described in Algorithm 1, except that the simulator aborts if some honest user i receives a valid signature of an honest user j on a public encryption key different from what user j sends to the server in the previous round. Then for each corrupt user $j \in \mathcal{U}_i \cap \mathcal{C}$, an honest user i stores $\mathsf{ek}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j, \mathsf{sk}_i)$. 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 $j \in \mathcal{U}_i^1 \cap \mathcal{C}$, 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 $j \in \mathcal{U}_i^1 \backslash \mathcal{C}$, 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 $H(k)^{X_i^*}$ to the server.
  - **Round 2**: Each honest user follows the protocol by signing the online set $\mathcal{O}$ it receives and sending the signature to the server.
  - **Round 3**: If any honest user receives at least t valid signatures on the online set $\mathcal{O}$ it receives in the previous round, the simulator queries the ideal functionality to get $w = \mathsf{Ideal}(\mathcal{O} \backslash \mathcal{C}, k)$. Then for all honest users $i \in \mathcal{O} \backslash \mathcal{C}$, the simulator uniformly randomly samples w_i^* for $i \in \mathcal{O} \backslash \mathcal{C}$ under the restriction $\sum_{i \in \mathcal{O} \backslash \mathcal{C}} w_i^* = w$. For each $i \in \mathcal{O} \backslash \mathcal{C}$, the simulator SIM calculates $r_i^* = X_i^* - w_i^*$, and calculates the shares $r_{i,j}^*$ for all $j \in \mathcal{U}_i \backslash \mathcal{C}$ such that $r_i^* = \mathsf{SS.recon}(\{r_{i,j}^*, j\}_{j \in \mathcal{U}_i \backslash \mathcal{C}}, \{r_{i,j}, j\}_{j \in \mathcal{C}}, t)$, where $r_{i,j}$ for $j \in \mathcal{C}$ are the shares that have already been sent to the corrupt users in the Setup phase. Let $r_{j,i}^* = r_{j,i}$ for $j \in \mathcal{C}$ and honest user i.

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

By Fact 4.3.1, there will be at most one unique set $\mathcal{O}$ that collects enough number of valid signatures and the Ideal functionality will be queried at most once each iteration.

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

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

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

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

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

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

Hyb3 In this hybrid, for any pair of two honest users i, j, the encryption of shares $c_{i,j}$ and $c_{j,i}$ they send between each other are encrypted and decrypted using a uniformly random key $\mathsf{ek}_{i,j}^*$ 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 $c^r_{i,j} = \mathsf{AE.enc}(\mathsf{ek}^*_{i,j}, r_{i,j})$ sent between honest parties in the Setup phase in the previous hybrids with the encryption of a dummy value, i.e., ${c^r_{i,j}}^* = \mathsf{AE.enc}(\perp, \mathsf{ek}^*_{i,j})$.

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

By Lemma 4.3.4, this hybrid is indistinguishable from the previous one.

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

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

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

This hybrid is identical to the previous one as there is only one unique $\mathcal{O}$ and if an honest user is not included in $\mathcal{O}$, the share $r^*_{i,j}$ for $j \in \mathcal{U} \backslash \mathcal{C}$ 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 $\sum_{i \in \mathcal{O} \backslash \mathcal{C}} x_i$ to sample r^*_i for $i \in \mathcal{O} \backslash \mathcal{C}$, the simulator makes a query to the functionality Ideal with the user set $\mathcal{O} \backslash \mathcal{C}$ and iteration counter k and use the output value to sample random value r^*_i in every iteration with $|\mathcal{O} \backslash \mathcal{C}| \geq t - n_{\mathcal{C}}$. Note that the Ideal functionality will not return $\perp$ in this case.

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

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

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

Lemma 4.3.2 (Extension of the DDH Assumption). *Let p and q be two primes while $p = 2q + 1$. Let g be a generator of field $\mathbb{Z}^*_p$. If the DDH assumption holds, then for uniformly random $a, b_1, \ldots, b_t, b'_1, \ldots b'_n \in \mathbb{Z}_q$, the following two distributions are computationally indistinguishable:*

$$(g^a, g^{b_1}, \ldots, g^{b_n}, g^{ab_1}, \ldots, g^{ab_n}) \tag{4.1}$$

$$(g^a, g^{b_1}, \ldots, g^{b_n}, g^{ab'_1}, \ldots, g^{ab'_n}) \tag{4.2}$$

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

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

and outputs the bit $\mathcal{A}$ outputs. Let $A = g^a$, $B = g^b$. Then, when $C = g^{ab}$, the distribution $\mathcal{D}$ feeds $\mathcal{A}$ is just the distribution of 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 (4.1) is the same as Hyb_n, and the distribution (4.2) is the same as Hyb_0, the two distributions in the lemma are computationally indistinguishable. □

Lemma 4.3.3. *Let $n, t, n_\mathcal{C}, K$ be integer parameters, $n_\mathcal{C} \leq n - t < t$. Let $x_1, \dots, x_{n-n_\mathcal{C}}$ be uniformly random elements of some finite field $\mathbb{F}$. Let $x_{i,j} \in \mathbb{Z}_q$ for $i \in [n - n_\mathcal{C}]$ and $j \in [n]$ be shares of x_i calculated with t-out-of-n Shamir's secret sharing algorithm, i.e., $\{x_{i,j}\}_{j \in [n]} \leftarrow \mathsf{SS.share}(x_i, [n], t)$ for $i \in [n - n_\mathcal{C}]$.*

For each $k = 1, \dots, K$, let $y_1^k, \dots, y_{n-n_\mathcal{C}}^k$ also be uniformly randomly chosen from $\mathbb{F}$. Let $y_{i,j}^k$ for $i, j \in [n - n_\mathcal{C}]$ be elements of $\mathbb{Z}_q$ such that $y_i^k = \mathsf{SS.recon}(\{y_{i,j}^k, j\}_{j \in [n-n_\mathcal{C}]}, \{x_{i,j}^k, j\}_{j \in [n-n_\mathcal{C}+1, n]}, t)$. 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 as-

sumption holds:

$$\{x_{i,j}\}_{i\in[n-n_{\mathcal{C}}],j\in[n-n_{\mathcal{C}}+1,n]}, \{\{H(k)^{x_i}\}_{i\in[n-n_{\mathcal{C}}]}, \{H(k)^{x_{i,j}}\}_{i,j\in[n-n_{\mathcal{C}}]}\}_{k\in[K]} \tag{4.3}$$

$$\{x_{i,j}\}_{i\in[n-n_{\mathcal{C}}],j\in[n-n_{\mathcal{C}}+1,n]}, \{\{H(k)^{y_i^k}\}_{i\in[n-n_{\mathcal{C}}]}, \{H(k)^{y_{i,j}^k}\}_{i,j\in[n-n_{\mathcal{C}}]}\}_{k\in[K]} \tag{4.4}$$

Proof. We prove the indistinguishability with a sequence of hybrids. Let Hyb_0 equal to the distribution (4.3). Then for each $k \in [K]$, Hyb_k is the same as Hyb_{k-1} except that $\{H(k)^{x_i}\}_{i\in[n-n_{\mathcal{C}}]}, \{H(k)^{x_{i,j}}\}_{i,j\in[n-n_{\mathcal{C}}]}$ are substituted with $\{H(k)^{y_i^k}\}_{i\in[n-n_{\mathcal{C}}]}, \{H(k)^{y_{i,j}^k}\}_{i,j\in[n-n_{\mathcal{C}}]}$. Defined in this way, Hyb_K is identical to the distribution (4.4) 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 $i \in [n-n_{\mathcal{C}}]$: $\mathsf{Hyb}_{k,i}$ is the same as $\mathsf{Hyb}_{k,i-1}$ except that $H(k)^{x_i}, \{H(k)^{x_{i,j}}\}_{j\in[n-n_{\mathcal{C}}]}$ in $\mathsf{Hyb}_{k,i-1}$ is substituted with $H(k)^{y_i^k}, \{H(k)^{y_{i,j}^k}\}_{j\in[n-n_{\mathcal{C}}]}$. Note that $\mathsf{Hyb}_{k,n-n_{\mathcal{C}}}$ 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

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

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 $i \in [n-n_{\mathcal{C}}]$, 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 $i \in [n-n_{\mathcal{C}}]$ 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 $i \in [n - n_\mathcal{C}], j \in [n - n_\mathcal{C} + 1, n]$; 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 $i, j \in [n - n_\mathcal{C}]$.
- 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 $\{Z_{k_0,i_0,j}\}_{j\in[n-n_\mathcal{C}]}$ in the following way: It calculates X_{i_0} by

$$X_{i_0}, C_1, \dots C_{t-1} = \mathsf{SS.exponentRecon}((B_1, 1), \dots (B_{t-n_\mathcal{C}}, t - n_\mathcal{C}),$$
$$(g^{x_{i_0,n-n_\mathcal{C}+1}}, n - n_\mathcal{C} + 1), \dots, (g^{x_{i_0,n}}, n), t),$$

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

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

Let $Z_{k_0,i_0} = X_{i_0}$. For $j \in [t - n_\mathcal{C}]$, let $Z_{k_0,i_0,j} = C_j$, and for $j \in [t - n_\mathcal{C} + 1, n - n_\mathcal{C}]$, 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 4.3.2, there is no such a distinguisher $\mathcal{D}'$. Thus, we have a contradiction.

□

Lemma 4.3.4. *If the DDH assumption holds, then the distributions of* Hyb4 *and* Hyb5 *are computationally indistinguishable.*

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

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

in the following way:

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

When the input of $\mathcal{D}$ is sampled from the distribution (4.3), the distribution of the simulation is identical to Hyb4, and when the the input of $\mathcal{D}$ is sampled from the distribution (4.4), the distribution of the simulation is identical to Hyb5. Thus, $\mathcal{D}$ successfully distinguishes between the two distributions when $\mathcal{A}$ succeeds. However, by Lemma 4.3.3, such a distinguisher $\mathcal{D}$ does not exists under the DDH assumption. Thus, we have a contradiction. □

4.4 MicroFedML$_2$: Improvement with user grouping

In the protocol MicroFedML$_1$ introduced in Section 4.3 we eliminate the communication cost of secret sharing in each iteration by reusing the random mask. However, each user still needs to know the online status of all participants to calculate the sum of the shares of the mask, which needs to be represented in at least $O(n)$ bits. In the malicious setting, this also means each user needs to receive $O(n)$ signatures of other online users from the server to guarantee the agreement on the online set.

To further reduce the communication cost, we divide all users into small groups so that each user only needs to know the status of a small number of neighbors in the same group in every iteration. The simplest construction is that each group of users run the previous protocol with the same central server in parallel. The server obtains the sum of all users' inputs by summing up the results of all protocol instances. Obviously, this strategy violates the security requirement that for each iteration, the server can only learn the sum of inputs of a single large subset of users. Thus, we add the mask h_i generated in a similar way as introduced in Section 4.3.1 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. More specifically, each pair of users i, j in two neighboring groups respectively agree on a mutual mask $\mathsf{mk}_{i,j}$, one adds $\mathsf{mk}_{i,j}$ to its input and the other one subtract $\mathsf{mk}_{i,j}$ from its input, so that when all masked inputs added up, the mutual masks cancel out. When the server runs the reconstruction for each group, it can only learn the sum of the group masked with the sum of the mutual masks of all online group members. Each user's mutual mask should also be secret shared in the group, so that other group members can help the server cancel out the mask if some user drops offline in the Aggregation phase. 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.

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 the assignment together. We chooses the group size in exactly the same way and under the same assumptions as neighborhood size is chosen in BBG+20 [9]. We discuss the group properties we need in Section 4.4.2.

We show the Setup phase and the Aggregation phase of MicroFedML$_2$ with privacy guarantee against malicious adversary in Figure 4.1 and Figure 4.2 respectively.

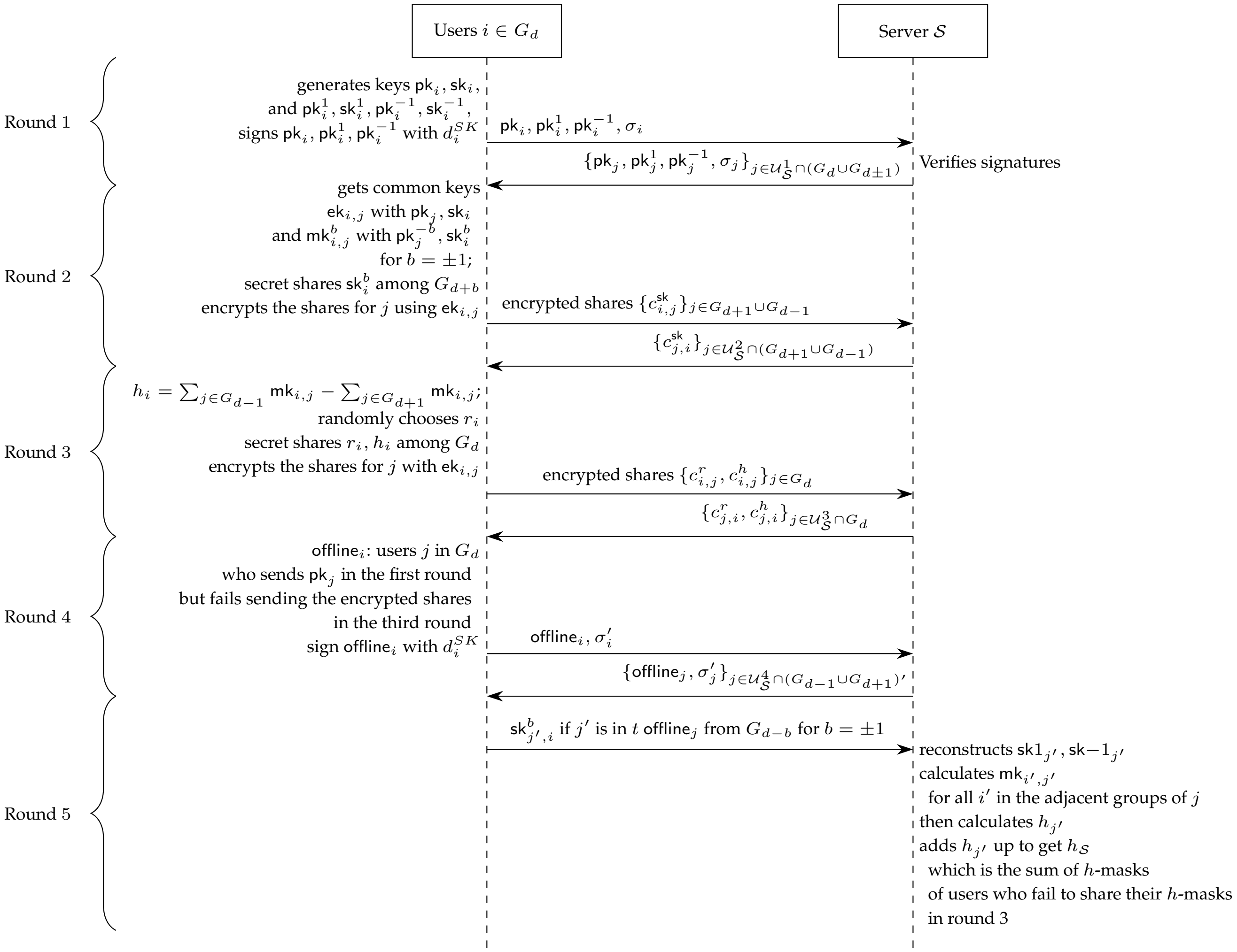

Users $i \in G_d$
Server $\mathcal{S}$
Round 1
generates keys $\mathsf{pk}_i, \mathsf{sk}_i$,
and $\mathsf{pk}_i^1, \mathsf{sk}_i^1, \mathsf{pk}_i^{-1}, \mathsf{sk}_i^{-1}$,
signs $\mathsf{pk}_i, \mathsf{pk}_i^1, \mathsf{pk}_i^{-1}$ with d_i^{SK}
$\mathsf{pk}_i, \mathsf{pk}_i^1, \mathsf{pk}_i^{-1}, \sigma_i$
$\{\mathsf{pk}_j, \mathsf{pk}_j^1, \mathsf{pk}_j^{-1}, \sigma_j\}_{j \in \mathcal{U}_\mathcal{S}^1 \cap (G_d \cup G_{d \pm 1})}$
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 $\{c_{i,j}^{\mathsf{sk}}\}_{j \in G_{d+1} \cup G_{d-1}}$
$\{c_{j,i}^{\mathsf{sk}}\}_{j \in \mathcal{U}_\mathcal{S}^2 \cap (G_{d+1} \cup G_{d-1})}$
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 $\{c_{i,j}^r, c_{i,j}^h\}_{j \in G_d}$
$\{c_{j,i}^r, c_{j,i}^h\}_{j \in \mathcal{U}_\mathcal{S}^3 \cap G_d}$
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}
$\mathsf{offline}_i, \sigma_i'$
$\{\mathsf{offline}_j, \sigma_j'\}_{j \in \mathcal{U}_\mathcal{S}^4 \cap (G_{d-1} \cup G_{d+1})'}$
Round 5
$\mathsf{sk}_{j',i}^b$ if j' is in t $\mathsf{offline}_j$ from G_{d-b} for $b = \pm 1$
reconstructs $\mathsf{sk}1_{j'}, \mathsf{sk}{-}1_{j'}$
calculates $\mathsf{mk}_{i',j'}$
for all i' in the adjacent groups of j
then calculates $h_{j'}$
adds $h_{j'}$ up to get $h_\mathcal{S}$
which is the sum of h-masks
of users who fail to share their h-masks
in round 3

Figure 4.1: An overview of the Setup phase of MicroFedML$_2$

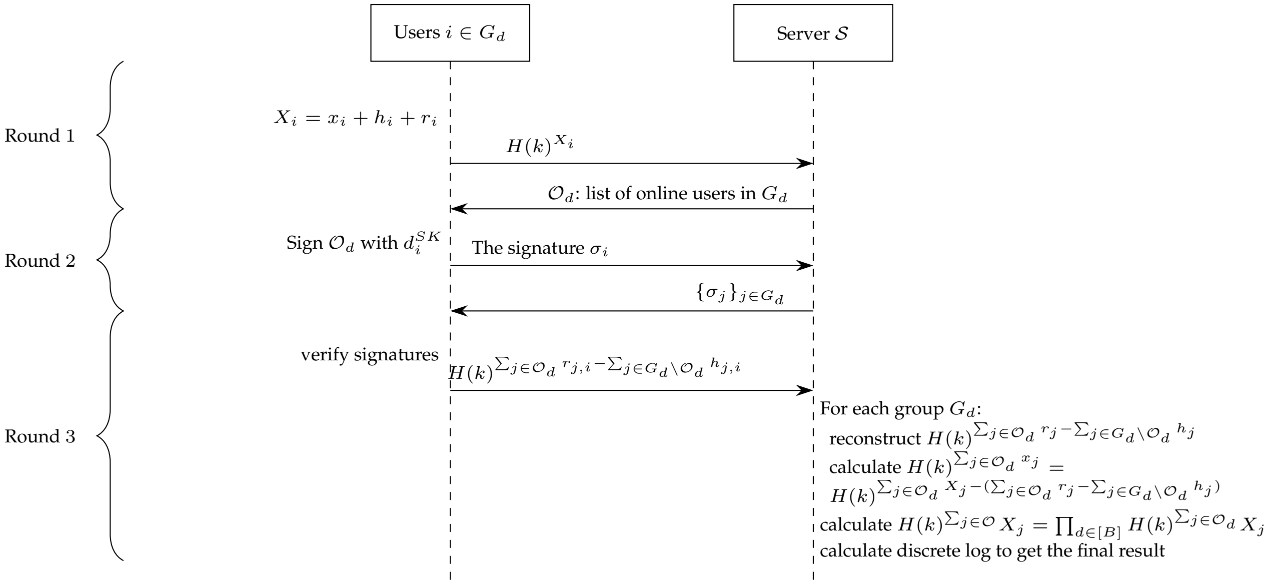

Users $i \in G_d$
Server $\mathcal{S}$
Round 1
Round 2
Round 3
$X_i = x_i + h_i + r_i$
$H(k)^{X_i}$
$\mathcal{O}_d$: list of online users in G_d
Sign $\mathcal{O}_d$ with d_i^{SK}
The signature σ_i
$\{\sigma_j\}_{j \in G_d}$
verify signatures
$H(k)^{\sum_{j \in \mathcal{O}_d} r_{j,i} - \sum_{j \in G_d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

Figure 4.2: An overview of the Aggregation phase of MicroFedML$_2$

4.4.1 Protocol with user grouping

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

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

Algorithm 3 Setup (MicroFedML$_2$)

This protocol uses the following algorithms defined in Section 4.2: a public key infrastructure, a Diffie-Hellman key exchange scheme (KA.setup, gen, KA.agree); a CCA2-secure authenticated encryption scheme (AE.enc, AE.dec); a Shamir's secret sharing scheme (SS.share, SS.recon, SS.exponentShare, SS.exponentRecon); 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 $|\mathcal{U}_{\mathcal{S}}| \geq Bt$ 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 $i \in G_d$: Let $\mathcal{U}_i^1$ denote the set of users from whom user i receives the public keys with valid signatures. Note that now $\mathcal{U}_i^1 \subseteq G_d \cup G_{d-1} \cup G_{d+1}$. It runs the key exchange algorithm to generate $\mathsf{ek}_{i,j}$ for $j \in \mathcal{U}_i^1$ as described in line 3 in Algorithm 1. Moreover, it runs the key exchange algorithm to generate $\mathsf{mk}_{i,j}$ for $j \in G_{d+b}$ with sk_i^b and pk_j^{-b} for $b = \{1, -1\}$.

It then calculates t-out-of-$\frac{n}{B}$ secret shares of sk_i^b among users in G_{d+b} to generate $\{\mathsf{sk}_{i,j}^b\}_{j \in G_{d+b}}$, 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 $\mathsf{ek}_{i,j}$ as described in line 3 of Algorithm 1. It then sends the encrypted shares $\{c_{i,j}^r, c_{i,j}^h\}_{j \in \mathcal{U}_i^1 \cap G_d}$ to the server.

6: Server $\mathcal{S}$: Denote the set of all users i who successfully sends the server encrypted shares with $\mathcal{U}_\mathcal{S}^3$. 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 $r_{j,i} = \mathsf{AE.dec}(c^r_{j,i}, \mathsf{ek}_{i,j})$, and $h_{j,i} = \mathsf{AE.dec}(c^h_{j,i}, \mathsf{ek}_{i,j})$. 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 $\mathsf{sk}^1_{j',i}$ for $j' \in \mathsf{offline}_{d-1}$ and $\mathsf{sk}^{-1}_{j',i}$ 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 $\mathsf{sk}^b_{i,j}$ 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 $h_{\mathcal{S}} = \sum_{i \in \mathsf{offline}_{\mathcal{S}}} h_i$.

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

Algorithm 4 SecAgg ($\mathsf{MicroFedML}_2$)

This protocol uses the following algorithms defined in Section 4.2: a public key infrastructure, a Shamir's secret sharing scheme ($\mathsf{SS.share}$, $\mathsf{SS.recon}$, SS.exponentShare, SS.exponentReco

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 $\sum_{i \in \mathcal{O}^k} x_i^k$, in which $\mathcal{O}^k$ denotes a set of at least Bt users.

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

```
for Iteration $k = 1, 2, \ldots$ do
    Round 1: Masked Input:
        User $i$: It masks the input by $X_i = x_i + r_i + h_i$ and sends $H(k)^{X_i}$ 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 $\mathcal{O}$ denote the set of users $i$ who successfully send the masked input to the server. For each group $G_d$, the server sends $\mathcal{O}_d = \mathcal{O} \cap G_d$ to all users $i \in \mathcal{O}_d$.
    Round 2: Online Set Checking:
        User $i$: It checks $|\mathcal{O}_d| \geq t$, signs $\mathcal{O}_d$ and sends back to the server as described in line 4 of Algorithm 2.
        Server $\mathcal{S}$: It distributes the signatures from users $G_d$ to $\mathcal{O}_d$ as described in line 5 of Algorithm 2.
    Round 3: Mask Reconstruction:
        User $i$: it checks that it receives at least $t$ valid signatures from the members of $G_d$ on $\mathcal{O}_d$. If any signature is invalid, abort. Then it calculates $\zeta_i = H(k)^{\sum_{j \in \mathcal{O}_d} r_{j,i} - \sum_{j \in \mathcal{U}_i \setminus \mathcal{O}_d} h_{j,i}}$ and sends $\zeta_i$ to the server.
        Server $\mathcal{S}$: If it receives $\zeta_i$ from less than $t$ users in any group $G_d$, abort. Otherwise,
    the server calculates
        $$z = \log_{H(k)} \left( H(k)^{\sum_{i\in\mathcal{O}} X_i} / \prod_{d\in[B]} \mathsf{SS.exponentRecon}(\{\zeta_i\}_{i\in\mathcal{O}_d}, t) \right)$$
    by brute force. Then it calcualtes $\sum_{i\in\mathcal{O}} x_i = z + h_{\mathcal{S}}$, in which $h_{\mathcal{S}}$ is from the server's output in the Setup phase.
end for
```

4.4.2 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 [9].

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

Definition 4.4.1 (Not too many corrupt members)**.** *Let N, t be integers such that $N < n$ and $t \in (N/2, N)$, and let $\mathcal{C} \subset [n]$. Let $\mathbf{G} = (G_1, \ldots, G_B)$ is a partition of $[n]$ so that $|G_d| = N$ for each $d \in [B]$. We define event E_1 as*

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

Definition 4.4.2 (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*

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

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 all but negligible probability.

Definition 4.4.3 (Nice Grouping). *Let N, σ, η be integers and let $\gamma, \delta \in [0,1]$. Let $\mathcal{C} \subset [n]$ and $|\mathcal{C}| \leq \gamma n$. Let $\mathcal{D}$ be a distribution over pairs $(\mathbf{G}, t)$. We say that $\mathcal{D}$ is $(\sigma, \eta, \mathcal{C})$-*nice *if, for all set $D \subset [n]$ such that $|D| \leq \delta n$, we have that*

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

Lemma 4.4.4. *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 $\mathcal{C} \subset [n]$, such that $\mathcal{C} \leq \gamma n$, be the set of corrupt clients. Then for sufficiently large n, the distribution $\mathcal{D}$ over pairs (G, t) implemented by uniformly randomly assigning all n users into n/N groups each of size N is $(\sigma, \eta, \mathcal{C})$-nice.

Proof. We first prove that constraint 1 in Definition 4.4.3 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.

□

4.4.3 Correctness

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

Theorem 14 (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 3 and Algorithm 4 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 $(\sigma, \eta, \mathcal{C})$-nice for $\mathcal{C} \subset \mathcal{U}$ with $|\mathcal{C}| < \gamma|\mathcal{U}|$.*

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 $(\sigma, \eta, \mathcal{C})$-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

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

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

4.4.4 Privacy

It is easy to see that same as MicroFedML$_1$, 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. Now we prove that this protocol guarantees privacy against malicious adversary who controls both users and the server.

Theorem 15 (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 be an instance of Algorithm 3 and Algorithm 4 guarantees privacy against γ-fraction of malicious adversary with δ offline rate with probability $1 - 2^{-\sigma}$ when the grouping is $(\sigma, \eta, \mathcal{C})$-nice. The Setup phase of the protocol runs in 5 rounds with $O(n)$ communication complexity per user and the Aggregation phase runs in 3 rounds with $O(n)$ communication complexity per user.*

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

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

Lemma 4.4.5. *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.*

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 $t - |\mathcal{C} \cap G_{d+1}|$ of which are from honest users. All of these honest users must have received at least t valid signatures on offline sets that includes user i. At least $t - |\mathcal{C} \cap G_d|$ of these signatures come from honest users in G_d who have put i in their offline list. By definition, i is not participating in the Aggregation phase. □

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

Fact 4.4.6 (Unique Common Online Set each Group)**.** *When $2t > n/B + |\mathcal{C} \cap G_d|$, there is at most one common online set $\mathcal{O}_d$ for each group d in every iteration.*

This statement can be proved with the same reasoning as the proof of Fact 4.3.1. Now, we give the proof for Theorem 15.

Proof. **(of Theorem 15)** By saying that an honest user uses r' (or h') as r-mask (or h-mask) in iteration k of the Aggregation phase, we mean that in the first round of the iteration k, the user uses r' (or h') to calculate X_i; it also calculates the shares $r'_{i,j}$ of r' for honest users in its group fixing the shares that have already been sent to its corrupt neighbors. Then in the third round, every honest user j in its group uses $r'_{i,j}$ or $h'_{i,j}$ to calculate the sum of shares.

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

- In the Setup phase:
  - **Round 1**: The simulator simulates each honest user following the protocol.
  - **Round 2**: Each honest user i receives the public keys and the signatures $(\mathsf{pk}_j, \sigma_j)$ from the server, and verifies the signatures as described in Algorithm 3, except that the simulator additionally aborts if some honest user i

receives a valid signature of an honest user j on a public encryption key different from what user j sends to the server in the previous round. Then for each corrupt user $j \in \mathcal{U}_i^1 \cap \mathcal{C}$, an honest user i stores $\mathsf{ek}_{i,j} = \mathsf{KA.agree}(\mathsf{pk}_j, \mathsf{sk}_i)$. 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 $t - n_{\mathcal{C}}$ 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 $\mathrm{j} \in G_{d-1}$) 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 $\mathcal{O}_d$ and sends the signature to the server.
  - **Round 3**: For each group G_d, the simulator checks if there are some honest users $i \in G_d$ receiving at least t valid signatures on $\mathcal{O}_d$ it receives in Round 2.
    * If there are such users in every group, then let $\mathcal{O} = \cup_{d \in [B]} \mathcal{O}_d$, the simulator queries the ideal functionality to get $w = \mathsf{Ideal}(\mathcal{O} \backslash \mathcal{C}, k)$. The simulator then uniformly randomly chooses w_i^* for $i \in \mathcal{O}$ under the restriction that $\sum_{i \in \mathcal{O} \backslash \mathcal{C}} w_i^* = w$. 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 $j \in G_d \backslash \mathcal{C}$ based on $r_{i,j}$ for $j \in G_d \cap \mathcal{C}$ that have already been sent to the corrupt users in the Setup phase. Let $r_{i,j}^* = r_{i,j}$ for $i \in \mathcal{C}$. The simulator sends ζ_i^{r*} 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 $\mathcal{O}_d$, the simulator uniformly randomly chooses the random mask r_i^* and the mutual mask h_i^* of each honest user i, and calculates the shares of r_i^* and h_i^* based on the shares that have already been sent to the corrupt users in the Setup phase. Then each honest user calculates 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 $\mathcal{C}$ in the real execution.

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

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

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

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

With the same reasoning as in the proof of Theorem 4.3.4, this hybrid is indistinguishable from the previous one.

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

This hybrid is identical to the previous one as there is atmost one unique $\mathcal{O}_d$ and if an honest user is not included in $\mathcal{O}_d$, the share $r^*_{i,j}$ for honest user i not in $\mathcal{O}$ will not be included in 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 $i \in \mathcal{O}_d \backslash \mathcal{C}$, instead of setting $r_i^* = X_i^* - x_i - h_i$, SIM randomly picks r_i^* under the constraint that $\sum_{i \in \mathcal{O}_d \backslash \mathcal{C}} r_i^* = \sum_{i \in \mathcal{O}_d \backslash \mathcal{C}} X_i^* - \sum_{i \in \mathcal{O}_d \backslash \mathcal{C}} (x_i + h_i)$.

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

Hyb8 In this hybrid, at the end of the Setup phase, the simulator uniformly randomly chooses $\mathsf{mk}_{i,j}^*$ 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 $j \in (\mathcal{C} \cup \mathsf{offline}_{\mathsf{SIM}}) \cap G_{d \pm 1}$, 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 4.4.9, 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 $\mathcal{O}_d$ exists for each $d \in [B]$, instead using the constraint $\sum_{i \in \mathcal{O}_d \backslash \mathcal{C}} r_i^* =$

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

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

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

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

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

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

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

Lemma 4.4.7. *Let $n, t, n_{\mathcal{C}}, K$ be integer parameters, $n_{\mathcal{C}} \leq n - t < t$. Let w be an element in $\mathbb{Z}_q$, and $w_i \in \mathbb{Z}_q$ for $i \in [n]$ be shares of w calculated with t-out-of-n Shamir secret sharing algorithm, i.e., $\{w_i\}_{i \in [n]} \leftarrow \mathsf{SS.share}(w, [n], t)$.*

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

The following two distributions are computationally indistinguishable:

$$w, \{w_i\}_{i \in [n-n_{\mathcal{C}}+1,n]}, \{H(k)^{w_i}\}_{i \in [n-n_{\mathcal{C}}], k \in [K]} \tag{4.5}$$

$$w, \{w_i\}_{i \in [n-n_{\mathcal{C}}+1,n]}, \{H(k)^{w_i^k}\}_{i \in [n-n_{\mathcal{C}}], k \in [K]} \tag{4.6}$$

Proof. We define a hybrid Hyb_0 to be identical to the distribution (4.5), 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, $H(k)^{w_i}_{i \in [n-n_{\mathcal{C}}]}$ are substituted with $H(k)^{w_i^k}_{i \in [n-n_{\mathcal{C}}]}$. Thus, Hyb_K is identical to distribution (4.6). 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

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

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

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

$\mathcal{D}'$ uniformly randomly picks $\{w_i\}_{i \in [n-n_{\mathcal{C}}+1,n]}$ as the second part of the input to $\mathcal{D}$, and calculates $W_i = \mathsf{SS.exponentRecon}((g^w, 0), \{B_j, j\}_{j \in [t-n_{\mathcal{C}}-1]}, \{g^{w_j}, j\}_{j \in [n-n_{\mathcal{C}}+1,n]}, t, i)$ for $i \in$

$[t-n_{\mathcal{C}}, n-n_{\mathcal{C}}]$. For $k \in [K]$ and $k \neq k_0$ it uniformly randomly picks $s_k \in \mathbb{Z}_q$, and sets $H(k) = g^{s_k}$.

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

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

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

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

When the input to $\mathcal{D}'$ is from the distribution (4.1), 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 (4.2), 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 4.3.2, such a distinguisher $\mathcal{D}'$ does not exist. Thus, we have a contradiction. □

Lemma 4.4.8. *Let $n, t, n_{\mathcal{C}}, K$ be integer parameters, $n_{\mathcal{C}} \leq n-t < t$. Let $x_1, \ldots, x_{n-n_{\mathcal{C}}}$ be uniformly random elements in $\mathbb{Z}_q$, and $\sum_{i\in[n-n_{\mathcal{C}}]} x_i = w$. Let $x_{i,j} \in \mathbb{Z}_q$ for $i \in [n-n_{\mathcal{C}}]$ and $j \in [n]$ be shares of x_i calculated with t-out-of-n Shamir's secret sharing algorithm, i.e., $\{x_{i,j}\}_{j\in[n]} \leftarrow$ $\mathsf{SS.share}(x_i, [n], t)$ for $i \in [n-n_{\mathcal{C}}]$.*

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

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

$$w, \{x_{i,j}\}_{i\in[n-n_{\mathcal{C}}],j\in[n-n_{\mathcal{C}}+1,n]}, \{\{H(k)^{x_i}\}_{i\in[n-n_{\mathcal{C}}]}, \{H(k)^{x_{i,j}}\}_{i,j\in[n-n_{\mathcal{C}}]}\}_{k\in[K]} \tag{4.7}$$

$$w, \{x_{i,j}\}_{i\in[n-n_{\mathcal{C}}],j\in[n-n_{\mathcal{C}}+1,n]}, \{\{H(k)^{y_i^k}\}_{i\in[n-n_{\mathcal{C}}]}, \{H(k)^{y_{i,j}^k}\}_{i,j\in[n-n_{\mathcal{C}}]}\}_{k\in[K]} \tag{4.8}$$

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 (4.7), 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 $i \in [n - n_{\mathcal{C}} - 1]$ as described in the Lemma. Then, instead of secret sharing $x_{n-n_{\mathcal{C}}}$, we calculates $x_{n-n_{\mathcal{C}},j} = w_j - \sum_{i\in[n-n_{\mathcal{C}}-1]} x_{i,j}$ for each $j \in [n]$.

This hybrid is identical to distribution (4.7) by the additive homomorphic property of Shamir's 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 $j \in [n - n_{\mathcal{C}}]$ such that $w = \mathsf{SS.recon}(\{w_j^k, j\}_{j\in[n-\mathcal{C}]}, \{w_j, j\}_{j\in[n-n_{\mathcal{C}}+1,n]}, t)$, and we calculates $y_{n-n_{\mathcal{C}},j}^k = w_{n-n_{\mathcal{C}}}^k - \sum_{i\in[n-n_{\mathcal{C}}-1]} x_{i,j}$. We substitutes $H(k)^{x_{n-n_{\mathcal{C}}}}$ with $H(k)^{y_{n-n_{\mathcal{C}}}^k}$ and $H(k)^{x_{n-n_{\mathcal{C}},j}}$ with $H(k)^{y_{n-n_{\mathcal{C}},j}^k}$ for $j \in [n - \mathcal{C}]$.

By Lemma 4.4.7, 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 $i \in [n - n_{\mathcal{C}} - 1]$, we choose y_i^k uniformly at random, calculates $\{y_{i,j}^k\}_{j\in[n-n_{\mathcal{C}}]}$ such that $y_i = \mathsf{SS.recon}(\{y_{i,j}^k, j\}_{j\in[n-n_{\mathcal{C}}]}, \{x_{i,j}, j\}_{j\in[n-n_{\mathcal{C}}+1,n]})$. 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 $i \in [n - \mathcal{C} - 1], j \in [n - \mathcal{C}]$.

By Lemma 4.3.3, This one is indistinguishable from the previous one. This hybrid is also identical to distribution (4.8).

□

Lemma 4.4.9. *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.

4.5 Performance Analysis

In this section, we analyze the asymptotic efficiency and simulation results of the secure aggregation protocols. We assume that the protocols run with n users and one single server, and the sum of all users' input is in a range $R \subset \mathbb{N}$ which can be represented with $\ell = O(1)$ bits without overflow, i.e., $R = 2^\ell$. As the Setup phase happens only once, we analyze the cost of the Setup phase separately.

	Communication cost		#Round
	User	Server	
MicroFedML$_1$ (Setup)	$O(n)$	$O(n^2)$	3
MicroFedML$_2$ (Setup)	$O(\log n)$	$O(n \log n)$	5

Table 4.1: Communication overhead of the Setup phase of aggregation of MicroFedML$_1$ and MicroFedML$_2$ guaranteeing privacy against malicious adversaries in which n denotes the total number of users. Note that BIK+17 and BBG+20 have $O(n^2)$ and $O(n \log n)$ communication cost, respectively, on the server side but they incur this cost in every iteration.

4.5.1 Asymptotic performance of MicroFedML$_1$

Semi-honest protocol

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

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

Computation We discuss the computation cost of each user first. In the Setup phase, each user i needs to 1) generate a pair of encryption keys pk_i and sk_i, 2) run the key exchange algorithm to obtain $\mathsf{ek}_{i,j}$ 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 $O(n)$. In the Aggregation phase, the computation cost of each user consists of calculating $H(k)^{x_i+r_i}$ and calculating $H(k)^{\sum_{j \in \mathcal{O}} r_{j,i}}$, which is $O(n)$ in total.

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

Protocol guaranteeing privacy against malicious adversary

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

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

4.5.2 Asymptotic performance of MicroFedML$_2$

Semi-honest protocol

Communication In the Setup phase, each user $i \in G_d$ needs to 1) send its public encryption key pk_i and two public masking keys pk_i^1 and pk_i^{-1} to the server and receive the public keys of the group members of its own group G_d and two neighboring groups G_{d+1} and G_{d-1}, 2) send the encrypted shares of sk_i^1 and sk_i^{-1} to all group members of G_{d+1} and G_{d-1} and receive encrypted shares from them, 3) send the encrypted shares of r_i and h_i to the group members in G_d and receive encrypted shares from them, 4) receive the list of offline users in G_{d+1} and G_{d-1} and send the shares of secret masking keys of those offline users to the server. When the size of each user group is set as $O(\log n)$, the communication cost for each user in the Setup phase is $O(R \log n)$. As messages between the server and each user is the same, the communication cost of the Setup phase for the server is $O(nR \log n)$.

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

Computation We discuss the computation cost of each user first. In the Setup phase, each user $i \in G_d$ needs to 1) generate three key pairs, 2) run key exchange algorithm to get the symmetric encryption key $\mathsf{ek}_{i,j}$ 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 $O(\log n)$

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

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

In the Aggregation phase, the server needs to reconstruct the sum of masks in the exponent, multiply the results of all groups together, and calculate the discrete log to get the final result. Assuming each group is of size $O(\log n)$, the computation cost of the server is $O(\log n + \frac{n}{\log n} + 2^\ell)$

Protocol guaranteeing privacy against malicious adversary

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

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

4.5.3 Implementation and concrete performance

To measure the concrete performance, we implement prototypes of both of our protocols, MicroFedML$_1$ and MicroFedML$_2$, as well as two benchmark protocols, BIK+17 and BBG+20 with ABIDES [12], a discrete event simulation framework with modification to enable simulation of federated learning protocol, in Python language. In all implementations, we assume semi-honest setting, thus we omit the marked parts of the protocols that only needed in the malicious setting.

We use the following cryptographic primitives:

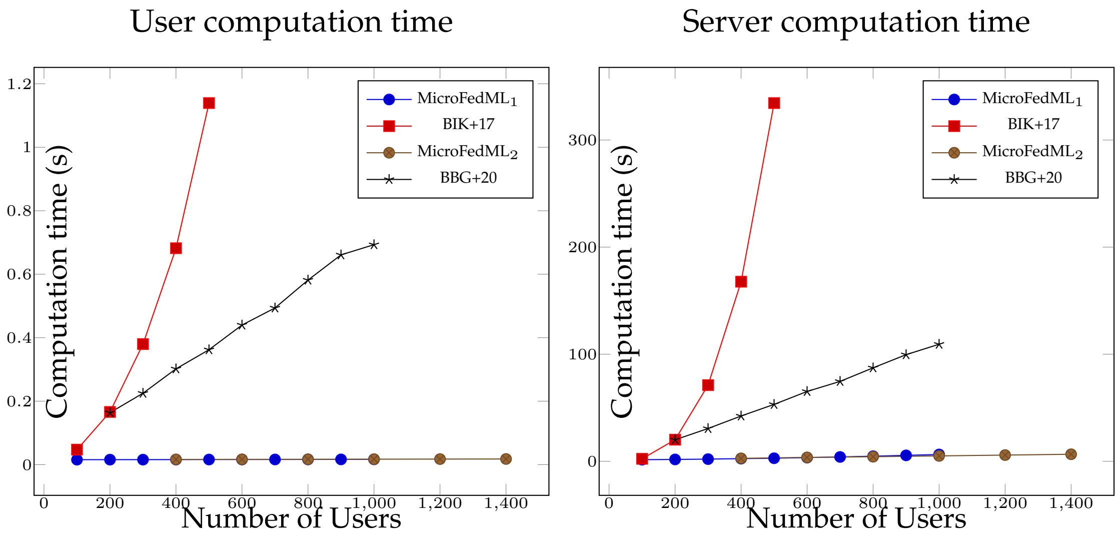
- For the finite field used in secret sharing, exponentiation and discrete log, We use a 2048-bit secure prime provided in `https://www.ietf.org/rfc/rfc3526.txt` as p.

- For key agreement, we use Diffie-Hellman key exchange algorithm and SHA-256 hash function provided by PyNaCl library.
- For secret sharing, we use the standard t-out-of-n Shamir's secret sharing extended with reconstruction in exponents as described in Section 4.2.
- For discrete log, we use the brute force algorithm, i.e., searching for the log result starting from 0.
- For Authenticated Encryption, we use the secret key encryption algorithm XSalsa20 https://libsodium.gitbook.io/doc/advanced/stream_ciphers/xsalsa20 provided by PyNaCl library with 256-bit keys.

We observe that the discrete log calculation is the most expensive part of both our protocols. This part can be optimized with several known discrete log algorithms with better efficiency, which improves the asymptotic running time from $O(n)$ to $O(\sqrt{n})$, in which n denotes the size of the range of the result.

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

In Figure 4.3, we compare the local computation time of four protocols with the length of result fixed to 20 bits and group/neighbor size fixed to 200 for MicroFedML$_2$ and BBG+20. As shown in the graph, the computation time of MicroFedML$_1$ is about 100


User computation time

Server computation time

Figure 4.3: Wall-clock local computation time of one iteration of the Aggregation phase of a user and the server as the number of user increases. The length of the sum of inputs is fixed to $\ell = 20$ bits in different lines, i.e., the input of each user is in the range $[2^{\ell}/n]$ when the total number of users is n. For the protocol MicroFedML$_2$ and BBG+20, the group size / neighbor size is set to 200.

times shorter than BBG+20 when the total number of users is 500, and the computation time of MicroFedML$_2$ is about 20 times faster than BBG+20 when the total number of user is 1000. Since the lines of MicroFedML$_1$ and MicroFedML$_2$ are overlapped in the chosen scale, we include zoom-in graphs in Figure 4.5 and Figure 4.6. In Figure 4.4 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 fixed to 100. The size of outgoing messages of each user of MicroFedML$_1$ and MicroFedML$_2$ are almost the same, which is about 1000 times smaller than BIK+17 and about 200 times smaller than BBG+20 when the total number of users is 500. The size of incoming messages from the server per user of MicroFedML$_1$ is also almost the same as MicroFedML$_2$, 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.

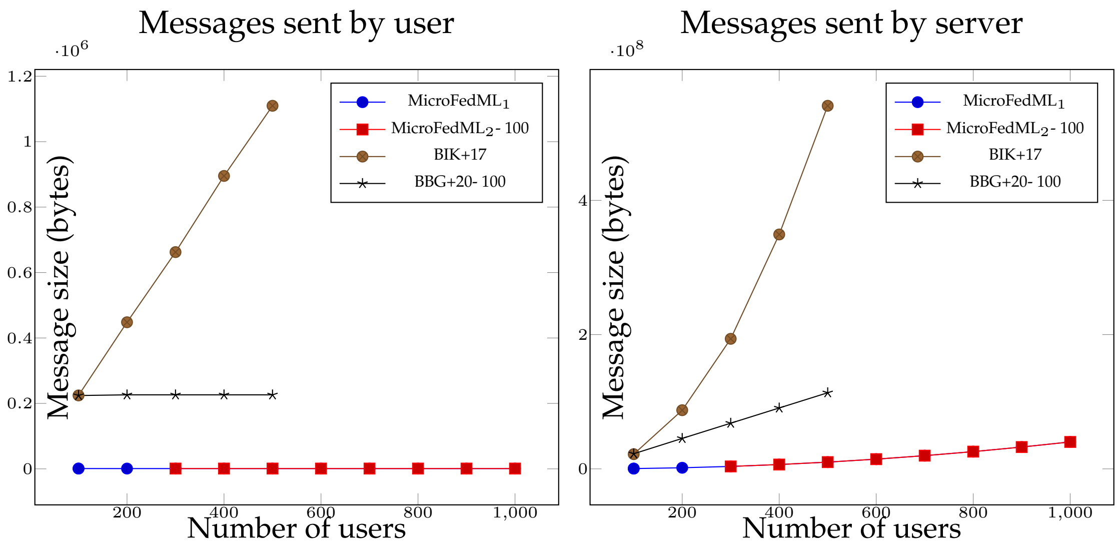


Figure 4.4: Outbound bandwidth cost (bytes) on the user and the server side of each iteration of different protocols when the total number of users grows. The size of the result is fixed to 20 bits. The neighbor size in protocol MicroFedML$_2$ and BBG+20 is fixed to 100.

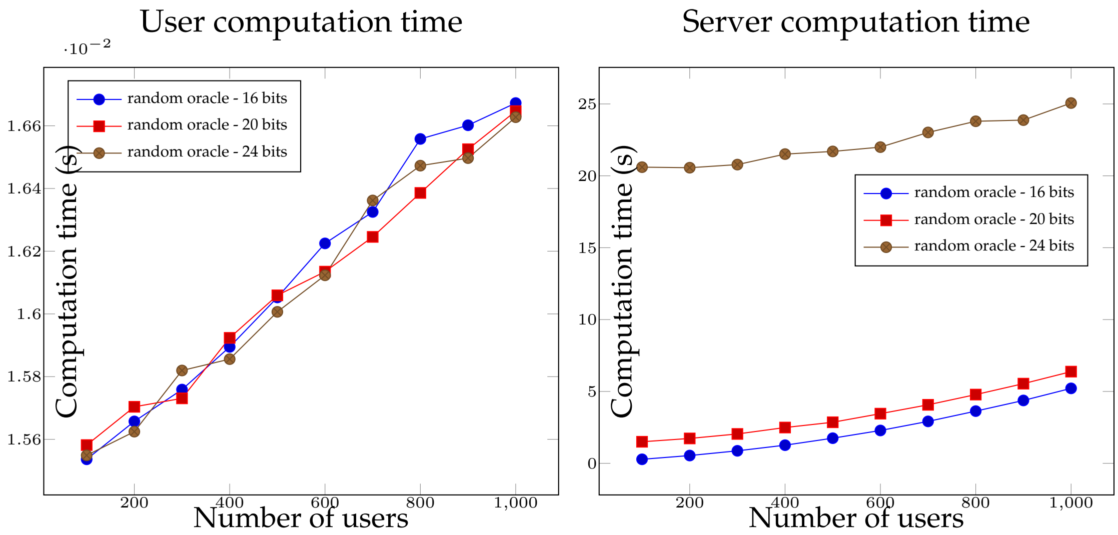


Figure 4.5: Zoom-in results of Figure 4.7 for the MicroFedML$_1$ protocol. The lines show wall-clock computation time of one iteration of the Aggregation phase of a user and the server, as the number of users increases. The length of the sum of inputs varies in different lines as shown in the legend.

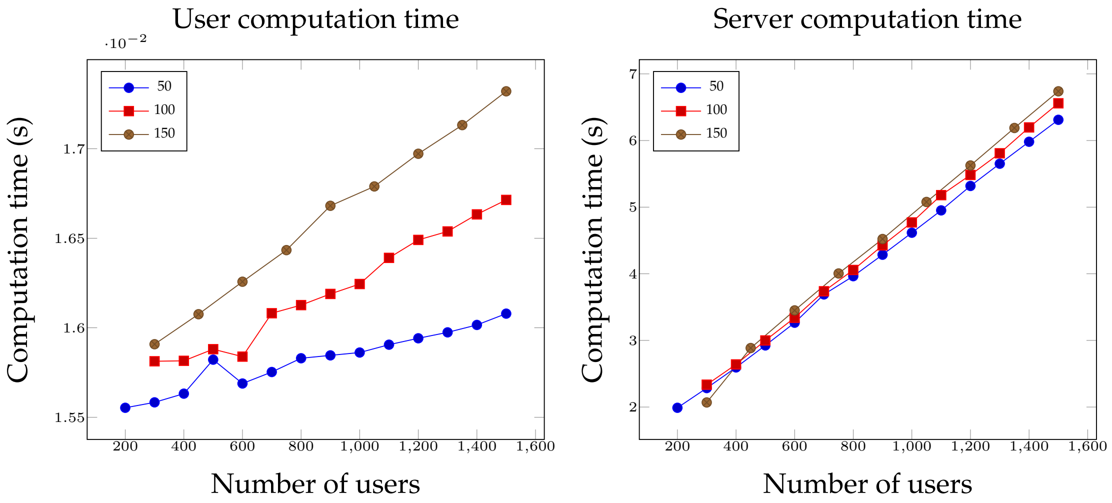


Figure 4.6: Wall-clock local computation time of MicroFedML$_2$ for a user and the server with different group size (as shown in the legend), as the total number of users increases. The maximum size of the aggregation result is fixed to 20 bits.

Figure 4.7 shows how the local computation time of each iteration of the Aggregation phase (y-axis) of each user and the server changes as the total number of users (x-axis) grows. Different lines show the computation time when the size of the sum of inputs are different. As shown in the graph, the running time of MicroFedML$_1$ is not affected significantly by the number of users but more affected by the size of the sum of inputs. On the contrary, the performance of BIK+17 is impacted by the total number of users and does not 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 keep the same in these scenarios. On the other hand, each user needs to share two secrets among 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 increase. On the contrary, in the Aggregation phase of MicroFedML$_1$, each user only calculates and sends one field element to the server in each round and the server

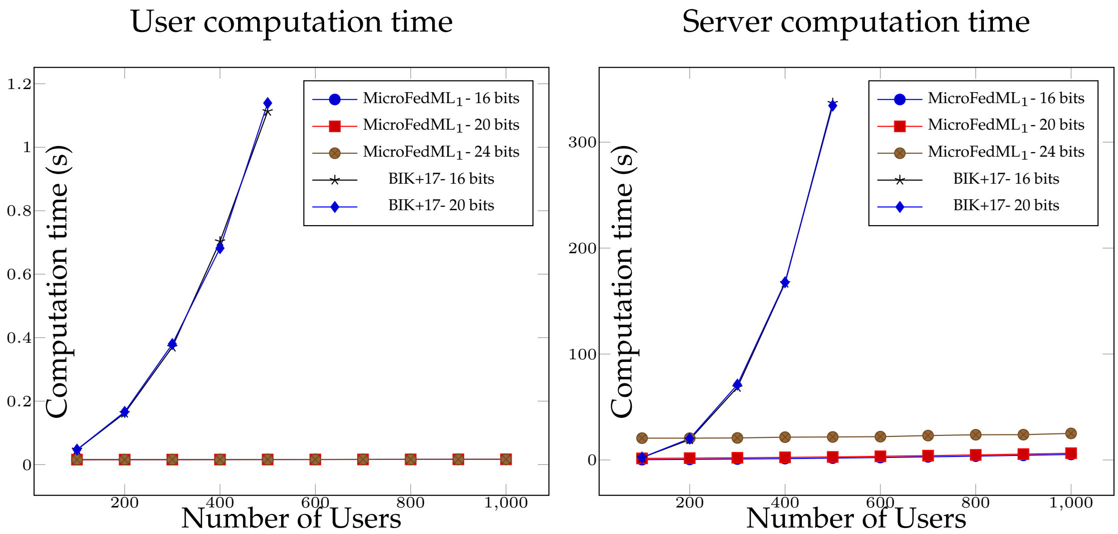


Figure 4.7: Wall-clock local computation time of one iteration of the Aggregation phase of a user and the server as the number of user increases. The length of the sum of inputs is set to $\ell = 16, 20, 24$ bits in different lines, i.e., the input of each user is in the range $[2^\ell/n]$ when the total number of users is n.

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. We show that the running time of users and server of our protocol does increase as total number of users increase in Figure 4.5 (which is a zoom-in of Figure 4.7 containing only running results of MicroFedML$_1$.)

In Figure 4.8 we show the local computation time of online users (who stay online in the whole iteration) and the server in one iteration of MicroFedML$_1$ and BIK+17 with 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 sending the masked inputs to the server (which happens in the first round of MicroFedML$_1$ and in the second round of BIK+17) then stay silent in the rest part of the iteration. The size of the sum of inputs is fixed to

20 bits. Different lines show the computation time of one iteration of the online users and the server in the cases with different δ. As shown in the left graph, the dropout rate does not affect the computation time of online users significantly in BIK+17. This is because in the round after the dropout event happens, each online user i needs to send one share of secret for each other user j to the server no matter user j is online or not. The graph on the right side shows that the computation time of the server in BBG+20 decreases as the fraction of the dropout users increases. 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, while in this work we assume each input is a single element and the server does not apply PRG over the symmetric masking key, which makes the impact of dropout rate less severe. Moreover, the implementation of the reconstruction of the Shamir's secret sharing in our experiment naively uses all shares received, which means the more users drop out, the less shares received by the server in the next round and the less time it takes to run the reconstruction algorithm.

In both graphs of Figure 4.8, we do not see significant change in computation time as the dropout fraction changes. We provide a zoom-in version including only MicroFedML$_1$ in Figure 4.9, in which we can see the higher value of δ leads to shorter computation time for both users and the server. This is because of the same reason as mentioned above — the less users are online, the less shares need to be included when computing the sum of shares on the user's side, and the less shares are included in the reconstruction in the exponent on the server's side.

Figure 4.10 shows the local computation time of each user and the server of each iteration of MicroFedML$_2$ 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

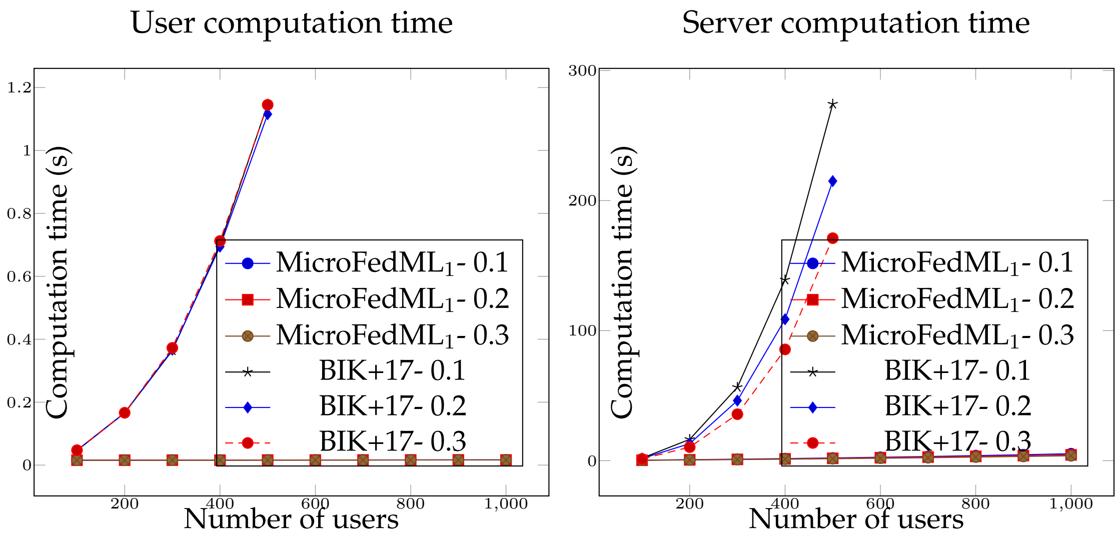


Figure 4.8: Wall-clock computation time of one iteration of the Aggregation phase of a user and the server, as the number of users increases. Different lines show the running time when the fraction of offline users are different. The length of the sum of inputs is fixed to 16 bits. In other words, the input of each client is in the range $[2^{16}/n]$ when the total number of users is n.

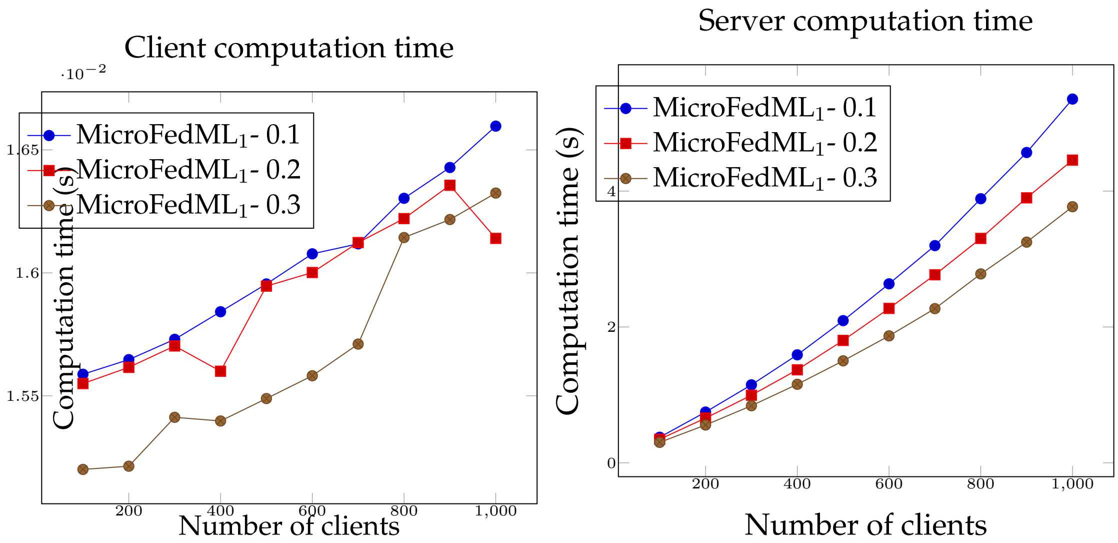


Figure 4.9: Zoom-in of Figure 4.8 with only MicroFedML$_1$ included. Wall-clock running time for a user and the server, as the number of users increases. Different lines show the running time when the fraction of offline users are different.

of neighbors each user has in BBG+20. Note that in real world application, 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 user 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 Appendix 4.4.2 and Section 3.5 of [9] 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 purpose. As shown in the running result, sharing information with only a small set of neighbors significantly improves the performance of the benchmark protocol BBG+20, as the number of users included in secret sharing and reconstruction is a major factor of computation overhead. On the contrary, the improvement brought by the grouping is not that obvious in $\mathrm{MicroFedML}_2$ as the only two things affected by the number of neighbors in the Aggregation phase are the size of the online set the server sends to each user and the number of shares the server uses in the reconstruction of secret in exponent. Both of these two parts compose only a very small fraction of total running time. We also present Figure 4.6 as a zoom-in which only includes our grouping protocol to show how the group size affects the computation 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, while the major computation cost on the server side is the discrete log, which is not affected by the total number of users.

In Figure 4.11, we report the total computation time of each user and the server in the Setup phase of $\mathrm{MicroFedML}_1$ and $\mathrm{MicroFedML}_2$ with different group sizes. The graph on

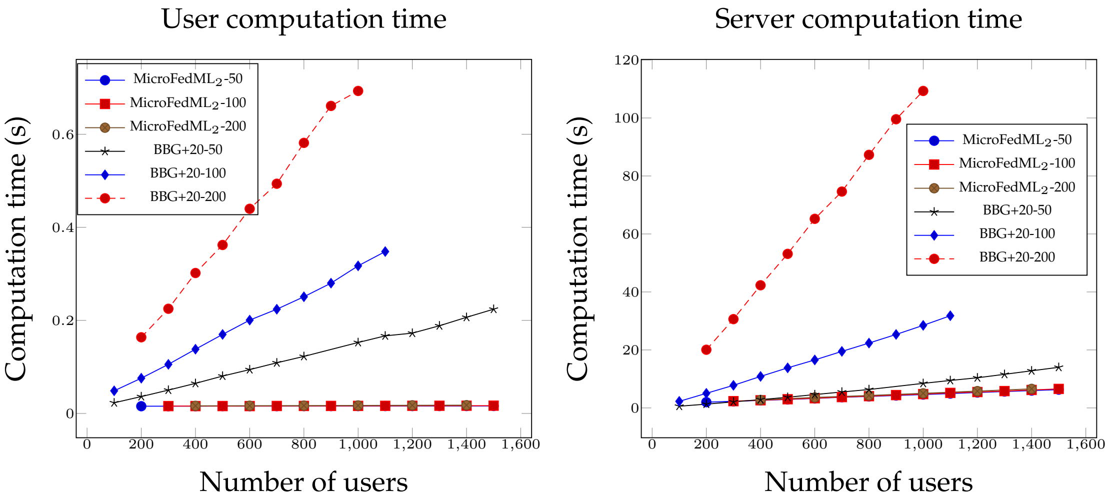


Figure 4.10: Wall-clock local computation time (y-axis) of one iteration of MicroFedML$_2$ and BBG+20 for a user (left) and the server (right) as the number of users (x-axis) increases. Different lines shows the running result with different group size (shown in the legend). The length of the sum of all inputs is fixed to 20 bits.

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. This is because in both protocols, the computation time grows when each user needs to share secrets with more parties.

In Figure 4.12, we report the bandwidth cost of the Setup phase of both MicroFedML$_1$ and MicroFedML$_2$. In MicroFedML$_1$, the bandwidth cost on the user side grows linearly with the growth of total number of users as each user needs to send one encrypted share to every other user, which also results in quadratic bandwidth growth on the server's side. In MicroFedML$_2$, 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.

We also implement federated learning protocol with MicroFedML$_1$ on the adult census income dataset [21] 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. [13] which is a cleaned version with one constant intercept feature added based on a preprocessed version of the dataset from Jayaraman et al. [31] The preprocessed dataset contains 105 features and 45,222 records, about 25% (11,208) of which are positive. The dataset is loaded once at the beginning of the protocol execution and randomly split into training set (75%) and testing set (25%). At the beginning of each training iteration, a user randomly selects 200 records from the training data as its local training data and test the model accuracy with the common test set. We run both the plain federated learning in which every user simply sends its update in plain text to the server and the version with MicroFedML$_1$ in which the model updates are aggregated with the secure aggregation protocol for 5 iterations of aggregation, each with 50 local training iteration. 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.

4.5.4 Discussion about the size of the inputs

As we use discrete log algorithm to recover the result in every iteration which is known to lack efficient solution for large values, the size of the result (i.e., the aggregated value generated in each iteration) significantly affects the server-side calculation time. We are using THE brute force algorithm to calculate the discrete log, which takes $O(R)$ time

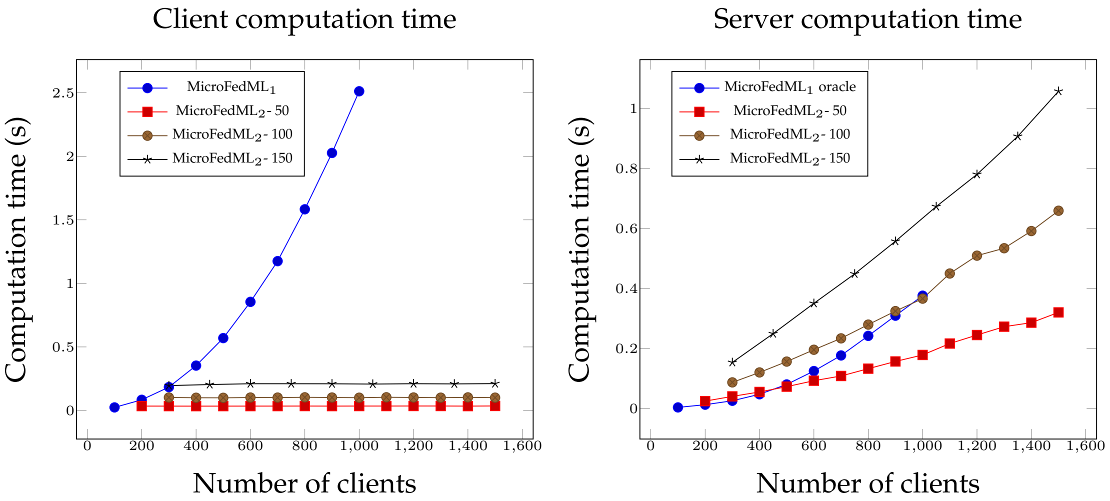


Figure 4.11: Wall-clock local computation time of the **Setup phase** for a user and the server, as the number of users increases. The length of the sum of inputs is fixed to 20 bits. Different lines shows the running results for MicroFedML$_1$ and MicroFedML$_2$ with different group size.

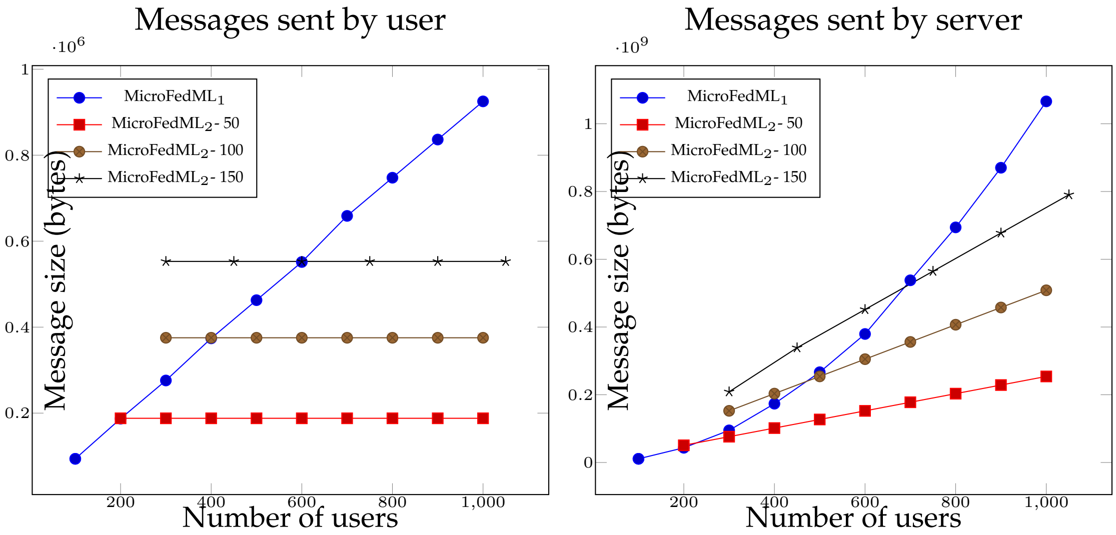


Figure 4.12: Outbound bandwidth cost (bytes) on the user and the server side of **the Setup phase** of MicroFedML$_1$ and MicroFedML$_2$ when the total number of users grows. The size of the result is fixed to 20 bits. Different lines shows the running results for MicroFedML$_1$ and MicroFedML$_2$ with different group size.

in which R denotes the size of the range of the result. There are several well-known algorithms [45, 49] which improves the amortized efficiency to $O(\sqrt{R})$. We are not using them in our experiments as they are not outperforming the brute force algorithm on the small domain size used in our experiments. We believe that the optimized algorithms can bring performance improvements on larger domains. We determine the domain of the inputs based on the size of the aggregation result and the number of users. Let R denote the size of the range of the result and n denote the total number of users, then the input of each user in each iteration is randomly sampled from the domain $[R/n]$. That said, our aggregation protocols are more suitable to models with small weights coming out of quantization, compression etc. See Section **??** for references.

In all of our experiments, we are using 2048 bits standard prime for the finite field, which means the order of the cyclic subgroup and the elements in the group are of the same size. The computation time can also change with other choices of the prime. Moreover, we only consider the case with single input in our experiments. When applying our protocol to the case with vectors as input, each vector element can be treated as an individual input, thus the communication cost grows linearly with the length of input, and the discrete log computation on each vector element can be executed in parallel. In comparison, BIK+17 and BBG+20 extends the shared randomness with pseudorandom generator (PRG) to cover the length of the input vector, thus the communication cost does not grow with the length of the input vector, but the computation time grows linearly as the operation of extending the randomness with PRG cannot be parallelized.

4.6 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, and show that our concrete performance is better than the previous works when the input domain is small. A future direction is to extend the result to larger input domain for other use cases.

4.7 ABIDES Framework

We use a discrete event simulation framework ABIDES [12] to implement the simulation of our protocol. In this section, we give an overview of this framework.

4.7.1 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.

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.

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.

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’].

4.7.2 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.

4.7.3 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.

BIBLIOGRAPHY

[1] Gmail and google drive are experiencing issues, and naturally people are complaining about it on twitter. https://www.huffingtonpost.com/entry/gmail-issue_n_3099988.

[2] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Efficient synchronous byzantine consensus. In *Financial Crypto*, 2019.

[3] Alham Fikri Aji and Kenneth Heafield. Sparse Communication for Distributed Gradient Descent. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 440–445, 2017. arXiv: 1704.05021.

[4] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. page 12.

[5] Mohammad Mohammadi Amiri and Deniz Gunduz. Federated Learning over Wireless Fading Channels. *arXiv:1907.09769 [cs, math]*, February 2020. arXiv: 1907.09769.

[6] Mohammad Mohammadi Amiri and Deniz Gunduz. Machine Learning at the Wireless Edge: Distributed Stochastic Gradient Descent Over-the-Air. *IEEE Transactions on Signal Processing*, 68:2155–2169, 2020. arXiv: 1901.00844.

[7] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Secure mpc: Laziness leads to god. Cryptology ePrint Archive, Report 2018/580, 2018.

[8] Debraj Basu, Deepesh Data, Can Karakus, and Suhas N. Diggavi. Qsparse-Local-SGD: Distributed SGD With Quantization, Sparsification, and Local Computations. *IEEE Journal on Selected Areas in Information Theory*, 1(1):217–226, May 2020.

[9] James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1253–1269, 2020.

[10] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, 1988.

[11] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1175–1191. ACM, 2017.

[12] David Byrd, Maria Hybinette, and Tucker Hybinette Balch. ABIDES: towards high-fidelity market simulation for AI research. *CoRR*, abs/1904.12066, 2019.

[13] David Byrd, Vaikkunth Mugunthan, Antigoni Polychroniadou, and Tucker Hybinette Balch. Collusion resistant federated learning with oblivious distributed differential privacy. *arXiv preprint arXiv:2202.09897*, 2022.

[14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, pages 524–541, 2001.

[15] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, pages 42–51, 1993.

[16] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

[17] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. AdaComp: Adaptive Residual Gradient Compression for Data-Parallel Distributed Training. page 9.

[18] Mingzhe Chen, Nir Shlezinger, H. Vincent Poor, Yonina C. Eldar, and Shuguang Cui. Communication-efficient federated learning. *Proceedings of the National Academy of Sciences*, 118(17):e2024789118, April 2021.

[19] Laizhong Cui, Xiaoxin Su, Yipeng Zhou, and Yi Pan. Slashing Communication Traffic in Federated Learning by Transmitting Clustered Model Updates. *IEEE Journal on Selected Areas in Communications*, pages 1–1, 2021.

[20] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *Siam Journal on Computing - SIAMCOMP*, 12(4):656–666, 1983.

[21] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[22] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

[23] Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. In *SIAM Journal of Computing*, 1997.

[24] Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Helen Möllering, Thien Duc Nguyen, Phillip Rieger, Ahmad-Reza Sadeghi, Thomas Schneider, Hossein Yalame, et al. Safelearn: Secure aggregation for private federated learning. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 56–62. IEEE, 2021.

[25] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[26] S. Dov Gordon, Feng-Hao Liu, and Elaine Shi. Constant-round MPC with fairness and guarantee of output delivery. In *CRYPTO*, pages 63–82, 2015.

[27] Jens Groth and Rafail Ostrovsky. Cryptography in the multi-string model. In *CRYPTO*, 2007.

[28] Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. Online full version of this paper, `https://eprint.iacr.org/2019/179.pdf`.

[29] Farzin Haddadpour, Mohammad Mahdi Kamani, Mehrdad Mahdavi, and Viveck R Cadambe. Trading Redundancy for Communication: Speeding up Distributed SGD for Non-convex Optimization. *convex Optimization*, page 10.

[30] Samuel Horvath, Chen-Yu Ho, Ludovit Horvath, Atal Narayan Sahu, Marco Canini, and Peter Richtarik. Natural Compression for Distributed Deep Learning. *arXiv:1905.10988 [cs, math, stat]*, February 2020. arXiv: 1905.10988.

[31] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. Distributed learning without distress: Privacy-preserving empirical risk minimization. *Advances in Neural Information Processing Systems*, 31, 2018.

[32] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *J. Comput. Syst. Sci.*, 75(2):91–112, February 2009.

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

[34] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[35] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[36] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J. Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. *arXiv:1712.01887 [cs, stat]*, June 2020. arXiv: 1712.01887.

[37] Yang Liu, Zhuo Ma, Ximeng Liu, Siqi Ma, Surya Nepal, Robert H Deng, and Kui Ren. Boosting privately: Federated extreme gradient boosting for mobile crowdsensing. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1–11. IEEE, 2020.

[38] Amirhossein Malekijoo, Mohammad Javad Fadaeieslam, Hanieh Malekijou, Morteza Homayounfar, Farshid Alizadeh-Shabdiz, and Reza Rawassizadeh. FEDZIP: A Compression Framework for Communication-Efficient Federated Learning. *arXiv:2102.01593 [cs]*, February 2021. arXiv: 2102.01593.

[39] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[40] Silvio Micali and Vinod Vaikuntanathan. Optimal and player-replaceable consensus with an honest majority. MIT CSAIL Technical Report, 2017-004, 2017.

[41] Milad Nasr, Reza Shokri, and Amir Houmansadr. 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)*, pages 739–753. IEEE, 2019.

[42] Milad Khademi Nori, Sangseok Yun, and Il-Min Kim. Fast Federated Learning by

Balancing Communication Trade-Offs. *IEEE Transactions on Communications*, pages 1–1, 2021.

[43] Rafael Pass and Elaine Shi. The sleepy model of consensus. In *Asiacrypt*, 2017.

[44] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Eurocrypt*, 2018.

[45] John M Pollard. Monte carlo methods for index computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978.

[46] Felix Sattler, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Sparse Binary Compression: Towards Distributed Deep Learning with minimal Communication. *arXiv:1805.08768 [cs, stat]*, May 2018. arXiv: 1805.08768.

[47] Felix Sattler, Simon Wiedemann, Klaus-Robert Muller, and Wojciech Samek. Robust and Communication-Efficient Federated Learning From Non-i.i.d. Data. *IEEE Transactions on Neural Networks and Learning Systems*, 31(9):3400–3413, September 2020.

[48] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[49] Daniel Shanks. Class number, a theory of factorization, and genera. In *Proc. of Symp. Math. Soc., 1971*, volume 20, pages 41–440, 1971.

[50] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM workshop on artificial intelligence and security*, pages 1–11, 2019.

[51] Twan Van Laarhoven. L2 regularization versus batch and weight normalization. *arXiv preprint arXiv:1706.05350*, 2017.

[52] Hongyi Wang, Scott Sievert, Zachary Charles, Shengchao Liu, Stephen Wright, and Dimitris Papailiopoulos. ATOMO: Communication-efficient Learning via Atomic Sparsification. *arXiv:1806.04090 [cs, stat]*, November 2018. arXiv: 1806.04090.

[53] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. *arXiv:1705.07878 [cs]*, December 2017. arXiv: 1705.07878.

[54] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. Error Compensated Quantized SGD and its Applications to Large-scale Distributed Optimization. page 9.

[55] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 13–23, 2019.

[56] Chien-Sheng Yang, Jinhyun So, Chaoyang He, Songze Li, Qian Yu, and Salman Avestimehr. Lightsecagg: Rethinking secure aggregation in federated learning. *arXiv preprint arXiv:2109.14236*, 2021.

[57] Haibo Yang, Jia Liu, and Elizabeth S. Bentley. CFedAvg: Achieving Efficient Communication and Fast Convergence in Non-IID Federated Learning. *arXiv:2106.07155 [cs]*, June 2021. arXiv: 2106.07155.