Fast Secure Federated Learning against Semi-honest and Dishonest Adversaries

Submitted to
Graduate School of Information Science and Technology
Osaka University

July 2024

Hiroki MASUDA

List of Publications

Journal Papers

1. Hiroki Masuda, Kentaro Kita, Yuki Koizumi, Junji Takemasa and Toru Hasegawa, "Byzantine-Resilient Secure Federated Learning on Low-Bandwidth Networks," *IEEE Access*, vol.11, pp. 51754–51766, May 2023.

Refereed Conference Papers

1. Hiroki Masuda, Kentaro Kita, Yuki Koizumi, Junji Takemasa and Toru Hasegawa, "Model Fragmentation, Shuffle and Aggregation to Mitigate Model Inversion in Federated Learning," in *Proceedings of IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pp. 1–6, July 2021.

Non-Refereed Technical Papers

1. Hiroki Masuda, Kentaro Kita, Yuki Koizumi, Junji Takemasa and Toru Hasegawa,'A study on designing learning protocol to protect privacy of training data in federated learning ," in *IEICE Communication Society Conference*, B-6-29, Sep. 20213 (in Japanese).
2. Hiroki Masuda, Kentaro Kita, Yuki Koizumi, Junji Takemasa and Toru Hasegawa,'A Study on Mitigation of Model Leakage by Model Fragmentation, Shuffle, and Aggregation for Federated Learning," in *Computer Security Symposium, Information Processing Society of Japan.*, pp. 260–267, 2023 (in Japanese).
3. Hiroki Masuda, Kentaro Kita, Yuki Koizumi, Junji Takemasa and Toru Hasegawa,'A study on Designing Byzantine-Resilient Secure Federated Learning on a Unicast-Based Distributed System," in *Computer Security Group, Information Processing Society of Japan.*, no.15, pp.

1–8, Feb. 2023 (in Japanese).

Preface

In machine learning systems, which are widely used in various fields, there is growing interest in utilizing training data collected from multiple edge devices (called users), such as smartphones and IoT devices. Since the training data collected by multiple users is vast and diverse, there is potential for providing high-quality machine learning services. However, privacy leakage hinders the widespread adoption of these services. For example, if training data contains sensitive information like medical information or location information, users do not wish to share such information. With the increasing momentum for regulations such as the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA), which restrict the handling of sensitive data, more users are concerned about privacy leakage.

To address the above issue, privacy-preserving machine learning systems have been proposed. Among these systems, the author focuses on *federated learning*, which achieves high model accuracy and low overhead simultaneously. Federated learning protects the privacy of training data by users sharing machine learning models that are trained locally on their training data, rather than training data. In federated learning, each user downloads a machine learning model (called a global model) from a node that maintains the global model (called a server). The user then trains the global model with their local data and sends back the trained model (called the local model). The server aggregates (adds) the users' local models and updates the global model using the aggregation result. Since federated learning does not add noise to the data or the model, it ensures high model accuracy. In addition, by not using heavy encryption operations for model training, it achieves low overhead.

Although the promise of federated learning has been acknowledged, federated learning is vulnerable to semi-honest adversaries and dishonest adversaries. Semi-honest adversaries include a semi-honest server and semi-honest users who perform privacy attacks to infer users' training data from the collected local models. Dishonest adversaries include not only semi-honest adversaries but also dishonest users (Byzantine users) who perform security attacks (Byzantine attacks) to degrade the accuracy of the global model by sending contaminated information pieces such as contaminated local

models (called contaminated models). Depending on the scenario, either semi-honest adversaries or dishonest adversaries are assumed.

To mitigate the attacks against the two types of adversaries, countermeasures have been proposed. To mitigate the attacks against semi-honest adversaries, methods called *secure aggregation* have been proposed, allowing the aggregation of users' local models while hiding (masking) individual local models. To mitigate the attacks against dishonest adversaries, methods called *Byzantine-resilient secure aggregation* have been proposed. These methods detect and remove such contaminated information pieces while performing secure aggregation. However, the seminal works on secure aggregation and Byzantine-resilient secure aggregation are heavy in terms of computation and communication.

The goal of this thesis is to make federated learning more accessible as a privacy-preserving machine learning system by improving performance while maintaining security. Through the analysis of existing secure aggregation protocols and Byzantine-resilient secure aggregation protocols, the author identifies the number of transfers of data equivalent in size to a model as a communication bottleneck, and intensive computations involving data equivalent in size to a model at a single node as a computation bottleneck. The authors' strategy for alleviating these bottlenecks is to reduce the number of transfers of model-sized data and optimize intensive computations involving model-sized data at a single node at the algorithm level.

In the first half of this thesis, the author designs a fast secure aggregation protocol for federated learning that mitigates the attacks by semi-honest adversaries. Existing secure aggregation protocols sacrifice either computation cost or communication cost for user dropout tolerance. A naive secure aggregation protocol called SecAgg achieves a small communication cost by secretly sharing random seeds instead of random masks (called shares) for local model masking. However, it necessitates that a server incurs a substantial computation cost to reconstruct the random masks from the dropout users' random seeds. To avoid such a reconstruction, a state-of-the-art secure aggregation protocol called LightSecAgg secretly shares random masks themselves. Although this approach avoids the computation cost of mask reconstruction, it incurs a large communication cost due to secretly sharing random masks. In summary, no secure aggregation protocol achieves a good balance between computation and communication costs. This thesis designs a secure aggregation protocol to achieve a good balance by complementing both types of secure aggregation protocols with each other. In the author's experiments, the author's protocol achieves up to $11.41\times$ faster while achieving the same level of privacy preservation and dropout tolerance as the existing secure aggregation protocols.

In the second half of this thesis, the author designs a fast Byzantine-resilient secure aggregation

protocol for federated learning that mitigates the attacks against dishonest adversaries. An existing Byzantine-resilient secure aggregation protocol called BREA incurs a significant communication cost due to the verification of shares generated from local models, aimed at mitigating Byzantine attacks. This thesis designs a communication-efficient share verification method for BREA to offload some parts of the share verification process from users to the semi-honest server, which avoids broadcasting large-size commitments to shares. In addition, to mitigate the increase in computation time due to computations offloaded to the server, the author's method makes the verification algorithm running on the server efficient and executes the server and user computations in parallel. In the author's experiments, the author's protocol provides a speedup of up to 15× on low-bandwidth networks like mobile networks. the author's protocol also preserves BREA's resilience against both privacy and Byzantine attacks.

Acknowledgments

This thesis could not have been completed without the support of many people. I express my gratitude here.

Firstly, I would like to express my greatest appreciation to my supervisor, Professor Toru Hasegawa. I am deeply grateful for his long-standing mentorship in research discussions and paper writing. In addition to the many valuable pieces of advice I received from him, his consistent commitment to thoroughly understanding every aspect of things has provided me with a solid foundation for my own research.

I would like to express my gratitude to the members of my committee, Professor Masayuki Murata, Professor Takashi Watanabe, Professor Hirozumi Yamaguchi, and Professor Hideyuki Shimonishi, for their thorough review of my thesis and their valuable comments. I am also grateful to Associate Professor Yuki Koizumi and Assistant Professor Junji Takemasa, my other mentors. Their deep insights have provided me with valuable knowledge.

In addition to my mentors, my research life was supported by all the members of the Information Sharing Platform Laboratory. In particular, I am deeply grateful to Nozomi Oda and Rie Maeda for their support in my daily life, and to Kentaro Kita, Yoji Yamamoto, Yasunaga Murai, Ryosuke Sasanuma, Ryo Koyama, Shimin Jing, and Yutaro Yoshinaka for enjoyable and valuable discussions. Among them, I am especially thankful to Kentaro Kita for a wealth of research advice. Through my interactions with them, I have developed diverse and flexible thinking.

I was also supported by my friends in important aspects outside of my research. Among them, I am especially grateful to Tsukasa Tajima, Kaho Fujikura, and Chihiro Harada for their active interaction and support.

Finally, I am deeply grateful to my family for their dedicated support throughout my life. Without the support from my father, mother, older brother, little brother, grandfather, and grandmother, my research activities would never have been possible. I am thankful that they have allowed me to lead the life I desired.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning systems are being applied across a wide range of fields, such as medical and industrial fields [1]. The main reason for this is that the training data collected by edge devices (called users) like smartphones and IoT devices in these systems is diverse and abundant, allowing machine learning models (called models) that learn from this data to provide good services for people. However, data privacy become a barrier to the proliferation of these services. If the training data collected by users includes sensitive information, such as medical records or vehicle location data, users may be reluctant to provide this training data due to the risk of privacy leakage. Furthermore, there is a growing trend to restrict the handling of sensitive information through regulations, such as the General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA). Protecting the privacy of training data is a crucial ethical requirement [2].

To address the above privacy issue, many privacy-preserving centralized machine learning methods have been proposed where the server trains a model in a way that the server cannot see the user's training data. These methods include methods based on data processing [3–5], Secure Multi-Party Computation (MPC) [6, 7], and Trusted Execution Environment (TEE) [8–10]. The advantage of these methods is that when the number of training data provided by each user is small, the computation overhead for each user is low. However, it is hard for these methods to simultaneously achieve the server's low computation overhead and high model accuracy.

In the data processing-based methods, each user adds noise to its training data or anonymizes it before sending it to the server, and the server trains the model with noise-applied data or anonymized data. Although this approach prevents the server from seeing the users' original training data, the noise-added or anonymized data degrades the model's accuracy. In the methods based on Secure Multi-Party Computation (MPC), each user encrypts its training data before sending it to the server,

and the server trains the model using the encrypted data. This prevents the server from seeing the training data in plaintext, but the computation involved in training on encrypted data incurs heavy computation overhead. In the methods based on TEE, such as those provided by secure chips like Intel SGX [11–13], each user encrypts its training data before sending it to the server. The server then stores and processes the training data within a secure memory space in plaintext, such as an enclave, and trains the model using it. Although the (semi-honest) server cannot see the training data in plaintext stored in the secure memory, the data in the secure memory cannot utilize devices typically used for accelerating training, such as GPUs. Even worse, if the server is dishonest, it may obtain the contents of the secure memory through cache attacks or side-channel attacks [14–16].

In contrast, federated learning [17,18], which is a privacy-preserving distributed machine learning system, achieves the aforementioned goals, without revealing users' training data. In this system, a semi-honest server and users jointly train a model (called global model) with users' training data through iterative processes. At each iteration, each user trains the global model with its training data and sends a resulting local model (called local model) to the server. The server aggregates (adds) the received local models to the new global model.

However, federated learning is vulnerable to semi-honest adversaries and dishonest adversaries. Semi-honest adversaries include a semi-honest server and semi-honest users who perform privacy attacks [19–22]. In these attacks, the semi-honest server and semi-honest users, which receive individual users' local models, infer the training data from these local models. For example, Wang et al. [20] demonstrated a privacy attack called a model inversion attack where the server uses a state-of-the-art Generative Adversarial Network (GAN) [23], which generates fake data indistinguishable from real data (i.e., training data). The key idea behind the attack is that the local model of a user strongly reflects the training data of the user, and hence a GAN can generate fake data similar to the user's training data. In contrast, dishonest adversaries include not only semi-honest adversaries but also dishonest users (Byzantine users) who perform security attacks (Byzantine attacks) [24–26]. In these attacks, the Byzantine users who send arbitrary messages modify the aggregate result, i.e., the sum of local models, thereby degrading the accuracy of the model. Byzantine users inject contaminated information pieces, such as contaminated local models (called contaminated models). For example, Blanchard et al. [24] demonstrated that even a single Byzantine user can alter the aggregated result by modifying its contaminated model's parameters. The idea behind this method is that the Byzantine user scales up each parameter of its contaminated model so that these parameters persist after the server's aggregation.

The assumed threat corresponds to either semi-honest adversaries or dishonest adversaries

depending on the application scenario. Semi-honest adversaries are assumed in applications where global models are used to enhance service quality, such as service provider-oriented applications (e.g., Google's Google Keyboard Query Suggestion [27]) and financial transaction applications (e.g., fraud detection models by banks). In these cases, while users such as Google keyboard users or banks may attempt to infer sensitive training data, they have no incentive to corrupt the global models themselves. In contrast, dishonest adversaries are assumed in highly competitive environments and security-enhanced applications. For example, in competitive environments, employees of a chat service provider may perform Byzantine attacks to sabotage rival service providers' model training. In security applications like phishing detection, adversaries may attempt to degrade the global model's performance to evade detection. Please note that in these applications, not only Byzantine users tamper with aggregation results, but also semi-honest users or the server are motivated to obtain the sensitive training data.

To enhance the security of federated learning, many studies to mitigate the attacks by semi-honest adversaries and dishonest adversaries have been proposed. To mitigate the attacks of semi-honest adversaries, methods called secure aggregation have been proposed [28–41]. Secure aggregation allows the users and server to collaboratively compute the sum of the local models while hiding the individual local models. In these methods, each user divides its local model into *shares* using cryptographic techniques such as Shamir's secret sharing [42] or additive secret sharing. This process of division is referred to as masking. Each user then sends these shares to other users and/or the server, aggregates the received shares, and sends the aggregated share to the server. Finally, the server aggregates all the shares and aggregated shares it received to obtain the sum of all the local models. Here, an important point is that the effect of masking is canceled by the server's aggregation. The server and users and cannot reconstruct other users' local models from exchanged information such as received shares or aggregated shares.

To mitigate the attacks of dishonest adversaries, methods called Byzantine-resilient secure aggregation have been proposed [25, 43] to detect and remove contaminated information pieces such as contaminated shares, contaminated aggregated shares, and contaminated models while performing secure aggregation. These methods detect and remove such contaminated information pieces as follows: First, each user generates and exchanges a commitment [44], which allows other users to verify if a share is generated from its local model corresponding to this commitment. If a contaminated share is detected, the server removes all the shares of the Byzantine user. Second, the server corrects contaminated aggregated shares by treating Shamir's secret sharing as a Reed-Solomon error correcting code [45]. Third, the users and the server jointly remove contaminated models by

securely checking local models using shares [24].

Towards the realization of federated learning, many studies that extend seminal works on secure aggregation or Byzantine-resistant secure aggregation have improved computational efficiency, communication efficiency, or both [32, 33, 37, 46–51]. However, these existing methods have compromised security, that is, privacy strength, Byzantine-resilient strength, or both (these studies are reviewed in the Related Work chapter).

The goal of this thesis is to make federated learning more accessible as a privacy-preserving machine learning system by improving performance while maintaining privacy strength and Byzantine-resilient strength. The communication bottleneck in existing protocols is due to the frequent transfer of shares and commitments, which are comparable in size to the model or even larger. Additionally, the computation bottleneck in existing protocols lies in the server's intensive computations related to shares from dropout users who leave the protocol and Byzantine users. The author reduces the communication cost by reducing the number of transfers of shares and commitments, and reduces the computation cost by optimizing computations at the algorithm level on the server. In the first half of this thesis, the author designs a fast secure aggregation protocol for secure federated learning against semi-honest adversaries, and in the second half of this thesis, the author designs a fast Byzanitne-resilient secure aggregation for secure federated learning against dishonest adversaries.

1.1 Fast Secure Federated Learning against Semi-honest Adversaries

1.1.1 Background

The goal of secure aggregation protocols for secure federated learning against semi-honest adversaries is to satisfy 1) *privacy preservation*, which means that individual local models are not revealed even if the server and/or users are semi-honest adversaries, and 2) *dropout tolerance*, which means that a sum of local models is computed even if some users drop out.

The key ideas of the secure aggregation protocols to satisfy privacy preservation and dropout tolerance are two-fold: First, users mask their local models with random vectors (random masks) before sending these local models to the server. (Here, the random masks and the masked local model correspond to the shares). Second, the server adds the masked local models of live users and subtracts the random masks of the live users and dropout users to compute the sum of the live users' local models (the series of processes at the server is the server's aggregation). An important point is that the users exchange information such as shares for the above methods.

There are two approaches to exchanging information pieces, depending on which improves

Protocol	Computation cost		Communication cost	
	User	Server	User	Server
SecAgg [29]	$O(nm)$	$O((n-r)m + r(n-r)m)$	$O(m)$	$O(nm)$
LightSecAgg [47]	$O(nm)$	$O(m)$	$O(nm)$	$O(n^2m)$
BalancedSecAgg	$O(nm)$	$O(rm)$	$O(rm)$	$O(rnm)$

Table 1.1: A cost analysis of SecAgg, LightSecAgg, and BalancedSecAgg. n is the number of all users and r is the number of dropout users. The defines a unit of communication cost as one transfer of an element of m-dimensional vector, such as a model or mask, and a unit of computation cost as the generation of an element of a mask. For a fair comparison, the author assumes the communication model of LightSecAgg is the same as that of SecAgg and BalancedSecAgg.

either communication or computation costs as shown in Table 1.1. The author defines a unit of communication cost as one transfer of an element of a model or mask with dimension m, and define a unit of computation cost as the generation of an element of a mask.

The first approach, proposed by Bonawitz et al. [29] and called SecAgg, has a low communication cost for each user. For privacy preservation, each user agrees on a pairwise random seed with every other user and generates a self-random seed. Then, each user generates the pairwise random masks and the self-random mask from these seeds. Then, each user adds these masks to its local model so that all the pairwise random masks of live users are canceled out during aggregation and then sends its masked local model to the server. For dropout tolerance, each user secretly shares information to recover the pairwise random seeds of the dropout users and the self-random seeds of the live users through Shamir's secret sharing [42]. The server collects their shares, reconstructs these seeds and masks for both dropout and live users, and subtracts these masks from the sum of the live users' masked local models to compute the sum of their local models. Since seeds and shares are much smaller than random masks, each user only sends its masked local model as m-dimensional vectors to the server. Therefore, the communication cost for each user is $O(m)$. At this expense, SecAgg has a disadvantage in terms of the computation cost for the server's mask reconstruction. Precisely, the server performs the reconstruction of one self-random mask for each live user and the reconstruction of $n-r$ pairwise random masks for each dropout user, where n and r are the numbers of all users and dropout users, respectively. Therefore, the computation cost for the server is $O((n-r)m + r(n-r)m)$. Bonawitz et al. claim that the server's mask reconstruction is time-consuming [29].

The second approach, proposed by So et al. [47] and called LightSecAgg, reduces the computation cost of the server's mask reconstruction by having users exchange and aggregate their masks. In LightSecAgg, each user masks its local model with a (self-) random mask and sends its masked

local model to the server. To compute the sum of live users' local models, the random masks of the live users are securely subtracted through a process of secretly sharing and aggregating them as follows: First, each user generates multiple encoded masks (shares) from its random mask through *t-private Maximum Distance Separable (MDS) codes* [42, 52]. Then, each user sends such encoded masks to the other users, and aggregates the received encoded masks. Finally, the server collects the aggregated encoded masks of the live users and decodes the sum of the live users' random masks. The t-private MDS codes prevent the server and any group of fewer than $t + 1$ semi-honest users in collusion from obtaining individual random masks of the other users. Additionally, these codes enable the server to reconstruct the aggregated random mask as long as the number of dropout users r is less than or equal to $r = n - (t + 1)$. Since the server reconstructs only the aggregated random mask, the computation cost for the server reduces to $O(m)$ compared to SecAgg. However, LightSecAgg incurs a high communication cost for each user. Each user sends an encoded mask as an m-dimensional vector to each other user, in addition to sending its masked local model and its aggregated encoded mask to the server. Unlike SecAgg, it is not possible to generate the encoded mask from a seed. Therefore, the communication cost for each user is $O(nm)$.

1.1.2 Approach

The author proposes a new approach, called *BalancedSecAgg*, to achieve a better balance between communication and computation costs than that of SecAgg and LightSecAgg by complementing both SecAgg and LightSecAgg. Assuming a total of n users, the author divides them into two groups: $t + 1$ users and $r = n - (t + 1)$ users. Each user generates $t + 1$ random masks and then generates r redundant masks from these random masks through Reed-Solomon erasure codes [53, 54]. Each user adds all the random masks and redundant masks to its local model and sends this masked local model to the server. To compute the sum of live users' local models, all the random masks and redundant masks of the live users are subtracted from the sum of the live users' masked local models. To this end, both the random masks and redundant masks are exchanged and aggregated among the n users. At this time, each user uses random seeds to exchange the $t + 1$ random masks. Then, the server collects the aggregated masks of the live users and, using these masks, reconstructs the aggregated masks attributed to the dropout users, as long as the number of dropout users is less than or equal to r. A key contribution is that the author demonstrates that using Reed-Solomon codes makes it unnecessary to prepare redundant masks for all users.

The contributions of this thesis are summarized below:

- The author designs the secure aggregation protocol, BalancedSecAgg, to achieve a better

balance between computation and communication costs. Compared to LightSecAgg, the communication cost for each user reduces to $O(rm)$. This is because each user sends a random seed to each of $t+1$ other users, a redundant mask to each of $r-1$ other users, its masked local model, and its aggregated mask to the server. In addition, compared to SecAgg, the computation cost for the server reduces to $O(rm)$. This is because the server reconstructs only one aggregated mask for each dropout user.

- To speed up actual protocol running time, the author adopts an efficient mask coding algorithm different from LightSecAgg. The t-private MDS code adopted by LightSecAgg typically involves Lagrange interpolation, which becomes inefficient for increasing t and is a computation bottleneck in LightSecAgg [47]. In contrast, the author adopts the Reed-Solomon erasure code for mask coding, leveraging its optimized implementation for rapid computation. In our experiment, BalancedSecAgg achieves up to 10.74× faster than LightSecAgg, and up to 11.41× faster than SecAgg.
- The author formally proves that BalancedSecAgg ensures privacy preservation and dropout tolerance in the condition $t \leq n-r-1$. This condition is the same level of privacy preservation and dropout tolerance as SecAgg and LightSecAgg. Although many secure aggregation protocols [32, 33, 37, 46–49] follow the principles of either SecAgg or LightSecAgg and aim to achieve communication and computation efficiency, these protocols sacrifice privacy preservation and dropout tolerance.

1.2 Fast Secure Federated Learning against Dishonest Adversaries

1.2.1 Background

In addition to privacy preservation and dropout tolerance, another goal of Byzantine-resilient secure aggregation for secure federated learning against dishonest adversaries is to satisfy *Byzantine-Resilience*, which means that if some users are Byzantine users, the sum of the local model is computed, excluding the Byzantine users' contaminated models. Among existing secure aggregations, The author focuses on BREA because the method can tolerate the largest number of Byzantine users (Please refer to the Related Work chapter for comparisons with other studies). BREA requires that each user broadcasts a commitment for share verification. In a naive method, a semi-honest server is responsible for broadcasting all users' commitments. Specifically, each user sends its commitment to the server and the server sends all the other users' commitments to each user using unicast channels. Since the server is semi-honest, the commitments are correctly delivered.

Protocol	Computation cost		Communication cost	
	User	Server	User	Server
BREA	$O(mn \log p + dn^2 \log n)$	-	$O(ntm)$	$O(n^2tm)$
BREA-SV	$O(mn \log p)$	$O(mn^2 \log^2 n)$	$O(tm)$	$O(ntm)$

Table 1.2: A cost analysis of BREA and BREA-SV. n, t, m, and p are the number of all users, the number of semi-honest users, the size of models, and the order of finite field, respectively.

However, in return for the strong resilience, BREA incurs a negative feature that the sizes of commitments are large (in bytes). In the naive method, each user uploads data of size tm, and downloads data of size $(n-1)tm$ because the size of each commitment is tm (t is the number of semi-honest users). Therefore, the communication cost for each user and the server is $O(ntm)$ and $O(n^2tm)$, respectively.

1.2.2 Approach

The aim of this thesis is to achieve faster learning by eliminating the communication bottleneck without relying on broadcasting. The author achieves the goal in two steps. In the first step, the author raises the question of how resilience is preserved without share verification. The motivation, here, is that Reed-Solomon code decoding [45] for correcting contaminated aggregated shares plays partially a role in share verification. In the second step, the author designs an algorithm for the share verification so that the computation of verification is performed by a semi-honest server, whereas each user performs the computation in BREA. A key contribution is that the author demonstrates that users can delegate share verification processes with commitments to the server without disclosing their secret information (shares) and that users need to send the commitments to only the server.

The contributions of this for the two steps are summarized below:

- THe author clarifies that Byzantine resilience is degraded when the share verification is omitted. Although the Reed-Solomon decoding detects contaminated shares of an attack of a Byzantine user, BREA without share verification is vulnerable to collusion between Byzantine users. The author designs a *colluding contaminated model injection attack*, where colluding Byzantine users use different shares for contaminated model removal and local model aggregation. This attack reduces the maximum number of Byzantine users from $\ell \leq (n-1-\max\{c+2, r+2t\})/2$ to $\ell < \lceil n/3 \rceil$, where ℓ, c, and r are the number of Byzantine users, users with local models selected for aggregation, and dropping users, respectively.
- The author designs an algorithm where some parts of the computation for the share verification

are offloaded from users to a server. The algorithm, which the author calls *BREA with Server's share Verification (BREA-SV)*, avoids each user downloading commitments from all the other users. Instead of downloading its commitment, each user just uploads its commitment to the server and the server is responsible for executing computation with users' commitments. This reduces the communication cost to $O(ntm)$, compared to BREA whose communication cost is $O(n^2tm)$, where m represents the size of vectors (models). Moreover, the author reduces computation time to achieve fast learning in the following two ways: First, the author reduces the number of multiplication operations of the server's computation to make the verification efficient. Second, the author enables users' model training and the server's computation in parallel. Since the server's computation can be offloaded to GPU, BREA-SV can be performed on the scale of hundreds of users.

- The author proves that BREA-SV satisfies the privacy preservation and Byzantine resilience simultaneously defined by BREA, under the same constraints as BREA, i.e., $\ell \leq (n - 1 - \max\{c + 2, r + 2t\})/2$.

Chapter 2

Related Work

This thesis adopts Byzantine-resilient and secure aggregation protocols for federated learning among many types of methods rather than centralized learning methods: the methods based on data processing like data anonymization [3] and differential privacy [4, 5], Secure Multi-Party Computation [6] and TEE [8–10]. The reason for not choosing these methods is that accuracy and low computation cost are not easy to be simultaneously achieved. This discussion is described in Chapter 1.

To reduce the computation and communication costs of Byzantine-resilient and secure aggregation for federated learning, many computation and communication-efficient approaches have been proposed. However, these existing approaches sacrifice the strength of privacy preservation, dropout tolerance, and Byzantine resilience. Specifically, these existing approaches 1) maintain the privacy preservation, dropout tolerance, and Byzantine resilience conditions, but reduce the probability of satisfying the conditions, or 2) weaken the conditions as shown in Table 2.1 and Table 2.2. The author shows that few studies improve efficiency without sacrificing the strength of privacy preservation, dropout tolerance, and Byzantine resilience.

2.1 Computation and Communication Efficient Secure Aggregation

In this section, the author reviews computation and/or communication efficient approaches to reduce the computation and communication costs of SecAgg and LightSecAgg.

SecAgg+ [32] and Lisa [46] are re seed exchange-based protocols that extend SecAgg to reduce both computation and communication costs. The main idea behind these protocols is that each user shares pairwise random seeds and secretly shared information pieces only with a subset of selected

Protocol	Privacy Preservation		Dropout Tolerance	
	Condition	Probability	Condition	Probability
SecAgg [29] LightSecAgg [47] BalancedSecAgg	$t \leq n-r-1$	1	$r \leq n-t-1$	1
SecAgg+ [32] Lisa [46]	$t \leq n-r-1$	$2^{-\epsilon}$	$r \leq n-t-1$	$2^{-\mu}$
TurboAgg [33]	$t \leq n-r-1$	$\frac{n}{n'}e^{-(\gamma n')}$	$r \leq n-t-1$	$\frac{n}{n'}e^{-(\delta n')}$
SwiftAgg [48]	$t \leq n'-r-1$	1	$r \leq n'-t-1$	1
SwiftAgg+ [49]	$t \leq n'-r-h$	1	$r \leq n'-t-h$	1
FastSecAgg [37] LightSecAgg (Extended version)	$t \leq n-r-h$	1	$r \leq n-t-h$	1

Table 2.1: Comparison of our protocol (BalancedSecAgg) with other related works. n' is the number of selected users. ϵ, μ, γ, and δ are system parameters.

users, rather than with all other users. This idea reduces communication costs because each user communicates only with the selected users. It also reduces computation costs by decreasing the number of pairwise random masks that the server needs to reconstruct in the mask subtraction phase. However, these protocols require that, among the selected users for each user, there are no more than $t+1$ semi-honest users and that no more than $n'-(t+1)$ dropout users where n' is the number of selected users. SecAgg satisfies these conditions with probability 1 because the selected users for each user include all the other users. In contrast, the above-mentioned protocols satisfy these conditions with a probability less than 1, these protocol involve a trade-off between efficiency and security. When the probabilities of satisfying conditions privacy preservation and dropout tolerance are $2^{-\epsilon}$ and $2^{-\mu}$, respectively, SecAgg+ and Lisa set n' to $O(\log n+\epsilon+\mu)$ and $O(\epsilon+\mu)$, respectively.

TurboAgg [33] and SwiftAgg [48] are mask exchange-based protocols like LightSecAgg. These methods reduce the communication costs and the computation costs by using ideas similar to the above approaches [32, 38, 40, 41, 46]. Specifically, each user secretly shares its local model only with a subset of selected users. Consequently, these methods satisfy privacy preservation and dropout tolerance with a probability of less than 1 for reasons similar to those explained above. Jahani-Nezhad et al. [48] claims that setting the number of selected users to $t+r$ allows the probability of satisfying security to be 1. However, this method reduces the number of acceptable semi-honest users to $t \leq n'-r-1$.

SwiftAgg+ [49] is the extended version of SwiftAgg. It additionally reduces the communication cost by reducing the amount of data in each share from an m-dimensional vector to an $\frac{m}{h}$-dimensional

vector ($1 \leq h \leq n - r$). The secure aggregation with this approach is performed as follows. First, each user splits its local model into h sub-models(vectors), each of which dimension $\frac{m}{h}$, generates a polynomial of degree $t + h$ from these sub-models and t random vectors of the same dimension, and sends an evaluation of this polynomial, i.e., a share, also of dimension $\frac{m}{h}$, to each other user. Then, the users aggregate the received shares and send the aggregated shares to the server. The server then reconstructs a polynomial, of degree $t + h$, that represents the sum of all users' polynomials and computes the sum of local models by combining the coefficients of model parts from this polynomial. However, this model-split approach incurs the problem that it requires $t + h = n - r$ aggregated shares for the server to reconstruct the server's polynomial. The larger the number of sub-vectors h becomes, the smaller the number of semi-honest users t becomes. Precisely, SwiftAgg+ achieves privacy preservation and dropout tolerance under the condition $t \leq n - r - h$.

FastSecAgg [37] adopts a communication-efficient approach similar to the model-split approach of SwiftAgg+ [49]. That is, each user splits its local model and then generates n shares, each of which is an $\frac{m}{h}$-dimensional vector, thereby reducing the communication cost. Furthermore, it reduces the computation cost involved in share generation using Fast Fourier Transform. However, FastSecAgg achieves privacy preservation and dropout tolerance under the condition $t \leq n - r - h$, which is the same as SwiftAgg+ [49].

Finally, LightSecAgg [47] also adopts a communication-efficient approach similar to the model-split approach [49]. In LightSecAgg, each user splits its random mask and then generates n encoded masks, each encoded mask is an $\frac{m}{h}$-dimensional vector, thereby reducing the communication cost. LightSecAgg achieves privacy preservation and dropout tolerance under the condition $t \leq n - r - h$ for the same reason as SwiftAgg+ [49].

2.2 Computation and Communication Efficient Byzantine-resilient Secure Aggregation

In this section, the author reviews computation and communication efficient approaches to reduce the computation and communication costs of BREA.

Rofl [50] is a Byzantine-resilient secure aggregation protocol that improves computation and communication efficiency by avoiding share verification. Since Multi-Krum in BREA cannot be used without share verification, Rofl introduces a mechanism of checking local model of each user and the aggregation result using *commitments* to users' local models. In addition to providing the aggregation result of the server through a secure aggregation protocol, users generate the commitments of their

Protocol	Privacy Preservation		Dropout Tolerance		Byzantine Resilience			
					Remove Contaminated model		Remove Contaminated computation result	
	Condition	Probability	Condition	Probability	Condition	Probability	Condition	Probability
Rofl [50]	$t \leq n - r - \ell - 1$	1	$r \leq n - t - \ell - 1$	1	$\ell \leq n$	1	-	-
BynSecAgg [51]	$t \leq n - (r + 2\ell + h)$	1	$r \leq n - (2t + h) - 2\ell$	1	$\ell < \frac{n-c-2}{2}$	1	-	-
BREA [25] BREA-SV	$t \leq n - (r + 2\ell + 1)$	1	$r \leq n - (2t + 1) - 2\ell$	1	$\ell < \frac{n-c-2}{2}$	1	$\ell \leq \left\lfloor \frac{n-(2t+1)-r}{2} \right\rfloor$	1

Table 2.2: Comparison of our protocol (BREA-SV) with other related works.

local models and sends them to the server. The server verifies whether the local models are outliers using the norm of the commitments. Furthermore, the server aggregates all the users' commitments and compares the resulting commitment with the aggregation result of all users' local models. This allows the server to confirm that the verified local models are included in the aggregation. However, Rofl lacks the ability to correct contaminated computation results like Reed-Solomon error correcting methods. Therefore, even if the server detects that the verified local models are not included in the aggregation result, it cannot remove them. If Reed-Solomon error correcting codes are introduced, Byzantine resilience still decreases, as shown in Chapter 5.

ByzSecAgg [51], which extends BREA to improve the communication efficiency of the share verification of BREA. BynSecAgg similarly performs the share verification, but reduces the size of each commitment. Here, let the commitment of a user i be denoted as $\mathbf{C}_{f_i(\theta,\mathbf{w}_i)} = (g^{\mathbf{w}_i}, g^{\mathbf{r}_{i1}}, g^{\mathbf{r}_{i2}}, \ldots, g^{\mathbf{r}_{it}})$, where f is a polynomial for generating shares, $(\mathbf{w}_i, \mathbf{r}_{i1}, \mathbf{r}_{i2}, \ldots, \mathbf{r}_{it})$ are its coefficients, and g is a generator selected from a finite field $\mathbb{F}_p$. The idea of reduction is to verify the elements in each share collectively, without verifying each element one by one like BREA. This idea is achieved by verifying whether a sum of elements of each exchanged share equals a sum of elements of a correct share or not. Specifically, each user i sends the sum of the elements in each column in the commitment $\mathbf{C}_{f_i(\theta,\mathbf{w}_i)} = (g^{\mathbf{w}_i}, g^{\mathbf{r}_{i1}}, g^{\mathbf{r}_{i2}}, \ldots, g^{\mathbf{r}_{it}})$ with a share $\mathbf{sh}_{i,j}$ to each verifying user j. For example, $g^{\mathbf{w}_i} = [g^{v_{01}}, g^{v_{02}}, \ldots, g^{v_{0d}}], g^{\mathbf{r}_{i1}} = [g^{v_{11}}, g^{v_{12}}, \cdots, g^{v_{1d}}], \ldots, g^{\mathbf{r}_{it}} = [g^{v_{t1}}, g^{v_{t2}}, \ldots, g^{v_{td}}]$, i sends $(g^{v_0}, g^{v_1}, \ldots, g^{v_t})$ with $\mathbf{sh}_{i,j} = [s_{i1}, s_{i2}, \ldots, s_{id}]$ to each verifying user j, where $g^{v_k} = g^{\sum_{i=1}^{d} v_{ki}}$. User j verifies whether the sum of all elements of $\mathbf{sh}_{i,j}$, $g^{\sum_{k=1}^{d}, s_{ik}}$, equals to $g^{v_0 + \sum_{k=1}^{t} v_k {\theta_j}^k}$. Whereas the communication cost is reduced from $O(n^2tm)$ to $O(tn^2)$, any Byzantine user b can send a contaminated share $\mathbf{a}_{bj} = [x_{b1}, x_{b2}, \ldots, x_{bd}] (\neq \mathbf{sh}_{bj} = [s_{b1}, s_{b2}, \ldots, s_{bj}])$ without being detected by this share verification because b can rewrite $\mathbf{sh}_{b,j}$ so that the sum of all the elements of the sending share, $\sum_{k=1}^{d} x_k$, equals to $\sum_{k=1}^{d} s_k$. For example, for $m = 2$ and $\mathbf{sh}_{b,j} = [1, 2]$, b can send

$\mathbf{a}_{b,j} = [0, 3]$ to user j. The random numbers $(g^{\beta^1}, g^{\beta^2}, \ldots, g^{\beta^d})$ $(\beta \in \mathbb{F}_p)$ introduced by [51] prevent this attack, but require a trusted entity to generate them. This attack weakens the Byzantine resistance of BynSecAgg to the same level as RoFL without the trusted entity. BREA-SV, on the other hand, has the same Byzantine resilience as BREA and achieves communication efficiency.

Chapter 3

Threat Models and Goals in Federated Learning

In this chapter, the author explains the two types of threats in federated learning and the goals for mitigating each threat. Chapter 4 focuses on how to achieve the first goal, while Chapter 5 focuses on how to achieve the second goal. In Section 3.1, the author first explains a federated learning system. Next, in Section 3.2, the author describes the two types of threat models and attacks, along with the application scenarios corresponding to each threat model. In Section 3.3, the author explains the goals for each of the two threat models.

The main symbols used in this thesis are listed in Table 3.1. The author uses bold and uppercase letters to denote vectors and sets, respectively, and lowercase letters for other purposes such as scalar variables.

3.1 Federated Learning

The author considers a federated learning system coordinated by a server, in which a set of n users $\mathcal{U}$ trains a global model collaboratively. At each iteration, each user i ($\forall i \in \mathcal{U}$) downloads the current global model $\mathbf{w}$ with dimension m from the server. The author assumes that the system uses a neural network, of which model $\mathbf{w}$ consists of several weight and bias vectors depending on the layering structure of the neural network. Next, each user i locally trains $\mathbf{w}$ with its training data D_i and obtains the individual model $\Delta\mathbf{w}_i = \mathcal{L}(\mathbf{w}, D_i)$, where $\mathcal{L}$ is a learning algorithm used in this system. Then, each user i sends $\Delta\mathbf{w}_i$ in the form of $\mathbf{w}_i = \Delta\mathbf{w}_i - \mathbf{w}$ to the server. The author refers to $\mathbf{w}_i$ as a *local model*. The server aggregates (adds) their collected local models and updates the global model using

Table 3.1: Summary of Symbols

Symbol	Description
n	Number of all users
c	Number of users selected by Multi-Krum
t	Number of semi-honest users
r	Number of dropout users
ℓ	Number of Byzantine users
m	Size of a model (vector)
$\mathbf{w}$	Global model
$\mathbf{w}_i$	Local model of a user i
$\mathbf{w}_b^C$	Contaminated model of a Byzantine user b
$\mathbf{sh}_{i,j}$	Share of a user i for a user j
D_i	Training data of a user i
$\mathcal{U}$	Set of all users
$\mathcal{T}$	Set of semi-honest users
$\mathcal{D}$	Set of dropout users who leave before sending their (masked) local models
$\mathcal{B}$	Set of Byzantine users
$\mathcal{U}^{sel}$	Set of users selected by Multi-Krum
$\mathcal{L}$	Learning algorithm

the sum of local models as follows:

$$\mathbf{w}' = \mathbf{w} + \eta \sum_{i \in \mathcal{U}} \mathbf{w}_i, \tag{3.1}$$

where η is the learning rate to determine how much $\mathbf{w}$ is updated at every iteration.

A certain number (r) of users may leave the federated learning before sending its local models due to fluctuations in communication conditions such as network congestion or low battery. These users refer to those who do not send their masked local model in local model aggregation protocol in Chapter 4 and Chapter 5. The author refers to the leaving users as dropout users, and in particular, let $\mathcal{D}$ be the set of dropout users who do not send their masked local models to the server. The author assumes the same communication model as that of Bonawitz et al. [55] where the communication between users is mediated by the server.

The author assumes a trusted authority that assigns public key certificates to all the users and the server. These keys are used to establish a private and authenticated channel among a pair of parties like users and the server.

3.2 Threat Models and Attacks

The author assumes two types of threats depending on the application scenarios.

3.2.1 Threat Model1: Semi-honest Model

In semi-honest model, all adversaries are semi-honest, which means that they follow the protocol but perform privacy attacks. These adversaries include the server and some users (up to t users), and their goal is to obtain the local models of any honest user. The author denotes the set of t semi-honest users as $\mathcal{T}$. When semi-honest adversaries collude, they share the information received through the protocol to obtain honest users' local models. If adversaries obtain any honest user's local model, they can infer the honest user's training data from the local model by performing privacy attacks described in the following section. In the naive federated learning system explained in Section 3.1, the server obtains the local models of all honest users, leading to privacy violations.

Privacy Attack

In this subsection, the author describes a specific privacy attack in which a semi-honest adversary, who obtains any honest user's local model, infers the honest user's training data.

Attack Example. Wang et al. [20] demonstrate that the semi-honest server successfully extracts training data from a local model by a model inversion attack.

Wang et al use GAN (Generative Adversarial Networks) for a privacy attack. GAN is an ensemble learning model that generates data similar to the input data. The server trains the ensemble learning model with data similar to each user's training data and then infers each user's training data using the trained ensemble learning model.

In federated learning, the server cannot obtain the training data of each user, but it can reconstruct data similar to the users' training data through the following iterative process using local models. 1) the server generates random noise as training data and trains the global model locally. 2) the server adjusts the random noise so that the local model generated by the server approximates the values of the local models received from each user.

Application Scenarios

The Semi-honest models described in the previous section are assumed in the following applications. The first is service provider-oriented applications where the global model is used to improve their

service quality. An example service is Google's Google Keyboard Query Suggestion [27], a machine learning service that provides search suggestions to Google Keyboard users. In this case, the training data is the search query text, which may contain sensitive information. The providers may perform privacy attacks but do not any motivation to contaminate the models. The second is financial transaction applications, where banks and financial institutions cooperate with each other. For example, they create a machine learning model for fraud detection. In this case, each institute may try to reveal others' training data, but they do not have any motivation to contaminate the models.

3.2.2 Threat Model2: Dishonest Model

In dishonest model, in addition to the semi-honest adversaries described in Section 3.2.1, Byzantine adversaries who conduct security attacks (Byzantine attacks) appear as well. These Byzantine adversaries include some users (up to ℓ users), and the goal of the Byzantine users is to tamper with a sum of the local models. The author denotes the set of ℓ semi-honest users as $\mathcal{B}$. If contaminated messages sent by Byzantine users are not detected and removed, the sum of local models may be compromised. In the naive federated learning system described in Section 3.1, it is possible to alter the aggregation result to arbitrary values by sending malicious models(called contaminated models), as discussed in the next subsection.

Byzantine Attack (Security attack)

The two types of Byzantine attacks are assumed: The first type is to inject a local model that a Byzantine user obtains by training a global model with contaminated training data. Hereafter, the author calls this model the *contaminated model*. In this attack, the Byzantine user *does* follow a protocol, i.e., sends benign shares and an aggregated share, but the contaminated model is injected into the aggregation result. The second type is to inject contaminated shares and contaminated share computation results, by not following the protocol. Byzantine users send such contaminated information pieces to the other users or the server.

Attack Example. Blanchard et al. [24] demonstrate a Byzantine attack where a single Byzantine user can alter the aggregation result to any desired value by sending a contaminated model to the server. The main idea is that when the Byzantine user b aims for the aggregation result to be a specific value V, it scales up the parameters of its contaminated model $\mathbf{w}_b^C$ to ensure V remains. In

the naive approach, the Byzantine user b sends a contaminated model

$$\mathbf{w}_b^C = V - \sum_{i \in \mathcal{U} \setminus \{b\}} \mathbf{w}_i \tag{3.2}$$

to the server. As a result, the server obtains the aggregation result V. In realistic settings, since Byzantine users do not know the exact value of the local models of honest users, they use the estimated local models of the honest users.

Application Scenarios

The dishonest models described in the previous section are assumed in the following applications. The first is an application in highly competitive environments. For example, assume that several interactive chat service providers develop machine learning models for improving their service quality. Employees of some providers have a motivation to contaminate the models of other providers by performing Byzantine attacks as users of the other providers. The second is in machine learning applications that enhance security. For example. phishing detection services, a security service provider aims to develop a machine learning model that detects phishing methods, using data such as email metadata, link patterns, and visited URLs of individuals' smartphones. In this scenario, an adversary may participate in federated learning with a motive to degrade the detection capabilities of the machine learning model to evade phishing detection. Please note that in these applications, not only Byzantine users tamper with aggregation results, but also semi-honest users or the server are motivated to obtain the sensitive training data.

3.3 Goals

3.3.1 Goal for Mitigating Threats in Semi-Honest Model

The goal is to protect the privacy of honest users' local models. Specifically, the goal is to prevent the server and t semi-honest users who collude from obtaining the local model of any honest user $h \in \mathcal{U} \setminus \mathcal{T}$ from information received during the protocol. In Chapter 4, the author introduces secure aggregation based on local model masking, where semi-honest adversaries cannot obtain the local model of any honest user, but the server computes the sum of all users' local models.

Note that the author does not consider the leakage of training data from a sum of local models as a privacy issue in many application scenarios. This is because even if the server can infer training

data from the sum of local models, it cannot determine which user's training data it is.

3.3.2 Goal for Mitigating Threats in Dishonest Model

There are two goals: The first goal is to prevent the server and t semi-honest users who collude from obtaining the local model of any honest user $h \in \mathcal{U} \setminus \mathcal{T}$ from information received during the protocol. The second goal is to aggregate the local models of all honest users in $\mathcal{U} \setminus \mathcal{B}$. In other words, contaminated models are excluded and only other local models are aggregated. In Chapter 5, the author introduces Byzantine-resilient secure aggregation based on the verification and division of local models, which ensures secure computation of the sum of local models while detecting and eliminating malicious messages.

Chapter 4

Fast Secure Aggregation against Semi-honest Adversaries

In this chapter, the author addresses threats in the semi-honest model and designs a faster secure aggregation protocol called BalancedSecAgg than the existing ones. The rest of the chapter is organized as follows: Section 4.1 describes the preliminaries used in this chapter. Section 4.2 describes the existing secure aggregation protocols, SecAgg and LightSecAgg, and then Section 4.3 describes our protocol, BalancedSecAgg. Section 4.4 analyzes BalancedSecAgg in terms of security and performance. Section 4.6 concludes this chapter.

In addition to Table 3.1, the main symbols used in this chapter are listed in Table 4.1. The author uses bold and uppercase letters to denote vectors and sets, respectively, and lowercase letters for other purposes such as scalar variables.

Table 4.1: Summary of Symbols

Symbol	Description
$\widetilde{\mathbf{w}}_i$	Masked local model of a user i
PRG	Pseudorandom generator
PRG$(s_{i,j})$	Random mask generated by **PRG** from a seed $s_{i,j}$ of a user i for a user j
$\mathbf{e}_{i,j}$	Encoded mask of a user i for a user j
$\mathbf{d}_{i,j}$	Redundant mask of a user i for a user j
RS.**correct**	Erasure correction function

4.1 Preliminaries

4.1.1 Key Agreement

A key agreement consists of a tuple of algorithms (**KA**.**param**, **KA**.**gen**, **KA**.**agree**). Given a security parameter κ, **KA**.**param**$(\kappa) \rightarrow pp$ generates some public parameters (over which our protocol will be parameterized). **KA**.**gen**$(pp) \rightarrow (pk_i, sk_i)$ allows any user i to generate a public-private key pair. **KA**.**agree**$(sk_i, pk_j) \rightarrow a_{i,j}$ allows any user i to agree on a private pairwise key $a_{i,j}$ with another user j by combining i's private key sk_i with j's public key pk_j.

Correctness requires that, for any pair of user i and j, **KA**.**agree**$(sk_i, pk_j) =$ **KA**.**agree**$(sk_j, pk_i) = a_{i,j}$. Security requires that, in the honest-but-curious model, $a_{i,j}$ is indistinguishable from a uniformly random string for an adversary who is given the public keys of i and j. Since the author uses Diffie-Hellman key agreement [56] as the specific key agreement scheme, the author relys on the Decisional Diffie-Hellman (DDH) assumption.

4.1.2 Pseudorandom Generator

A secure pseudorandom generator [57, 58] **PRG** takes a uniformly random seed of fixed length as an input and produces a sequence of numbers where each number is in the range $[0, p)$. Security requires that the output is computationally indistinguishable from a sequence of truly random numbers drawn uniformly from the same range, given that the seed is hidden from the distinguisher.

4.1.3 Authenticated Encryption

Authenticated encryption provides messages exchanged between two parties with confidentiality and integrity guarantees. It consists of an encryption algorithm **AE**.**enc** that encrypts a message using a key to produce a ciphertext, and a decryption algorithm **AE**.**dec** that decrypts the ciphertext with the key to either reveal the original plaintext or output a special error symbol($\perp$).

Correctness requires that for all keys $k \in \{0,1\}^\kappa$ and all messages x, **AE**.**dec**$(k,$ **AE**.**enc**$(k, x)) = x$. Security requires indistinguishability under a chosen plaintext attack (IND-CPA) and ciphertext integrity (IND-CTXT) [59].

4.1.4 Shamir's Secret Sharing

Shamir's $(t+1)$-out-of-n secret sharing [42] allows a user to divide a secret s into n shares so that any $t+1$ shares can be used to reconstruct s, but any t or fewer shares reveal no information about

s. This scheme is parameterized over a finite field $\mathbb{F}$ and consists of the two algorithms **SS**.**share** and **SS**.**recon**. **SS**.**share**$(s, t+1, \mathcal{U}) \rightarrow \{(i, sh_i^s)\}_{i\in\mathcal{U}}$ takes as inputs a secret s, a threshold $t+1$, and a set $\mathcal{U}$ of n field elements, and outputs a set of n shares. The algorithm chooses a random polynomial $f \in \mathbb{F}[X]$ such that $f(0) = s$ and generates the shares as $\{(i, f(i))\}_{i\in\mathcal{U}} = \{(i, sh_i^s)\}_{i\in\mathcal{U}}$. **SS**.**recon**$(\{(i, sh_i^s)\}_{i\in\mathcal{V}}, t+1) = s$ takes as inputs the shares corresponding to a subset $\mathcal{V} \subseteq \mathcal{U}$ such that $|\mathcal{V}| \geq t+1$ and the threshold $t+1$, outputs s.

Correctness requires that $\forall s \in \mathbb{F}$, $\forall t+1, n$ with $1 \leq t+1 \leq n$, $\mathcal{U} \subseteq \mathbb{F}$ where $|\mathcal{U}| = n$, if **SS**.**share**$(s, t+1, \mathcal{U}) \rightarrow \{(i, sh_i^s)\}_{i\in\mathcal{U}}$, $\mathcal{V} \subseteq \mathcal{U}$ and $|\mathcal{V}| \geq t+1$, then **SS**.**recon**$(\{(i, sh_i^s)\}_{i\in\mathcal{V}}, t+1) = s$. Security requires that $\forall s, s' \in \mathbb{F}$ and any $\mathcal{V} \subseteq \mathcal{U}$ such that $|\mathcal{V}| < t+1$:

$$\{\textbf{SS}.\textbf{share}(s, t+1, \mathcal{U}) \rightarrow \{(i, sh_i^s)\}_{i\in\mathcal{U}} : \{(i, sh_i^s)\}_{i\in\mathcal{V}}\} \equiv$$
$$\{\textbf{SS}.\textbf{share}(s', t+1, \mathcal{U}) \rightarrow \{(i, sh_i^{s'})\}_{i\in\mathcal{U}} : \{(i, sh_i^{s'})\}_{i\in\mathcal{V}}\}$$

where $\equiv$ denotes that two distributions are identical.

Shamir's Secret Sharing scheme is characterized by the following features. First, this scheme has *linearity* with respect to addition: Consider two sets of shares $\{(i, sh_i^{s_a})\}_{i\in\mathcal{U}}$ and $\{(i, sh_i^{s_b})\}_{i\in\mathcal{U}}$ generated from two secrets s_a and s_b, respectively. The set $\{(i, sh_i^{s_a} + sh_i^{s_b})\}_{i\in\mathcal{U}}$ can be considered as the set of shares generated from the secret $s_a + s_b$. Second, this scheme is closely related to a polynomial-based Reed-Solomon erasure code [53]. This relationship stems from the fact that n shares generated by **SS**.**share** can be considered equivalent to the n symbols composing a codeword in Reed-Solomon erasure codes [53].

4.1.5 Reed-Solomon Erasure Codes

Reed-Solomon erasure codes provide encoding and decoding processes, which are defined by two functions: **RS**.**encode** and **RS**.**decode**. **RS**.**encode**$(s_1, s_2, \ldots, s_{t+1}, n) \rightarrow c_1, c_2, \ldots, c_n$ takes $t+1$ symbols composing the information word $(s_1, s_2, \ldots, s_{t+1})$ and the length of the codeword (n), and outputs n symbols composing the codeword $(c_1, c_2, \ldots, c_n)$. In contrast, **RS**.**decode**$(\{c_i\}_{i\in\mathcal{V}}, t+1) = s_1, s_2, \ldots, s_{t+1}$ takes as the input a set of symbols in the codeword c_i for $i \in \mathcal{V}$, where $|\mathcal{V}| \geq t+1$, and outputs the original information word.

The important features are two-folded: The first feature is that **RS**.**decode** includes an erasure correction function, **RS**.**correct**, which generates $n-(t+1)$ dropout symbols in the codeword. Given

n symbols composing the codeword $(c_1, c_2, \ldots, c_n)$, $\mathbf{RS.correct}(\{c_i\}_{i\in\mathcal{V}}, t+1) \rightarrow \{c_j\}_{j\in[n]\setminus\mathcal{V}}$ takes as the input any set of $t+1$ symbols c_i for $i \in \mathcal{V}$, where $|\mathcal{V}| \geq t+1$, and outputs the set of the remaining $n-(t+1)$ symbols. This function can be considered to generate additional $n-(t+1)$ symbols from $t+1$ symbols. The author refers to the $t+1$ symbols as *input symbols* and the additional $n-(t+1)$ symbols as *redundant symbols*. The second feature is the linearity with respect to addition like Shamir's secret sharing. Consider two sets of symbols $\{c_i^a\}_{i\in\mathcal{V}}$ and $\{c_i^b\}_{i\in\mathcal{V}}$. The addition of the erasure correction results of these two sets is the same as the erasure correction result of $\{c_i^a + c_i^b\}_{i\in\mathcal{V}}$.

4.2 Existing Secure Model Aggregation: SecAgg and LightSecAgg

This section analyzes the cost of existing secure aggregation protocols, SecAgg and LightSecAgg. In Section 4.2.1, the author describes secure aggregation techniques as the rationale for attack mitigation in the semi-honest model. In Section 4.2.2 and Section 4.2.3, the author overviews existing secure aggregation protocol of SecAgg and LightSecAgg, respectively. The author describes the communication and computation costs of these protocols shown in Table 1.1.

4.2.1 Rationale for Attack Mitigation of Semi-Honest Model

To mitigate the privacy attack, secure aggregation techniques based on local model masking have been proposed. These techniques have two ideas: the local model masking method and the mask subtraction method. In the local model masking method, users generate random masks (shares), add them to their local models, and send their masked local models to the server. In the mask subtraction method, the server aggregates users' masked local models and subtracts the random masks of some users to obtain the sum of the users' local models.

Example1: Pairwise Random Masking Method. A naive secure aggregation adopts a pairwise masking method. Each pair of users (i, j) agrees on a random vector (called a random mask) $\mathbf{r}_{i,j}$. If one of them adds $\mathbf{r}_{i,j}$ to its local model, and the other user subtracts $\mathbf{r}_{i,j}$, all the random masks are canceled out when all the masked local model are added, while all the local models are not revealed to the server. That is, each user sends its masked local model (its share) $\widetilde{\mathbf{w}}_i$ denoted by

$$\widetilde{\mathbf{w}}_i = \mathbf{w}_i + \sum_{j\in\mathcal{U}:i<j} \mathbf{r}_{i,j} - \sum_{j\in\mathcal{U}:i>j} \mathbf{r}_{i,j} \pmod{p}. \tag{4.1}$$

to the server. Then, the server aggregates the masked local models of the users as follows:

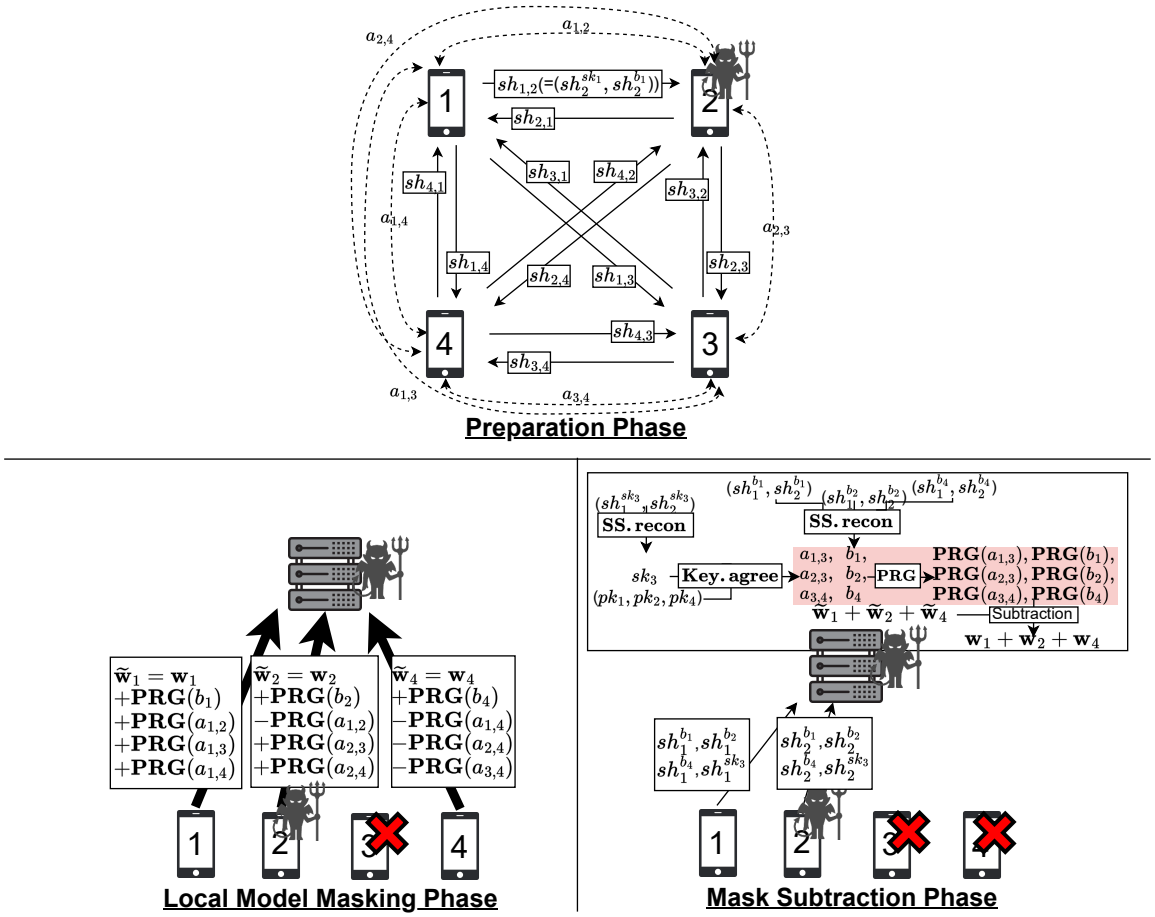

Figure 4.1: An example of SecAgg with $n = 4$ users, where user 2 is a semi-honest user ($t = 1$) and user 3 and user 4 are dropout users ($r = 2, \mathcal{D} = \{3\}$). Thin and thick lines indicate scalars and vectors, respectively.

$$\sum_{i\in\mathcal{U}} \widetilde{\mathbf{w}}_i = \sum_{i\in\mathcal{U}} \left(\mathbf{w}_i + \sum_{j\in\mathcal{U}:i<j} \mathbf{r}_{i,j} - \sum_{j\in\mathcal{U}:i>j} \mathbf{r}_{i,j} \right) = \sum_{i\in\mathcal{U}} \mathbf{w}_i \pmod{p}. \tag{4.2}$$

Example2: Self-Random Masking Method. The author adopts a self-random masking method to reduce the computation cost for the server due to dropout users, as shown in the author's protocol called BalancedSecAgg in Chapter 4.

First, each user i ($\forall i \in \mathcal{U}$) divides its local model $\mathbf{w}_i$ into $n+1$ shares, $\{\mathbf{sh}_{i,1}, \mathbf{sh}_{i,2}, \ldots, \mathbf{sh}_{i,n+1}\}$. The first n of $n+1$ shares are random masks ($(\mathbf{sh}_{i,1}, \ldots, \mathbf{sh}_{i,n}) = (\mathbf{r}_{i1}, \ldots, \mathbf{r}_{in})$), and the last share is

the masked local model ($\mathbf{w}_i + \sum_{j\in[n]}(-\mathbf{r}_{ij})$). Therefore, shares satisfy

$$\mathbf{w}_i = \sum_{j\in[n+1]} \mathbf{sh}_{i,j} \pmod{p}. \tag{4.3}$$

Then each user exchanges its shares with the other users and the server. Specifically, each user sends a different random mask to each other user and sends its masked local models to the server. Next, each user and the server ($j \in \mathcal{U} \cup \{S\}$) add the received shares, referred to as an *aggregated shares* λ_j. Next, each user sends the aggregated share λ_i to the server. Finally, the server computes the sum of users' local models by aggregating all the aggregated shares and the masked local models of users as follows:

$$\sum_{j\in\mathcal{U}\cup\{S\}} \lambda_j = \sum_{j\in\mathcal{U}} \mathbf{w}_j \pmod{p}. \tag{4.4}$$

4.2.2 SecAgg

Technical Intuition

SecAgg uses two types of random seeds for generating masks. First, each pair of users (i, j) securely agree on a pairwise random seed $a_{i,j} = \mathbf{KA}.\mathbf{agree}(sk_i, pk_j) = \mathbf{KA}.\mathbf{agree}(sk_j, pk_i)$ using their private-public key pairs (sk_i, pk_i), (sk_j, pk_j). Both users i and j generate the same pairwise random mask $\mathbf{PRG}(a_{i,j})$ through a secure pseudorandom generator $\mathbf{PRG}$. One of them adds $\mathbf{PRG}(a_{i,j})$ to its local model, and the other user subtracts $\mathbf{PRG}(a_{i,j})$. Second, each user i introduces a self-random seed b_i for use when it is delayed, which means that i sends its masked local model after the server considers i as a dropout user. This b_i is crucial because all the pairwise masks of i are revealed to the server when i is regarded as a dropout user, as explained in the next paragraph. Each user i also adds a self-random mask $\mathbf{PRG}(b_i)$ to its local model. As a result, each user i generates its masked local model $\widetilde{\mathbf{w}}_i$ denoted by

$$\widetilde{\mathbf{w}}_i = \mathbf{w}_i + \underbrace{\mathbf{PRG}(b_i)}_{\text{self-random mask}} + \underbrace{\sum_{j\in\mathcal{U}:i<j} \mathbf{PRG}(a_{i,j}) - \sum_{j\in\mathcal{U}:i>j} \mathbf{PRG}(a_{j,i})}_{\text{pairwise random masks}} \pmod{p}. \tag{4.5}$$

The server collects and aggregates the masked local models of live users, i.e., the users in $\mathcal{U} \setminus \mathcal{D}$. This aggregate includes the self-random masks of all the live users and the pairwise random masks of all the dropout users. Therefore, all these masks should be subtracted to compute the sum of the local models of the live users. To this end, each user i secretly shares its random seed b_i and private key sk_i with the other users through $(t+1)$-out-of-n Shamir's secret sharing [42]. This ensures that an original value cannot be reconstructed from at most any t shares and it can be reconstructed even if up to $n-(t+1)$ shares are lost. Then, the server collects the shares of both the private keys of all the live users and the random seeds of all the dropout users. Using these shares, the server reconstructs their random seeds and their private keys, and then computes the sum of the local models of all the live users as follows:

$$\sum_{i\in\mathcal{U}\setminus\mathcal{D}} \mathbf{w}_i = \sum_{i\in\mathcal{U}\setminus\mathcal{D}} \widetilde{\mathbf{w}}_i - \sum_{i\in\mathcal{U}\setminus\mathcal{D}} \mathbf{PRG}(b_i) - \sum_{i\in\mathcal{D}} \left(\sum_{j\in\mathcal{U}\setminus\mathcal{D}:i>j} \mathbf{PRG}(a_{j,i}) - \sum_{j\in\mathcal{U}\setminus\mathcal{D}:i<j} \mathbf{PRG}(a_{i,j}) \right) \pmod{p}. \tag{4.6}$$

Protocol Example

The author explains SecAgg through a simple example as illustrated in Fig. 4.1. There are $n = 4$ users, where user 2 is a semi-honest user ($t = 1$) and user 3 and user 4 are dropout users ($r = 2, \mathcal{D} = \{3\}$). In this example, the communication cost for each user, which is the advantage of SecAgg, is m, whereas the computation cost for the server, which is the disadvantage of SecAgg, is $6m$. SecAgg consists of the following three phases.

Preparation Phase. In this phase, each pair of users (i, j) agrees on a pairwise random seed $a_{i,j}$. In addition, each user i secretly shares its private key sk_i and its self-random seed b_i. Specifically, user $i \in \{1,2,3,4\}$ generates shares of sk_i ($\{(j, sh_j^{sk_i})\}_{j\in\{1,2,3,4\}}$) and b_i ($\{(j, sh_j^{b_i})\}_{i\in\{1,2,3,4\}}$) through **SS.share** of 2-out-of-4 Shamir's secret sharing and sends $sh_{i,j} = (sh_j^{sk_i}, sh_j^{b_i})$ to each user j.

Local Model Masking Phase. In this phase, each user i masks its local models with the two types of random masks and sends its masked local model. Specifically, each user $i \in \{1,2,4\}$ generates its

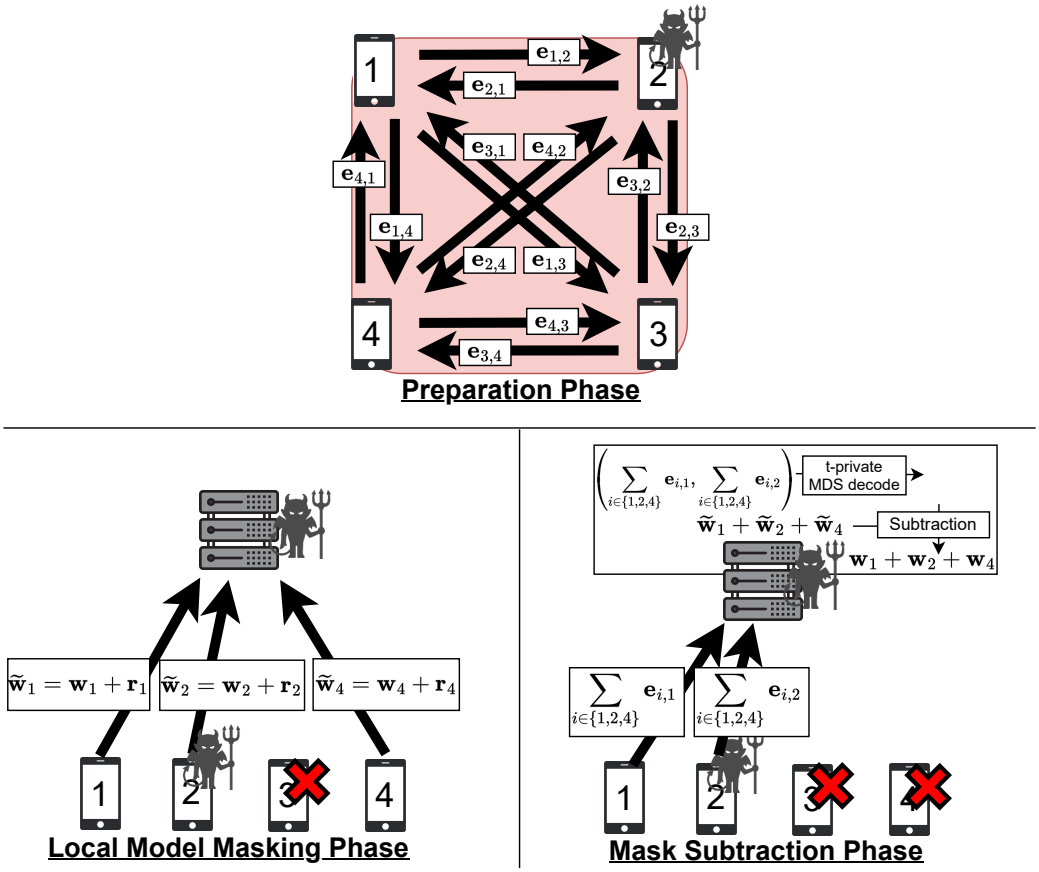

Figure 4.2: An example of LightSecAgg with $n = 4$ users, where user 2 is a semi-honest user ($t = 1$) and user 3 and user 4 are dropout users ($r = 2, \mathcal{D} = \{3\}$). Thin and thick lines indicate scalars and vectors, respectively.

masked local model $\widetilde{\mathbf{w}}_i$ as follows:

$$\begin{aligned}
\widetilde{\mathbf{w}}_1 &= \mathbf{w}_1 + \mathbf{PRG}(b_1) \\
&\quad + \mathbf{PRG}(a_{1,2}) + \mathbf{PRG}(a_{1,3}) + \mathbf{PRG}(a_{1,4}), \\
\widetilde{\mathbf{w}}_2 &= \mathbf{w}_2 + \mathbf{PRG}(b_2) \\
&\quad - \mathbf{PRG}(a_{1,2}) + \mathbf{PRG}(a_{2,3}) + \mathbf{PRG}(a_{2,4}), \\
\widetilde{\mathbf{w}}_4 &= \mathbf{w}_4 + \mathbf{PRG}(b_4) \\
&\quad - \mathbf{PRG}(a_{1,4}) - \mathbf{PRG}(a_{2,4}) - \mathbf{PRG}(a_{3,4}).
\end{aligned}$$

Mask Subtraction Phase. In this phase, the server obtains the sum of the live users' local models in the following steps.

First, each user $i \in \{1, 2\}$ sends the shares of b_1, b_2, b_4, and sk_3 ($\{sh_i^{b_j}\}_{j\in\{1,2,4\}}, sh_i^{sk_3}$) to the server. Next, the server reconstructs b_1, b_2, b_4, sk_3 through **SS**.**recon**, and reconstructs $a_{1,3}, a_{2,3}, a_{3,4}$ using pk_1, pk_2, pk_4, sk_3 through **Key**.**agree**. Then, the server reconstructs $\mathbf{PRG}(b_1)$, $\mathbf{PRG}(b_2)$, $\mathbf{PRG}(b_4)$, $\mathbf{PRG}(a_{1,3})$, $\mathbf{PRG}(a_{2,3})$, and $\mathbf{PRG}(a_{3,4})$ through **PRG**. Finally, the server computes $\sum_{i\in\{1,2,4\}} \mathbf{w}_i = \sum_{i\in\{1,2,4\}} \widetilde{\mathbf{w}}_i - \sum_{i\in\{1,2,4\}} \mathbf{PRG}(b_i) - (\sum_{i\in\{1,2\}} \mathbf{PRG}(a_{i,3}) - \mathbf{PRG}(a_{3,4}))$.

4.2.3 LightSecAgg

Rationale

LightSecAgg avoids the need for numerous random mask reconstructions in SecAgg by introducing two ideas. The first idea is that users solely use self-random masks, unlike SecAgg, which employs both self-random masks and pairwise random masks. Hereafter, the self-random mask is simply referred to as the random mask. The second idea is to eliminate the need for the server to individually reconstruct random masks in the following steps: First, the users secretly share the random masks rather than random seeds. This sharing is performed through the encoding of a t-private Maximum Distance Separable (MDS) code (this code typically refers to Lagrange code [52] and Shamir's secret sharing). The role of the MDS encoding is that any random mask cannot be reconstructed from at most any t encoded masks and that it can be reconstructed even if up to $n - (t + 1)$ encoded masks are lost. Second, each user aggregates the shares locally. This is possible because the MDS codes have linearity with respect to addition like Shamir's secret sharing as explained in Section 4.1.4. As a result, the server reconstructs only one mask, the aggregated random mask.

Technical Intuition

Each user i generates a random mask $\mathbf{r}_i$ and adds it to its local model as follows:

$$\widetilde{\mathbf{w}}_i = \mathbf{w}_i + \mathbf{r}_i \pmod{p}, \tag{4.7}$$

and sends $\widetilde{\mathbf{w}}_i$ to the server.

After the server collects the masked local models of all live users, the server subtracts the sum of the random masks of all the live users ($\sum_{i\in\mathcal{U}\setminus\mathcal{D}} \mathbf{r}_i$), which is an aggregated random mask, from the sum of their masked local models.

The reconstruction of the aggregated random mask is securely performed as follows. First, each user i generates multiple encoded masks $\{\mathbf{e}_{i,j}\}_{j\in\mathcal{U}}$ of its random mask $\mathbf{r}_i$ through a t-private MDS encoding algorithm. The encoding of $\mathbf{r}_i$ is performed element-wise. Next, each user i sends an encoded mask $\mathbf{e}_{i,j}$ to each other user $j \in \mathcal{U} \setminus \{i\}$. Then, each user i aggregates the received encoded masks of all the live users and sends the aggregated encoded mask $\sum_{j\in\mathcal{U}\setminus\mathcal{D}} \mathbf{e}_{j,i}$ to the server. Since each $\sum_{j\in\mathcal{U}\setminus\mathcal{D}} \mathbf{e}_{j,i}$ can be regarded as an encoded mask of $\sum_{i\in\mathcal{U}\setminus\mathcal{D}} \mathbf{r}_i$, the server collects at least $t+1$ aggregated encoded masks and reconstructs $\sum_{i\in\mathcal{U}\setminus\mathcal{D}} \mathbf{r}_i$ from the collected aggregated encoded masks through a t-private MDS decoding algorithm.

Eventually, the server obtains the sum of the local models of all the live users by computing

$$\sum_{i\in\mathcal{U}\setminus\mathcal{D}} \mathbf{w}_i = \sum_{i\in\mathcal{U}\setminus\mathcal{D}} \widetilde{\mathbf{w}}_i - \sum_{i\in\mathcal{U}\setminus\mathcal{D}} \mathbf{r}_i \pmod{p}. \tag{4.8}$$

Protocol Example

As illustrated in Fig. 4.2, the author explains LightSecAgg through a simple example in the same setting as SecAgg. In this example, the computation cost for the server, which is the advantage of LightSecAgg, is m, whereas the communication cost for each user, which is the disadvantage of LightSecAgg, is $5m$.

Preparation Phase. In this phase, each user $i \in \{1,2,3,4\}$ generates its random mask $\mathbf{r}_i$, and four encoded masks $\{\mathbf{e}_{i,j}\}_{j\in\{1,2,3,4\}}$ of $\mathbf{r}_i$ through a t-private MDS encoding. Then, each user i sends $\mathbf{e}_{i,j}$ to each other user $j \in \{1,2,3,4\} \setminus \{i\}$.

Local Model Masking Phase. In this phase, each user $i \in \{1,2,4\}$ generates its masked local model $\widetilde{\mathbf{w}}_i = \mathbf{w}_i + \mathbf{r}_i$ and sends it to the server.

Mask Subtraction Phase. In this phase, each user $i \in \{1,2\}$ aggregates the received encoded masks of all the live users and sends $\sum_{j\in\{1,2,4\}} \mathbf{e}_{j,i}$ to the server. Then, the server reconstructs $\sum_{i\in\{1,2,4\}} \mathbf{r}_i$ from $\{\sum_{j\in\{1,2,4\}} \mathbf{e}_{j,i}\}_{i\in\{1,2\}}$ through a t-private MDS decoding, and obtains $\sum_{i\in\{1,2,4\}} \mathbf{w}_i$ by computing $\sum_{i\in\{1,2,4\}} \mathbf{w}_i = \sum_{i\in\{1,2,4\}} \widetilde{\mathbf{w}}_i - \sum_{i\in\{1,2,4\}} \mathbf{r}_i$.

4.3 BalancedSecAgg

4.3.1 Overview

SecAgg achieves a low communication cost for each user by exchanging seeds rather than masks at the expense of a server's high computation cost of reconstructing masks of dropped users. In contrast,

LightSecAgg reduces such a server's mask reconstruction cost by exchanging masks and redundant masks at the expense of a communication cost. BlancedSecAgg is designed, inspired by the positive sides of SecAgg and LightSecAgg, that is, seed exchanges and redundant mask exchanges.

The key idea is that seeds are exchanged rather than t masks, and that $n - t - 1$ redundant random masks themselves are exchanged. This reduces the communication cost of t masks, which is a problem of LightSecAgg by using the idea of SecAgg, while reducing the reconstruction cost of $n - t - 1$ redundant random mask generation. The key contribution of BalancedSecAgg is that the key idea is achieved by adopting Reed-Solomon codes to prepare redundant random masks.

4.3.2 Rationale

Our motivation is to achieve a better balance between communication and computation costs for faster protocol execution compared to SecAgg and LightSecAgg. The key idea is to regard the corresponding elements across n masks (shares) as n symbols composing a codeword in Reed-Solomon erasure code. Each set of n symbols consists of $t + 1$ input symbols and $n - (t + 1)$ redundant symbols. Hereafter, the author refers to a mask containing input symbols as a *random* mask and one containing redundant symbols as a *redundant* mask. Each user adds $t + 1$ random masks and $n - (t + 1)$ redundant masks to its local model and sends this masked local model to the server. Then, all the random masks and redundant masks of the users are exchanged and aggregated among the users. The server collects and aggregates these masks from the users and subtracts the result from the sum of live users' masked local models to compute the sum of the live users' local models. Dropout tolerance is achieved by the Reed-Solomon erasure code.

This idea enables BalancedSecAgg to make SecAgg and LightSecAgg complement each other. BalancedSecAgg has three key advantages. First, the fact that random masks and redundant masks are exchanged and aggregated among users eliminates the need for the server to reconstruct these masks individually. This reduces the computation cost of BalancedSecAgg compared to SecAgg. Second, the users use random seeds for exchanging their random masks. This reduces the communication cost of BalancedSecAgg compared to LightSecAgg. Third, BalancedSecAgg adopts Reed-Solomon erasure codes rather than t-private MDS codes to reduce the actual computation time of encoding/decoding masks. The t-private MDS codes involve Lagrange interpolation, which reconstructs a t-degree polynomial from $t + 1$ evaluation points, and the execution of Lagrange interpolation is a computation bottleneck in LightSecAgg [47].

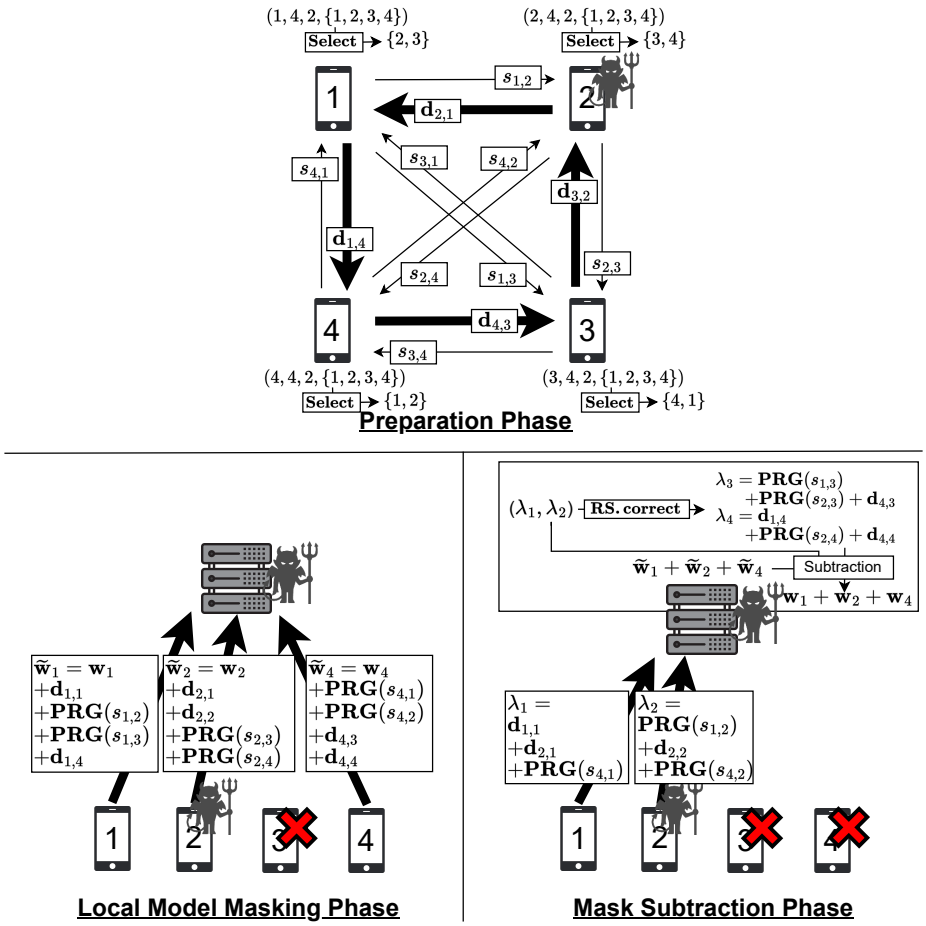

Figure 4.3: An example of BalancedSecAgg with $n = 4$ users, where user 2 is a semi-honest user ($t = 1$) and user 3 and user 4 are dropout users ($r = 2, \mathcal{D} = \{3\}$). Thin and thick lines indicate scalars and vectors, respectively.

4.3.3 Technical Intuition

Each user i generates $t + 1$ random masks $\{\mathbf{PRG}(s_{i,j})\}_{j\in\mathcal{S}_i}$ ($|\mathcal{S}_i| = t + 1$) and $n - (t + 1)$ redundant masks $\{\mathbf{d}_{i,k}\}_{\mathcal{U}\setminus\mathcal{S}_i}$ ($|\mathcal{U} \setminus \mathcal{S}_i| = n - (t + 1)$) through **RS.correct**, denoted by

$$\{\mathbf{d}_{i,k}\}_{k\in\mathcal{U}\setminus\mathcal{S}_i} = \mathbf{RS.correct}(\{\mathbf{PRG}(s_{i,j})\}_{j\in\mathcal{S}_i}, t + 1), \tag{4.9}$$

where $\mathcal{S}_i$ is a subset of users generated according to Algorithm 1. Each user j in $\mathcal{S}_i$ receives a random seed $s_{i,j}$ from i. Algorithm 1 ensures that users' redundant mask exchange is equally loaded, as explained in Section 4.3.4. The generating of each redundant mask is performed element-wise.

Then, each user i generates its masked local model,

$$\widetilde{\mathbf{w}}_i = \mathbf{w}_i + \sum_{j \in \mathcal{S}_i} \mathbf{PRG}(s_{i,j}) + \sum_{j \in \mathcal{U} \setminus \mathcal{S}_i} \mathbf{d}_{i,j} \pmod{p}, \tag{4.10}$$

and sends it to the server.

After the server collects live users' masked local models, the server subtracts the sum of all the random masks and redundant masks of the live users, from the sum of their masked local models.

To this end, each user i sends the random seed $s_{i,j}$ to each user $j \in (\mathcal{U} \setminus \mathcal{D}) \cap \mathcal{S}_i$ and the redundant mask $\mathbf{d}_{i,k}$ to each user $k \in (\mathcal{U} \setminus \mathcal{D}) \setminus \mathcal{S}_i$. Then, each user i aggregates the received masks of all the live users and sends the aggregated mask,

$$\lambda_i = \sum_{j \in (\mathcal{U} \setminus \mathcal{D}) \cap \{k | i \in \mathcal{S}_k\}} \mathbf{PRG}(s_{j,i}) + \sum_{j \in (\mathcal{U} \setminus \mathcal{D}) \setminus \{k | i \in \mathcal{S}_k\}} \mathbf{d}_{j,i} \pmod{p}, \tag{4.11}$$

to the server. Since $t+1$ aggregated masks can be regarded as $t+1$ random masks out of $\{\lambda_j\}_{j \in \mathcal{U}}$, the server collects $\{\lambda_j\}_{j \in \mathcal{U} \setminus \mathcal{D}}$ and reconstructs all the dropout users' aggregated masks through $\mathbf{RS.correct}\left(\{\lambda_k\}_{k \in \mathcal{D}} = \mathbf{RS.correct}(\{\lambda_j\}_{j \in \mathcal{U} \setminus \mathcal{D}}, t+1)\right)$. Finally, the server obtains the sum of local models of the live users $\mathbf{v} = \sum_{i \in \mathcal{U} \setminus \mathcal{D}} \mathbf{w}_i$ by computing

$$\begin{aligned} \mathbf{v} &= \sum_{i \in \mathcal{U} \setminus \mathcal{D}} \widetilde{\mathbf{w}}_i - \sum_{i \in \mathcal{U}} \lambda_i \\ &= \sum_{i \in \mathcal{U} \setminus \mathcal{D}} (\mathbf{w}_i + \sum_{j \in \mathcal{S}_i} \mathbf{PRG}(s_{i,j}) + \sum_{j \in \mathcal{U} \setminus \mathcal{S}_i} \mathbf{d}_{i,j}) \\ &\quad - \sum_{i \in \mathcal{U}} (\sum_{j \in (\mathcal{U} \setminus \mathcal{D}) \cap \{k | i \in \mathcal{S}_k\}} \mathbf{PRG}(s_{j,i}) + \sum_{j \in (\mathcal{U} \setminus \mathcal{D}) \setminus \{k | i \in \mathcal{S}_k\}} \mathbf{d}_{j,i}) \\ &\quad \pmod{p}. \end{aligned} \tag{4.12}$$

4.3.4 Protocol

This subsection describes the protocol of BalancedSecAgg in details according to Fig. 4.4. The protocol involves the setup phase followed by the three phases. The author uses $\mathcal{U}_k \subseteq \mathcal{U}$ to denote the set of users who correctly send messages to the server in phase k and the author has

Algorithm 1 Select algorithm on a user i

```
1: procedure Select(i, n, t + 1, U_1)
2:     S_i ← {}                                          ▷ Initialize
3:     c ← t + 1
4:     for a = 1, 2, ··· , n − 1 do
5:         b ← (i + a)  mod n;
6:         if b ∈ U_1 then
7:             S_i ← S_i ∪ {b}
8:             c ← c − 1
9:             if c = 0 then
10:                break;
11:            end if
12:        end if
13:    end for
14:    return S_i
15: end procedure
```

$\mathcal{U} \supseteq \mathcal{U}_1 \supseteq \mathcal{U}_2 \supseteq \mathcal{U}_3 \supseteq \mathcal{U}_4$. $\mathcal{U}_k \setminus \mathcal{U}_{k+1}$ means that the set of dropout users who correctly send messages in phase k but drop out before sending messages in phase $k+1$. The author also shows a simple example in the same setting as SecAgg and LightSecAgg in Fig. 4.3. In this example, the communication cost for each user reduces to $3m$ compared to LightSecAgg, while the computation cost for the server reduces to $2m$ compared to SecAgg.

Setup Phase. In this phase, all the parties are initialized with the system parameters: the security parameter κ, the number of all users n, the number of semi-honest users t, the number of dropout users $r(= n - (t+1))$, public parameters for the key agreement protocol $pp \leftarrow \mathbf{KA}.\mathbf{param}(\kappa)$, and a finite field $\mathbb{F}_p$ for large prime p. For notational simplicity, the author assumes that each user is assigned a unique logical ID, represented as an integer i in the range from 0 to $n-1$. Each user has a private and authenticated channel with the server.

Furthermore, each user broadcasts its public key to establish private and authenticated channels with the other users. Specifically, each user i generates a key pair $(pk_i, sk_i) \leftarrow \mathbf{KA}.\mathbf{gen}(pp)$ and sends its public key to the server. The server sends the public keys of all the users in $\mathcal{U}_1$ ($\{pk_j\}_{j\in\mathcal{U}_1}$) to each user $j \in \mathcal{U}_1$, where $\mathcal{U}_1$ is a set of users who send their public keys to the server. In the subsequent phases, all the messages between any pair of users (i, j) are sent over the private and authenticated channel using pairwise keys $\mathbf{Key}.\mathbf{agree}(sk_i, pk_j)(= \mathbf{Key}.\mathbf{agree}(sk_j, pk_i))$.

Preparation Phase. In this phase, each user generates its random masks and redundant masks and exchanges these masks with other users.

First, each user i selects a set of users, denoted by $\mathcal{S}_i$, where each user $j \in \mathcal{S}_i$ receives a random seed $s_{i,j}$ according to Algorithm 1. Algorithm 1 enables each user i to select different sets of users who receive the random seeds of i. The input values of Algorithm 1 are i, n, $t+1$, and $\mathcal{U}_1$. Each user i selects the subsequent $t+1$ users (line 4– 13). The output value is the set of selected $t+1$ users. For example, in the leftmost figure in Fig. 4.3, user 1 inputs $i = 1, n = 4, t+1 = 2$, and $\mathcal{U}_1 = \{1, 2, 3, 4\}$ to **Select** and outputs $\mathcal{S}_1 = \{2, 3\}$.

Next, each user i generates random masks $\{\mathbf{PRG}(s_{i,j})\}_{j\in\mathcal{S}_i}$ and redundant masks $\{\mathbf{d}_{i,k}\}_{\mathcal{U}_1\setminus\mathcal{S}_i}$ through **RS**.**correct**. Then, each user i encrypts $s_{i,j}$ of each user $j \in \mathcal{S}_i$ with $\mathbf{Key}.\mathbf{agree}(sk_i, pk_j)$ ($c_{i,j} \leftarrow \mathbf{AE}.\mathbf{enc}(\mathbf{KA}.\mathbf{agree}(sk_i, pk_j), i||j||s_{i,j})$) and encrypts $\mathbf{d}_{i,k}$ of each user $k \in \mathcal{U}_1 \setminus \mathcal{S}_i$ with $\mathbf{Key}.\mathbf{agree}(sk_i, pk_k)$ ($c_{i,k} \leftarrow \mathbf{AE}.\mathbf{enc}(\mathbf{KA}.\mathbf{agree}(sk_i, pk_k), i||k||\mathbf{d}_{i,k})$). Finally, each user i sends the list of ciphertext $\{c_{i,j}\}_{j\in\mathcal{U}_1\setminus\{i\}}$ to the server. The server sends a list of all ciphertexts for i ($\{c_{j,i}\}_{j\in\mathcal{U}_2}$) to each user $i \in \mathcal{U}_2$, where $\mathcal{U}_2$ is a set of users who sends the list of ciphertexts to the server.

Local Model Masking Phase. In this phase, each user sends its masked local model,

$$\widetilde{\mathbf{w}}_i = \mathbf{w}_i + \sum_{j\in\mathcal{S}_i} \mathbf{PRG}(s_{i,j}) + \sum_{j\in\mathcal{U}_1\setminus\mathcal{S}_i} \mathbf{d}_{i,j} \pmod{p}, \tag{4.13}$$

to the server. Then, the server sends a list of live users, denoted by $\mathcal{U}_3$, to each live user $i \in \mathcal{U}_3$.

Mask Subtraction Phase. In this phase, each user sends its aggregated masks, and the server eventually computes the sum of the local models of the live users.

First, each user i decrypts the ciphertext $c_{j,i}$ of each user $j \in \mathcal{U}_3 \cap \{k \mid i \in \mathcal{S}_k\}$ with $\mathbf{KA}.\mathbf{agree}(sk_i, pk_j)$ ($j||i||s_{j,i} \leftarrow \mathbf{AE}.\mathbf{dec}(\mathbf{KA}.\mathbf{agree}(sk_i, pk_j), c_{j,i})$) and the ciphertext $c_{k,i}$ of each user $k \in (\mathcal{U}_3\setminus\{j \mid i \in \mathcal{S}_j\})\setminus\{i\}$ with $\mathbf{KA}.\mathbf{agree}(sk_i, pk_k)$ ($k||i||\mathbf{d}_{k,i} \leftarrow \mathbf{AE}.\mathbf{dec}(\mathbf{KA}.\mathbf{agree}(sk_i, pk_k), c_{k,i})$). Next, each user i generates random masks $\{\mathbf{PRG}(s_{j,i})\}_{j\in\mathcal{U}_3\cap\{k|i\in\mathcal{S}_k\}}$ from $\{s_{j,i}\}_{j\in\mathcal{U}_3\cap\{k|i\in\mathcal{S}_k\}}$. Then, each user i computes its aggregated masks λ_i by aggregating all the received masks as follows:

$$\lambda_i = \sum_{j\in\mathcal{U}_3\cap\{k|i\in\mathcal{S}_k\}} \mathbf{PRG}(s_{j,i}) + \sum_{j\in\mathcal{U}_3\setminus\{k|i\in\mathcal{S}_k\}} \mathbf{d}_{j,i} \pmod{p}. \tag{4.14}$$

Finally, each user i sends λ_i to the server.

After the server collects the aggregated masks of at least $t+1$ users, and reconstructs $\{\lambda_k\}_{k\in\mathcal{U}_1\setminus\mathcal{U}_4}$

through **RS**.**correct** $(\{\lambda_k\}_{k\in\mathcal{U}_1\setminus\mathcal{U}_4} = \mathbf{RS}.\mathbf{correct}(\{\lambda_j\}_{j\in\mathcal{U}_4}, t+1))$, where $\mathcal{U}_4$ is a set of users who send their aggregated masks. Finally, the server computes $\mathbf{v} = \sum_{i\in\mathcal{U}_3} \mathbf{w}_i$ by computing.

$$\begin{aligned}\mathbf{v} &= \sum_{i\in\mathcal{U}_3} \widetilde{\mathbf{w}}_i - \sum_{i\in\mathcal{U}_1} \lambda_i \\ &= \sum_{i\in\mathcal{U}_3} (\mathbf{w}_i + \sum_{j\in\mathcal{S}_i} \mathbf{PRG}(s_{i,j}) + \sum_{j\in\mathcal{U}_1\setminus\mathcal{S}_i} \mathbf{d}_{i,j}) \\ &\quad - \sum_{i\in\mathcal{U}_1} (\sum_{j\in\mathcal{U}_3\cap\{k|i\in\mathcal{S}_k\}} \mathbf{PRG}(s_{j,i}) + \sum_{j\in\mathcal{U}_3\setminus\{k|i\in\mathcal{S}_k\}} \mathbf{d}_{j,i}) \\ &\quad \pmod p. \end{aligned} \tag{4.15}$$

4.4 Security Analysis

In this section, the author proves that our protocol achieves privacy preservation as described in Section 3.3. The author proves that the information shared by the server and up to t semi-honest users (i.e., their joint view) does not leak any information about the inputs (local models) of other users, beyond what can be inferred from the output of the computation (a sum of local models). The author introduces some notations to use for the proof.

In the proposed protocol, the view of a party consists of its internal state (its input, randomness, and public parameters) and all messages that this party received from other parties. If this party aborts, it stops receiving messages. The author uses S to denote the server and $\mathbf{w}_{\mathcal{U}'} = \{\mathbf{w}_i\}_{i\in\mathcal{U}'}$ to denote the set of local models of a subset of users $\mathcal{U}' \subseteq \mathcal{U}$. Given a subset of users $\mathcal{T} \subset \mathcal{U}$, The author uses $\mathrm{REAL}_{\mathcal{T}}^{\mathcal{U},t,\kappa}(\mathbf{w}_{\mathcal{U}}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4)$ to denote a random variable representing the joint view of all parties in $\mathcal{T} \cup \{S\}$ in our protocol execution, where the randomness is over the internal randomness of all parties in the setup phase.

In the following theorem, the author proves that the joint view of any group of t semi-honest users and the server can be simulated by using only the inputs of the users in that group, and the sum of inputs of the remaining users. Intuitively, this means that these users and the server learn nothing other than the sum of their own inputs and the sum of the inputs of the other users. Additionally, the author proves that if too many users abort before completing the local model masking phase, the author can simulate the joint view of that group without any information about the inputs of the remaining users.

Theorem 1. *There exists a probabilistic polynomial time (PPT) simulator SIM such that for all t, $\mathbf{w}_{\mathcal{U}}$, $\mathcal{U}_1$, $\mathcal{U}_2$, $\mathcal{U}_3$, $\mathcal{U}_4$, and $\mathcal{T}$ such that $\mathcal{T} \subset \mathcal{U}$, $|\mathcal{T} \setminus \{S\}| \leq t$, $\mathcal{U} \supseteq \mathcal{U}_1 \supseteq \mathcal{U}_2 \supseteq \mathcal{U}_3 \supseteq \mathcal{U}_4$, the output of* SIM *is computationally indistinguishable from the joint view* $\mathrm{REAL}_{\mathcal{T}}^{\mathcal{U},t,\kappa}$ *of the server and the set of semi-honest users $\mathcal{T}$, i.e.,*

$$\begin{aligned}&\mathrm{REAL}_{\mathcal{T}}^{\mathcal{U},t,\kappa}(\mathbf{w}_{\mathcal{U}}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4)\\ \approx&\mathrm{SIM}_{\mathcal{T}}^{\mathcal{U},t,\kappa}(\mathbf{w}_{\mathcal{T}}, \mathbf{v}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4),\end{aligned}$$

where

$$\mathbf{v} = \begin{cases} \sum_{i \in \mathcal{U}_3 \setminus \mathcal{T}} \mathbf{w}_i & \text{if } |\mathcal{U}_3| \geq t + 2, \\ \perp & \text{otherwise.} \end{cases}$$

Proof. The author proves Theorem 1 by a standard hybrid argument. The author defines SIM through a series of modifications from REAL such that the random variables are computationally indistinguishable.

Hybrid$_0$: This random variable is the same as REAL.

Hybrid$_1$: In this hybrid, for all the honest users $i \in \mathcal{U}_2 \setminus \mathcal{T}$, the author replaces $\mathbf{KA}.\mathbf{agree}(sk_i, pk_j)$ for encrypting and decrypting messages to the other honest users $j \in \mathcal{U}_1 \setminus \mathcal{T}$ with a uniformly random key selected by the simulator. The DDH assumption guarantees that this hybrid is indistinguishable from **Hybrid$_0$**.

Hybrid$_2$: In this hybrid, for all the honest users $i \in \mathcal{U}_2 \setminus \mathcal{T}$, the author replaces all ciphertexts of random seeds and redundant masks encrypted by i and sent to the other honest users $j \in \mathcal{U}_1 \setminus \mathcal{T}$ with encryptions of random values or random masks (e.g., 0 or m-dimensional zero vector). However, the honest users in that set continue to use the correct random seeds and redundant masks in the mask subtraction phase. IND-CPA security guarantees that this hybrid is indistinguishable from **Hybrid$_1$**.

Hybrid$_3$: Define

$$\mathcal{U}^* = \begin{cases} \mathcal{U}_2 \setminus \mathcal{T} & \text{if } \mathbf{v} = \perp, \\ \mathcal{U}_2 \setminus \mathcal{U}_3 \setminus \mathcal{T} & \text{otherwise.} \end{cases}$$

In this hybrid, for all the honest users $i \in \mathcal{U}^*$, the author replaces $\mathbf{PRG}(s_{i,j})$, shared with the other honest users $j \in \mathcal{S}_i \setminus \mathcal{T}$, with a uniformly random mask $\mathbf{u}_{i,j}$. Security of **PRG** ensures that this

hybrid is indistinguishable from **Hybrid$_2$**.

Hybrid$_4$: In this hybrid, for all the honest users $i \in \mathcal{U}^*$, the author replaces

$$\widetilde{\mathbf{w}}_i = \mathbf{w}_i + \sum_{j \in \mathcal{S}_i \setminus \mathcal{T}} \mathbf{u}_{i,j} + \sum_{j \in \mathcal{S}_i \cap \mathcal{T}} \mathbf{PRG}(s_{i,j}) + \sum_{j \in \mathcal{U}_1 \setminus \mathcal{S}_i} \mathbf{d}_{i,j},$$

with

$$\widetilde{\mathbf{w}}_i = \sum_{j \in \mathcal{S}_i \setminus \mathcal{T}} \mathbf{u}_{i,j} + \sum_{j \in \mathcal{S}_i \cap \mathcal{T}} \mathbf{PRG}(s_{i,j}) + \sum_{j \in \mathcal{U}_1 \setminus \mathcal{S}_i} \mathbf{d}_{i,j}.$$

Since $\mathbf{w}_i + \sum_{j \in \mathcal{S}_i \setminus \mathcal{T}} \mathbf{u}_{i,j}$ is uniformly random, this hybrid and **Hybrid$_3$** are identically distributed. Furthermore, this hybrid and all the subsequent hybrids do not depend on $\mathbf{w}_i$ for $i \in \mathcal{U}^*$.

If $\mathbf{v}$ is $\perp$, let SIM be as described in **Hybrid$_4$**, SIM can simulate REAL without knowing $\mathbf{w}_i$ for all the honest users $i \in \mathcal{U} \setminus \mathcal{T}$. Therefore, the author assumes $\mathbf{v} \neq \perp$ in the following hybrids.

Hybrid$_5$: In this hybrid, for all the honest users $i \in \mathcal{U}_2 \setminus \mathcal{T}$, the author replaces

$$\{\mathbf{d}_{i,k}\}_{k \in \mathcal{U}_1 \setminus \mathcal{S}_i} = \mathbf{RS.correct}(\{\mathbf{PRG}(s_{i,j})\}_{j \in \mathcal{S}_i}, t+1)$$

with

$$\{\mathbf{q}_{i,k}\}_{k \in \mathcal{U}_1 \setminus \mathcal{S}_i} = \mathbf{RS.correct}(\{\mathbf{u}_{i,j}\}_{j \in \mathcal{S}_i \setminus \mathcal{T}} \cup \{\mathbf{PRG}(s_{i,j})\}_{j \in \mathcal{S}_i \cap \mathcal{T}}, t+1),$$

where each $\mathbf{u}_{i,j}$ is a uniformly random mask. Note that both the two sets of $|\mathcal{U}_1|$ masks, $\mathcal{M}_i^1 = \{\mathbf{d}_{i,k}\}_{k \in \mathcal{U}_1 \setminus \mathcal{S}_i} \cup \{\mathbf{PRG}(s_{i,j})\}_{j \in \mathcal{S}_i \setminus \mathcal{T}} \cup \{\mathbf{PRG}(s_{i,j})\}_{j \in \mathcal{S}_i \cap \mathcal{T}}$ and $\mathcal{M}_i^2 = \{\mathbf{q}_{i,k}\}_{k \in \mathcal{U}_1 \setminus \mathcal{S}_i} \cup \{\mathbf{u}_{i,j}\}_{j \in \mathcal{S}_i \setminus \mathcal{T}} \cup \{\mathbf{PRG}(s_{i,j})\}_{j \in \mathcal{S}_i \cap \mathcal{T}}$, can be regarded as sets of $|\mathcal{U}_1|$ shares of (uniformly random) secrets y an y', i.e., the outputs of $\mathbf{SS.Share}(y, t+1, |\mathcal{U}_1|)$ and $\mathbf{SS.Share}(y', t+1, |\mathcal{U}_1|)$. Since the joint view of all the semi-honest users can contain only any t masks, the distribution of any t masks in $\mathcal{M}_i^1$ is identical to the distribution of an equivalent number of masks in $\mathcal{M}_i^2$. Therefore, this hybrid is identically distributed to **Hybrid$_4$**.

Hybrid$_6$: The author fixes a specific user $\ell \in \mathcal{U}_3 \setminus \mathcal{T}$. In this hybrid, all the honest users include the honest user ℓ in the set of selected users. Specifically, for each honest user i who does not send a

random mask to ℓ, $i \in (\mathcal{U}_1 \setminus \mathcal{T}) \cap \{k \mid \ell \notin \mathcal{S}_k\}$, the author replaces $\mathcal{S}_i$ with $\mathcal{S}'_i$ is denoted by

$$\mathcal{S}'_i = \mathcal{S}_i \setminus \{i^{\text{sel}}\} \cup \{\ell\},$$

where i^{sel} is a selected user from $\mathcal{S}_i$. This modification allows for all the honest users in $\mathcal{U}_2 \setminus \mathcal{T}$ to send their random masks to ℓ instead of their encoded masks. Here, let $\mathcal{M}_i^3$ be the resulting set of masks of user i, i.e., $\mathcal{M}_i^3 = \{\mathbf{q}_{i,k}\}_{k \in \mathcal{U}_1 \setminus \mathcal{S}'_i} \cup \{\mathbf{u}_{i,j}\}_{j \in \mathcal{S}'_i \setminus \mathcal{T}} \cup \{\mathbf{PRG}(s_{i,j})\}_{j \in \mathcal{S}'_i \cap \mathcal{T}}$. Since the joint view of all the semi-honest users can contain only any t masks, the distribution of any t masks in $\mathcal{M}_i^2$ is identical to the distribution of an equivalent number of masks in $\mathcal{M}_i^3$. Therefore, this hybrid is identically distributed to **Hybrid$_5$**.

Hybrid$_7$: In this hybrid, for all the honest users $i \in \mathcal{U}_3 \setminus \mathcal{T}$, the author replaces

$$\begin{aligned}\widetilde{\mathbf{w}}_i &= \mathbf{w}_i + \sum_{j \in \mathcal{S}'_i} \mathbf{u}_{i,j} + \sum_{j \in \mathcal{U}_1 \setminus \mathcal{S}'_i} \mathbf{d}_{i,j} \\ &= \mathbf{w}_i + \mathbf{u}_{i,\ell} + \sum_{j \in \mathcal{S}'_i \setminus \{\ell\}} \mathbf{u}_{i,j} + \sum_{j \in \mathcal{U}_1 \setminus \mathcal{S}'_i} \mathbf{d}_{i,j}\end{aligned}$$

with

$$\widetilde{\mathbf{w}}'_i = \mathbf{x}_i + \sum_{j \in \mathcal{S}'_i \setminus \{\ell\}} \mathbf{u}_{i,j} + \sum_{j \in \mathcal{U}_1 \setminus \mathcal{S}'_i} \mathbf{d}_{i,j},$$

where $\{\mathbf{x}_i\}_{i \in \mathcal{U}_3 \setminus \mathcal{T}}$ are uniformly random, subject to $\sum_{i \in \mathcal{U}_3 \setminus \mathcal{T}} \mathbf{x}_i = \sum_{i \in \mathcal{U}_3 \setminus \mathcal{T}} (\mathbf{w}_i + \mathbf{u}_{i,\ell})$. Thus, this hybrid is identically distributed to **Hybrid$_6$**. Let SIM be as described in **Hybrid$_7$**, SIM can simulate REAL without knowing $\mathbf{w}_i$ for all the honest users $i \in \mathcal{U} \setminus \mathcal{T}$. Since the argument above proves that the output of SIM is computationally indistinguishable from the output of REAL, completing the proof. □

4.5 Performance Analysis

The author measures the execution time of the secure aggregation protocol of one round of federated learning which the author calls the protocol running time. The protocol running time is defined as the time from when each user, who generates its local model, starts random masks for local model masking to the time when the server finishes computing the sum of local models. The protocol running time is the sum of the computation time and the communication time. Since users' computations are performed in parallel, the computation time is the sum of each user's computation

Protocol	m, n / r	m = 1M			m = 2M			m = 3M		
		n = 50	n = 100	n = 150	n = 50	n = 100	n = 150	n = 50	n = 100	n = 150
SecAgg	$r = 0.1n$	10.07	33.99	71.55	19.9	68.35	145.32	30.18	102.67	217.73
	$r = 0.2n$	15.05	54.89	121.1	30.75	112.23	243.55	46.27	167.47	364.35
	$r = 0.3n$	18.96	69.75	153.91	38.19	142.84	312.97	57.818	213.52	467.68
LightSecAgg	$r = 0.1n$	13.58	50.05	108.52	27.06	98.91	216.48	40.86	149.45	326.08
	$r = 0.2n$	11.87	43.31	93.54	23.41	86.69	188.7	31.62	114.44	282.26
	$r = 0.3n$	9.04	32.77	74.95	18.17	65.14	159.03	26.65	95.92	237.08
BalancedSecAgg	$r = 0.1n$	3.25	6.7	10.14	6.39	13.13	19.99	9.91	20.41	31.29
	$r = 0.2n$	3.17	6.45	9.87	5.85	12.8	21.51	9.36	19.22	32.69
	$r = 0.3n$	2.79	6.01	9.52	5.67	11.81	19.13	8.78	18.47	30.22

Table 4.2: Computation time of SecAgg, LightSecAgg, and BalancedSecAgg with parameters the number of all users n, the size of models m, and the number of dropout users r.

time and the server's computation time. In contrast, the communication time is the time for each user's sending and receiving seeds, masks, and redundant masks and sending an aggregated mask to the server. Since shortening the execution time of secure aggregation leads to a reduction in the total execution time of one round of federated learning, the author focuses on accelerating secure aggregation to improve the execution time of one round of federated learning.

The protocol running time is computed by simulating each protocol behavior using the measured values of computation time and communication time.

4.5.1 Measurement Method

The author implements the code in C/C++ and run the code on a Linux workstation with Intel Core i9 19-10900X CPU (3.70GHz), NVIDIA GeForce RTX 3090 GPU, and 64GB of RAM. Each parameter in the local model is a 32-bit entry and the field size, p, is set as a prime with 32 bits. Furthermore, the author implements Reed-Solomon erasure code in Jerasure library [60] which leverages Simple Instruction Multiple Data (SIMD) instructions.

Table 4.2, Table 4.3, and Table 4.4 show the computation time, the communication time, and the total time (protocol running time) of the three protocols, respectively, with varying the following parameters:

- n: the number of all users. The number of users is a variable for evaluating the scalability of a federated learning system.
- m: the size of vectors(models). For mobile applications, the author assumes using models such as SqueezeNet [61] and MobileNetV3 [62], which have up to several tens of MB in size.
- r: the number of dropout users. The author sets $r = 0.1n$, $r = 0.2n$, and $r = 0.3n$. In the realistic setting, dropout rates range between 0.06 and 0.1 [63], so it is reasonable to assume

Protocol	m, n / r	m = 1M			m = 2M			m = 3M		
		n = 50	n = 100	n = 150	n = 50	n = 100	n = 150	n = 50	n = 100	n = 150
SecAgg	$r = 0.1n$	0.32 (0.03)	0.32 (0.03)	0.32 (0.03)	0.65 (0.07)	0.65 (0.07)	0.65 (0.07)	0.97 (0.11)	0.98 (0.11)	0.98 (0.11)
	$r = 0.2n$	0.32 (0.03)	0.32 (0.03)	0.32 (0.03)	0.65 (0.07)	0.65 (0.07)	0.65 (0.07)	0.97 (0.11)	0.98 (0.11)	0.98 (0.11)
	$r = 0.3n$	0.32 (0.03)	0.32 (0.03)	0.32 (0.03)	0.65 (0.07)	0.65 (0.07)	0.65 (0.07)	0.97 (0.11)	0.98 (0.11)	0.98 (0.11)
LightSecAgg	$r = 0.1n$	32.65 (3.99)	65.3 (7.98)	97.95 (11.97)	65.3 (7.98)	130.61 (15.96)	195.91 (23.94)	97.95 (11.97)	195.91 (23.94)	293.87 (35.91)
	$r = 0.2n$	32.65 (3.99)	65.3 (7.98)	97.95 (11.97)	65.3 (7.98)	130.61 (15.96)	195.91 (23.94)	97.95 (11.97)	195.91 (23.94)	293.87 (35.91)
	$r = 0.3n$	32.65 (3.99)	65.3 (7.98)	97.95 (11.97)	65.3 (7.98)	130.61 (15.96)	195.91 (23.94)	97.95 (11.97)	195.91 (23.94)	293.87 (35.91)
BalancedSecAgg	$r = 0.1n$	3.26 (0.39)	6.53 (0.79)	9.79 (1.19)	6.53 (0.79)	13.06 (1.59)	19.59 (2.39)	9.79 (1.19)	19.59 (2.39)	29.38 (3.59)
	$r = 0.2n$	6.53 (0.79)	13.06 (1.59)	19.59 (2.39)	13.06 (1.59)	26.12 (3.19)	39.18 (4.78)	19.59 (2.39)	39.18 (4.78)	58.77 (7.18)
	$r = 0.3n$	9.79 (1.19)	19.59 (2.39)	29.38 (3.59)	19.59 (2.39)	39.18 (4.78)	58.77 (7.18)	29.38 (3.59)	58.77 (7.18)	88.16 (10.77)

Table 4.3: Communication time of SecAgg, LightSecAgg, and BalancedSecAgg with parameters the number of all users n, the size of models m, the number of dropout users r, and end-to-end throughput a. In each cell, values outside parentheses correspond to cases where a = 98M, while values inside parentheses represent cases with a = 802M.

that dropout users are a minority.

- a: the end-to-end throughput (bps) between each user and a server. Similar to LightSecAgg's experiment, the author assumes 4G and 5G networks with throughputs of 98Mbps and 802Mbps, respectively, based on realistic settings [64]. It is assumed here that the end-to-end throughput between all users and the server is the same.

4.5.2 Result

In this subsection, the author explains the observations from the experimental results shown in Table 4.2, Table 4.3, and Table 4.4.

Computation time

As shown in Table 4.2, in SecAgg, the computation time is dominated by **PRG** for the reconstruction of $(n - r) + r(n - r)$ random masks on the server. As the values of m, n, and r increase, the computation time of SecAgg increases according to the server's computation cost. In LightSecAgg, the computation time is dominated by the Lagrange interpolation in t-private MDS decoding for mask reconstruction on the server. Since the computation complexity of Lagrange interpolation for generating each element of the sum of random masks is $O(t^2)$, the computation time of LightSecAgg

Protocol	m, n / r	m = 1M			m = 2M			m = 3M		
		n = 50	n = 100	n = 150	n = 50	n = 100	n = 150	n = 50	n = 100	n = 150
SecAgg	$r = 0.1n$	10.39 (10.10)	34.31 (34.02)	71.87 (71.58)	20.55 (19.97)	69 (68.42)	145.97 (145.39)	31.15 (30.29)	103.65 (102.78)	218.71 (217.84)
	$r = 0.2n$	15.37 (15.08)	55.21 (54.92)	121.42 (121.13)	31.4 (30.82)	112.88 (112.3)	244.2 (243.62)	47.24 (46.38)	168.45 (167.58)	365.33 (364.46)
	$r = 0.3n$	19.28 (18.99)	70.07 (69.78)	154.23 (153.94)	38.84 (38.26)	143.49 (142.91)	313.62 (313.04)	58.79 (57.93)	214.5 (213.63)	468.66 (467.79)
LightSecAgg	$r = 0.1n$	46.23 (17.57)	115.35 (58.03)	206.47 (120.49)	92.36 (35.04)	229.52 (114.87)	412.39 (240.42)	138.81 (52.83)	345.36 (173.39)	619.95 (361.99)
	$r = 0.2n$	44.52 (15.86)	108.61 (51.29)	191.49 (105.51)	88.71 (31.39)	217.3 (102.65)	384.61 (212.64)	129.57 (43.59)	310.35 (138.38)	576.13 (318.17)
	$r = 0.3n$	41.69 (13.03)	98.07 (40.75)	172.9 (86.92)	83.47 (26.15)	195.75 (81.1)	354.94 (182.97)	124.6 (38.62)	291.83 (119.86)	530.95 (272.99)
BalancedSecAgg	$r = 0.1n$	6.51 (3.64)	13.23 (7.49)	19.93 (11.33)	12.92 (7.18)	26.19 (14.72)	39.58 (22.38)	19.7 (11.1)	40 (22.8)	60.67 (34.88)
	$r = 0.2n$	9.7 (3.96)	19.51 (8.04)	29.46 (12.26)	18.91 (7.44)	38.92 (15.99)	60.69 (26.29)	28.95 (11.75)	58.4 (24)	91.46 (39.87)
	$r = 0.3n$	12.58 (3.98)	25.6 (8.4)	38.9 (13.11)	25.26 (8.06)	50.99 (16.59)	77.9 (26.31)	38.16 (12.37)	77.24 (25.65)	118.38 (40.99)

Table 4.4: Protocol running time of SecAgg, LightSecAgg, and BalancedSecAgg with parameters the number of all users n, the size of models m, the number of dropout users r, and end-to-end throughput a. In each cell, values outside parentheses correspond to cases where a = 98M, while values inside parentheses represent cases with a = 802M.

increases with the increase in m and n. On the other hand, as the value of r increases, the value of t decreases. This results in a decrease in the computation time of LightSecAgg. In BalancedSecAgg, the computation time is dominated by **PRG** for the reconstruction of t random masks on each user. While the computation time of BalancedSecAgg increases with the increase in m and n, it decreases with the increase in r.

As shown in Table 4.2, the computation time of BalancedSecAgg is faster than that of SecAgg. This is because the computation cost for the server in BalancedSecAgg is smaller than that in SecAgg, as shown in Table 1.1. In contrast, despite the fact that BalancedSecAgg has a higher computation cost for the server compared to LightSecAgg, BalancedSecAgg outperforms LightSecAgg in terms of actual computation time. This performance advantage is attributed to the use of SIMD instructions in the Jerasure library, which enables element-wise encoding/decoding of masks in parallel, significantly enhancing computational efficiency. Please note that SecAgg cannot use SIMD instructions to execute element-wise random mask generation through **PRG** in parallel, because each random number depends on the previous one. On the other hand, LightSecAgg, similar to BalancedSecAgg, might be able to use SIMD instructions for element-wise t-private MDS decoding in parallel. However, as shown in the next subsection, LightSecAgg incurs longer communication time.

When the server has multiple cores, the computation time for SecAgg and LightSecAgg decreases.

Specifically, when the server has e cores, the computation time for both SecAgg and LightSecAgg reduces to approximately $\frac{1}{e}$. On the other hand, since the computation time for BalancedSecAgg depends on the user's computation, it decreases only slightly even if the server has multiple cores. Therefore, in Table 4.2 indicating the case for $e = 1$, if the computation time for SecAgg and LightSecAgg is less than e times that of BalancedSecAgg, the computation time for SecAgg and LightSecAgg is faster when the number of the server's cores is e.

Communication time

As shown in Table 4.3, the communication time of BalancedSecAgg is faster than that of LightSecAgg, but slower than that of SecAgg. Specifically, given the number of dropout users r, the communication time of BalancedSecAgg is approximately $2r$ times that of SecAgg and approximately $\frac{r}{n}$ times that of LightSecAgg. This is due to the communication cost for each user as shown in Table 1.1. Please note that the number of message transfers from each user to every other user effectively doubles in our communication model where messages between users are mediated through a server. This is because each message is first sent to the server and then forwarded from the server to the recipient.

Protocol running time

As shown in Table 4.4, the protocol running time of BalancedSecAgg is always faster than that of LightSecAgg. This is because the computation and communication times of BalancedSecAgg are always faster than those of LightSecAgg. On the other hand, when values for n, m, and r are fixed, whether SecAgg or BalancedSecAgg is faster is determined by the end-to-end throughput. This is because the computation time of BalancedSecAgg is faster than that of SecAgg, but the communication time of BalancedSecAgg is slower than that of SecAgg. The higher the throughput, the better the performance of BalancedSecAgg. Table 4.4 shows that the running time of BalancedSecAgg is about 11.41× faster than that of SecAgg in the case that $n = 150$, $m = 3$M, $r = 0.3n$, and $a = 802$M, and about 10.74× faster than that of LightSecAgg in the case that $n = 150$, $m = 2$M, $r = 0.1n$, and $a = 802$M.

However, as explained in 4.5.2, when the server has multiple cores, the computation time of SecAgg and LightSecAgg may be faster than that of BalancedSecAgg. In this case, since the communication time of SecAgg is always faster than the communication time of BalancedSecAgg, the protocol running time of SecAgg is faster than the protocol running time of BalancedSecAgg. On the other hand, since the communication time of BalancedSecAgg is smaller than the communication

time of LightSecAgg, which protocol has the faster running time between LightSecAgg and BalancedSecAgg depends on the end-to-end throughput. In summary, BalancedSecAgg is faster than SecAgg and LightSecAgg when the server has limited computational resources.

4.6 Conclusion

The author designs a new secure aggregation protocol, called BalancedSecAgg, to achieve a good balance between computation and communication costs. The key idea is to leverage Reed-Solomon erasure codes both to hide local models and to achieve tolerance against dropout users. Compared to LightSecAgg, where each user transfers n encoded masks, BalancedSecAgg transfers $n - (t + 1)$ redundant masks, thus reducing each user's communication cost. In contrast, compared to SecAgg, where the server reconstructs $n-r$ random masks individually for each dropped user, BalancedSecAgg reduces the computation cost of the server because the server only reconstructs one aggregate mask for each dropped user. In addition, the Reed-Solomon erasure codes, which have an optimized implementation for mask encoding/decoding, reduce the actual computation time.

An interesting future work is to apply the communication cost reduction techniques of BalancedSecAgg to Byzantine-resilient secure aggregation protocols. Existing Byzantine-resilient secure aggregation protocols [25, 43, 51] have a communication cost of $O(n^2)$, where n is the number of all users since each user sends one share of its local model to each other users like LightSecAgg. A major challenge when applying our communication cost reduction techniques to these protocols is how to verify the information pieces, including shares, sent by Byzantine users.

- **Setup Phase**
  - All parties are given the security parameter κ, the number of all users n, the number of semi-honest users t, the number of dropout users r, key agreement parameter $pp \leftarrow \mathbf{KA.param}(\kappa)$, and a field $\mathbb{F}_p$. Each user has a private and authenticated channel with the server.

  User i:
  - Generate a key pair $(pk_i, sk_i) \leftarrow \mathbf{KA.gen}(pp)$
  - Send pk_i to the server.

  Server:
  - Collect public keys from at least $t+2$ users (denoted by $\mathcal{U}_1 \subseteq \mathcal{U}$, where $\mathcal{U}_1$ is the set of these users). Otherwise, abort.
  - Send a list $\{j, pk_j\}_{j\in\mathcal{U}_1}$ to all the users in $\mathcal{U}_1$.
- **Preparation Phase:**

  User i:
  - Receive the list $\{j, pk_j\}_{j\in\mathcal{U}_1}$ from the server. Assert that $|\mathcal{U}_1| \geq t+2$, otherwise abort.
  - Generate a subset of users, denoted by $\mathcal{S}_i$, where each user j receives a random seed of user i ($\mathcal{S}_i \leftarrow \mathbf{Select}(i, n, t+1, \mathcal{U}_1)$).
  - For each user $j \in \mathcal{S}_i$, generate a random seed $s_{i,j}$.
  - Expand the random seeds $\{s_{i,j}\}_{j\in\mathcal{S}_i}$ using a **PRG** into random masks $\{\mathbf{PRG}(s_{i,j})\}_{j\in\mathcal{S}_i}$.
  - For users in $\mathcal{U}_1 \setminus \mathcal{S}_i$, compute redundant masks $\{\mathbf{d}_{i,k}\}_{k\in\mathcal{U}_1\setminus\mathcal{S}_i} = \mathbf{RS.correct}(\{\mathbf{PRG}(s_{i,j})\}_{j\in\mathcal{S}_i}, t+1)$.
  - For each user $j \in \mathcal{S}_i$, generate $c_{i,j} \leftarrow \mathbf{AE.enc}(\mathbf{KA.agree}(sk_i, pk_j), i||j||s_{i,j})$
  - For each user $k \in \mathcal{U}_1 \setminus \mathcal{S}_i$, generate $c_{i,k} \leftarrow \mathbf{AE.enc}(\mathbf{KA.agree}(sk_i, pk_k), i||k||\mathbf{d}_{i,k})$.
  - Send all the ciphertexts $\{c_{i,j}\}_{j\in\mathcal{U}_1\setminus\{i\}}$ to the server.

  Server:
  - Collect lists of ciphertexts from at least $t+2$ users (denoted by $\mathcal{U}_2 \subseteq \mathcal{U}_1$, where $\mathcal{U}_2$ is the set of these users). Otherwise, abort.
  - Send to each user $i \in \mathcal{U}_2$ a list of all ciphertexts encrypted for j: $\{c_{j,i}\}_{j\in\mathcal{U}_2}$.
- **Local Model Masking Phase:**

  User i:
  - Receive from the server the list $\{c_{j,i}\}_{j\in\mathcal{U}_2}$. If $|\mathcal{U}_2|$ is less than $t+2$, abort.
  - Compute the masked local model $\widetilde{\mathbf{w}}_i = \mathbf{w}_i + \sum_{j\in\mathcal{S}_i} \mathbf{PRG}(s_{i,j}) + \sum_{j\in\mathcal{U}_1\setminus\mathcal{S}_i} \mathbf{d}_{i,j}$.
  - Send $\widetilde{\mathbf{w}}_i$ to the server.

  Server:
  - Collect lists of masked local models from at least $t+2$ users (denoted by $\mathcal{U}_3 \subseteq \mathcal{U}_2$, where $\mathcal{U}_3$ is the set of these users). Otherwise, abort.
  - Send the list of all the live users $\mathcal{U}_3$ to each user $j \in \mathcal{U}_3$.
- **Mask Subtraction Phase:**

  User i:
  - Receive $\mathcal{U}_3$ from the server. If $|\mathcal{U}_3|$ is less than $t+2$, abort.
  - For each user $j \in \mathcal{U}_3 \cap \{k \mid i \in \mathcal{S}_k\}$, decrypt the ciphertext $j||i||s_{j,i} \leftarrow \mathbf{AE.dec}(\mathbf{KA.agree}(sk_i, pk_j), c_{j,i})$.
  - For each user $k \in (\mathcal{U}_3 \setminus \{j \mid i \in \mathcal{S}_j\}) \setminus \{i\}$, decrypt the ciphertext $,k||i||\mathbf{d}_{k,i} \leftarrow \mathbf{AE.dec}(\mathbf{KA.agree}(sk_i, pk_k), c_{k,i})$.
  - Expand the received random seeds $\{s_{j,i}\}_{j\in\mathcal{U}_3\cap\{k|i\in\mathcal{S}_k\}}$ using a **PRG** into random masks $\{\mathbf{PRG}(s_{j,i})\}_{j\in\mathcal{U}_3\cap\{k|i\in\mathcal{S}_k\}}$.
  - Compute $\lambda_i = \sum_{j\in\mathcal{U}_3\cap\{k|i\in\mathcal{S}_k\}} \mathbf{PRG}(s_{j,i}) + \sum_{j\in\mathcal{U}_3\setminus\{k|i\in\mathcal{S}_k\}} \mathbf{d}_{j,i}$.
  - Send λ_i to the server.

  Server:
  - Collect aggregated masks from at least $t+1$ users (denoted by $\mathcal{U}_4 \subseteq \mathcal{U}_3$, where $\mathcal{U}_4$ is the set of these users). Otherwise, abort.
  - Reconstruct the aggregated masks of the users in $\mathcal{U}_1 \setminus \mathcal{U}_4$ by computing $\{\lambda_k\}_{k\in\mathcal{U}_1\setminus\mathcal{U}_4} = \mathbf{RS.correct}(\{\lambda_j\}_{j\in\mathcal{U}_4}, t+1)$.
  - Compute $\mathbf{v} = \sum_{i\in\mathcal{U}_3} \mathbf{w}_i = \sum_{i\in\mathcal{U}_3} \widetilde{\mathbf{w}}_i - \sum_{i\in\mathcal{U}_1} \lambda_i$

Figure 4.4: Detailed description of our protocol. Notably, $|\mathcal{U}_1| \geq t+2$, $|\mathcal{U}_2| \geq t+2$, and $|\mathcal{U}_3| \geq t+2$ avoid the situation where there is only one honest user and all other users are semi-honest.

Chapter 5

Fast Secure Aggregation against Dishonest Adversaries

In this chapter, the author designs a faster secure aggregation called BREA with Server's Verification (BREA-SV) than the existing one called BREA to address the threats in the dishonest model. The rest of the chapter is organized as follows: Section 5.1 describes BREA, which is an existing Byzantine-resilient secure aggregation protocol. Section 5.2 describes our protocol, BREA-SV. Section 5.3 and Section 5.4 analyze BREA-SV in terms of security and performance. Section 5.5 concludes this chapter. In addition to Table 3.1, the main symbols used in this chapter are listed in Table 5.1. The author uses bold and uppercase letters to denote vectors and sets, respectively, and lowercase letters for other purposes such as scalar variables.

5.1 Existing Protocol: BREA

5.1.1 Rationale for Attack Mitigation in Dishonest Model

To mitigate privacy attacks, each user randomly divides its local model into multiple *masked-fragments* (shares) through cryptographic techniques such as additive secret sharing or Shamir's secret sharing [42]. Here, the author calls the process of division *masking*. Such shares are aggregated (added) in the following steps: Each user sends such shares to the other users, it aggregates received shares to an aggregated share, and a server aggregates all the aggregated shares to obtain a sum of all the local models. Here, an important point is that the effect of masking is cancelled by the aggregation and so the server only knows the aggregation result. Adversaries cannot recover the

Table 5.1: Summary of Symbols

Symbol	Description
$\mathbf{s}_i^{\mathcal{U}}$	Aggregated shares in set $\mathcal{U}$ computed by user i
$d_i^{j,k}$	Share distance between user j and k computed by user i
$\|\mathbf{w}_j - \mathbf{w}_k\|^2$	Model distance between user j and k
$\mathbf{C}$	Commitment
$\mathcal{H}$	Set of honest users
Multi-Krum()	Multi-Krum algorithm

local model of any honest user unless they collect a certain number of shares. For example, when shares are generated from a local model through $(t+1, n)$-threshold Shamir's secret sharing, the local model cannot be recovered without at least $t+1$ shares.

To mitigate Byzantine attacks, the following methods detect and remove both types of security attacks as illustrated in Figure. 5.1 and Figure. 5.2: The method against the first type of attack is that the server removes outliers as contaminated models using a distance-based outlier removal mechanism called Multi-Krum [24, 65] (Outlier detection in Figure. 5.1 and Figure. 5.2). Users *securely* compute pairwise model distances by using share distances (Computing distance in Fig. 5.1). The methods against the second type of attack are prepared for both contaminated shares and contaminated share computation results: The former is *share verification* (Share verification in Fig. 5.2) to prevent Byzantine users from injecting contaminated shares. All the users generate *commitments* and mutually verify that the shares are honestly generated using the commitments. The latter is to remove contaminated share distances and aggregated shares generated by Byzantine users by leveraging Reed-Solomon decoding algorithms [45] (Share error-correction in Fig. 5.2). BREA and the author's method called BREA-SV satisfy Byzantine-resilience through a Multi-Krum incorporated protocol that takes users' local models $\{\mathbf{w}_i\}_{i\in\mathcal{U}}$ as inputs, executes as follows: The protocol takes the pairwise model distances as the inputs of Multi-Krum, which is formulated by

$$\mathcal{U}^{sel} = \text{Multi-Krum}(\{\|\mathbf{w}_j - \mathbf{w}_k\|^2\}_{j,k\in\mathcal{U}}), \tag{5.1}$$

where $\mathcal{U}^{sel}(\subset \mathcal{U})$ is a set of a certain number c ($c \leq n - \ell$) users whose local models that are close to each other, c is a pre-determined parameter for ℓ, and $\|\cdot\|^2$ is the Euclidean distance. As a result, the protocol computes $\sum_{i\in\mathcal{U}^{sel}} \mathbf{w}_i$ as an aggregation result.

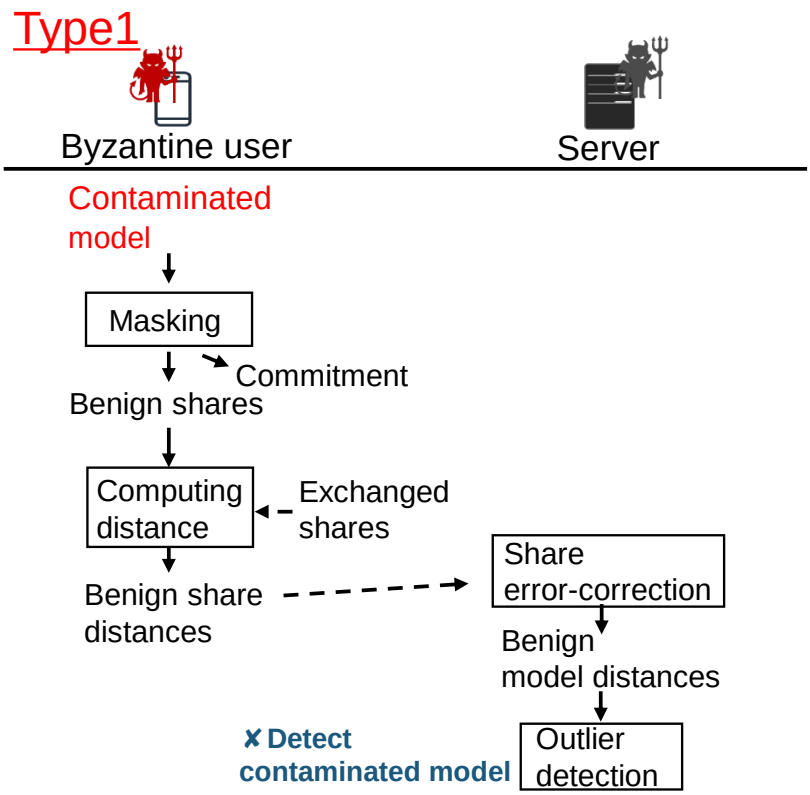

Figure 5.1: First type of Byzantine attacks and mitigation methods

5.1.2 Protocol

BREA consists of three phases as illustrated in Fig. 5.3 and mitigates privacy and security attacks in individual phases.

Secret Sharing Phase

In this phase, each user i distributes shares of its local model $\mathbf{w}_i$ to the other users. Precisely, each user i sends a share $\mathbf{sh}_{i,j}$, denoted by

$$\mathbf{sh}_{i,j} = f_i(\theta_j, \mathbf{w}_i), \tag{5.2}$$

and sends it to each user j, where $f_i : \mathbb{F}_p \rightarrow \mathbb{F}_p^m$ is a t-degree polynomial defined by

$$f_i(\theta, \mathbf{w}_i) = \mathbf{w}_i + \sum_{k=1}^{t} \mathbf{r}_{ik}\theta^k, \tag{5.3}$$

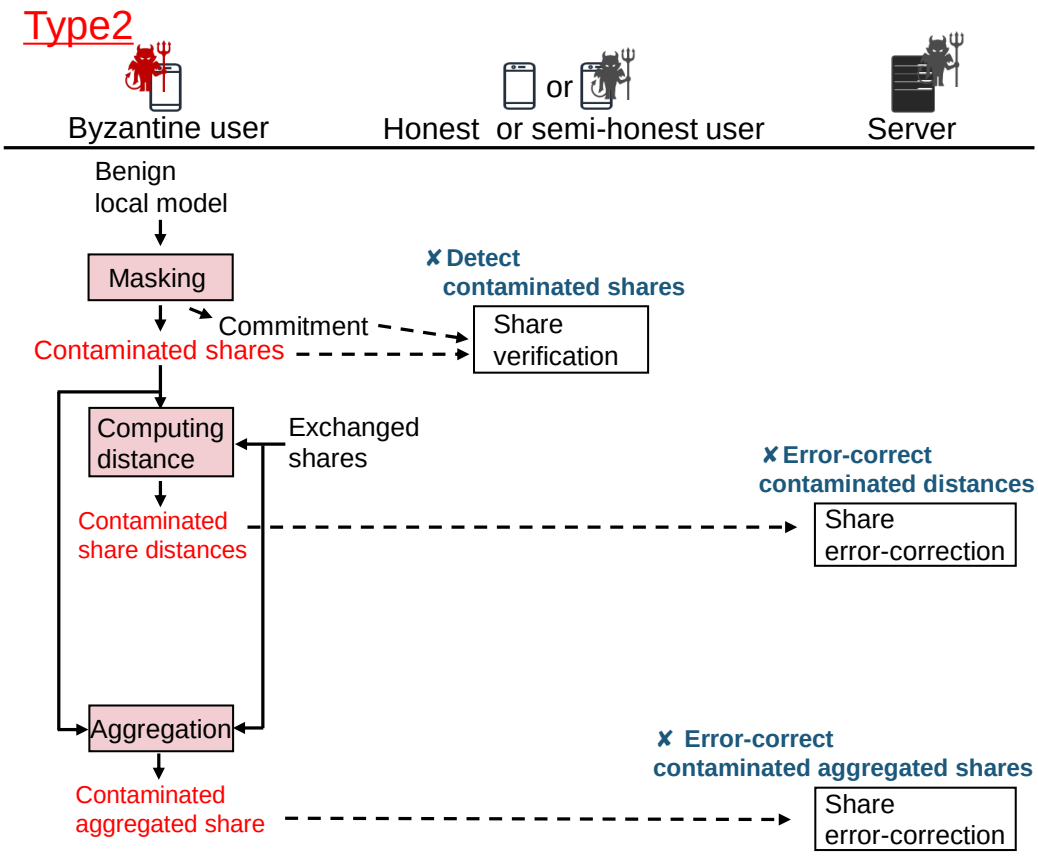

Figure 5.2: Second type of Byzantine attacks and mitigation methods

$\mathbb{F}_p$ is a finite field for a large prime p, $\mathbf{r}_{ik}$ is a random vector chosen from $\mathbb{F}_p^m$, and θ_j is an element chosen from $\mathbb{F}_p$ agreed on in advance by all the users and the server.

Share Verification Each user i *broadcasts* its commitment $\mathbf{C}_{f_i(\theta,\mathbf{w}_i)}$ to the coefficients of the polynomial f_i, which is one-to-one correspondence with f_i, given by

$$\mathbf{C}_{f_i(\theta,\mathbf{w}_i)} = (g^{\mathbf{w}_i}, g^{\mathbf{r}_{i1}}, g^{\mathbf{r}_{i2}}, \cdots, g^{\mathbf{r}_{it}}), \tag{5.4}$$

where g is a generator selected from $\mathbb{F}_p$. Broadcasting plays an important role to guarantee that all honest users receive the same commitments.

Then, each receiver j verifies that the received share from each user i is the correct function

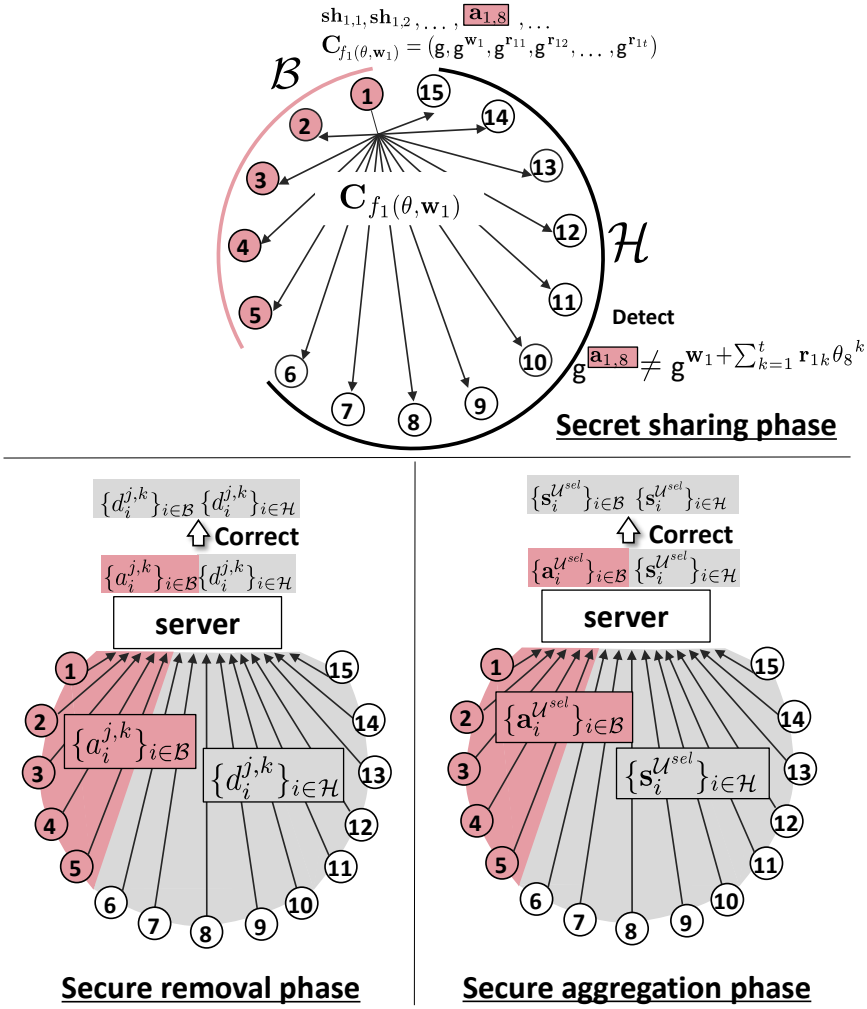

Figure 5.3: Second type of security attacks of Byzantine users to three phases and mitigation methods.

value $\mathbf{sh}_{i,j}(= f_i(\theta_j, \mathbf{w}_i))$ of $f_i(\theta, \mathbf{w}_i)$ by verifying if the following equation holds:

$$g^{\mathbf{sh}_{i,j}} = g^{\mathbf{w}_i + \sum_{k=1}^{t} \mathbf{r}_{ik} \theta_j{}^k}. \tag{5.5}$$

If a Byzantine user b sends an contaminated share $\mathbf{a}_{b,j}(\neq \mathbf{sh}_{b,j})$ to any user j, j detects the attack because (5.5) does not hold. Meanwhile, since BREA assumes the intractability of computing the discrete logarithm, the server and all the users cannot compute $\mathbf{w}_i$ from $g^{\mathbf{w}_i}$.

Secure Removal Phase

In this phase, all the users and the server securely and jointly perform Multi-Krum.

For each pair of the exchanged shares, each user i locally computes a share distance, $d_i^{j,k} := \|\mathbf{sh}_{j,i} - \mathbf{sh}_{k,i}\|^2$, and send it to the server. The server collects at least $2t + 1$ share distances for users j and k, and obtains the model distance between user j and k, i.e., $\|\mathbf{w}_j - \mathbf{w}_k\|^2$, through an $[n, 2t + 1, n - 2t]_p$ Reed-Solomon decoding algorithm that can correct up to ℓ errors and r erasures. The idea of recovery is that share distances correspond to the evaluation points of a univariate polynomial $h^{j,k} : \mathbb{F}_p \to \mathbb{F}_p$ of degree at most $2t$, defined by $h^{j,k}(\theta) := \|f_j(\theta, \mathbf{w}_j) - f_k(\theta, \mathbf{w}_k)\|^2$. $h^{j,k}$ can be viewed as the encoding polynomial of the $[n, 2t + 1, n - 2t]_p$ Reed-Solomon decoding algorithm. Here, any Byzantine user b's contaminated share distances $\{a_b^{j,k}\}_{j,k\in\mathcal{U}} (\neq \{d_i^{j,k}\}_{j,k\in\mathcal{U}})$ correspond to errors and dropping users' missing computations correspond to erasures. Then, the server obtains the model distance by using the relation $\|f_j(0, \mathbf{w}_j) - f_k(0, \mathbf{w}_k)\|^2 = \|\mathbf{w}_j - \mathbf{w}_k\|^2$.

After obtaining $\{\|\mathbf{w}_j - \mathbf{w}_k\|^2\}_{j,k\in\mathcal{U}}$, the server generates $\mathcal{U}^{sel}$ through Multi-Krum and sends it to all the users.

Secure Aggregation Phase

In this phase, the server and the users securely and jointly compute the aggregation result of the selected local models. Each user i computes the aggregated share $\mathbf{s}_i^{\mathcal{U}^{sel}} = \sum_{j\in\mathcal{U}^{sel}} \mathbf{sh}_{j,i}$ and sends it to the server. The server collects at least $t + 1$ aggregated shares, and recovers an univariate polynomial $h^{\mathcal{U}^{sel}} : \mathbb{F}_p \to \mathbb{F}_p^m$ of degree at most t defined by $h^{\mathcal{U}^{sel}}(\theta) := \sum_{j\in\mathcal{U}} f_j(\theta, \mathbf{w}_j)$ using an $[n, t+1, n-t]_p$ Reed-Solomon code decoding algorithm in a similar manner to that described in the previous section. Here, any Byzantine user b's contaminated aggregated share $\mathbf{a}_b^{\mathcal{U}^{sel}} (\neq \mathbf{s}_b^{\mathcal{U}^{sel}})$ corresponds to an error. Then, the server obtains the aggregation result by using the relation $\sum_{i\in\mathcal{U}^{sel}} f_i(0, \mathbf{w}_i) = \sum_{i\in\mathcal{U}^{sel}} \mathbf{w}_i$.

Summary

BREA satisfies three goals as described in section 3.3 as long as $(n - 1 - \max\{c + 2, r + 2t\})/2 \geq \ell$ because $[n, 2t + 1, n - 2t]_p$ Reed-Solomon code and Multi-Krum succeeds as long as $\lfloor (n - (2t + 1) - r)/2 \rfloor \geq \ell$ and $\lfloor (n - c - 3)/2 \rfloor \geq \ell$, respectively.

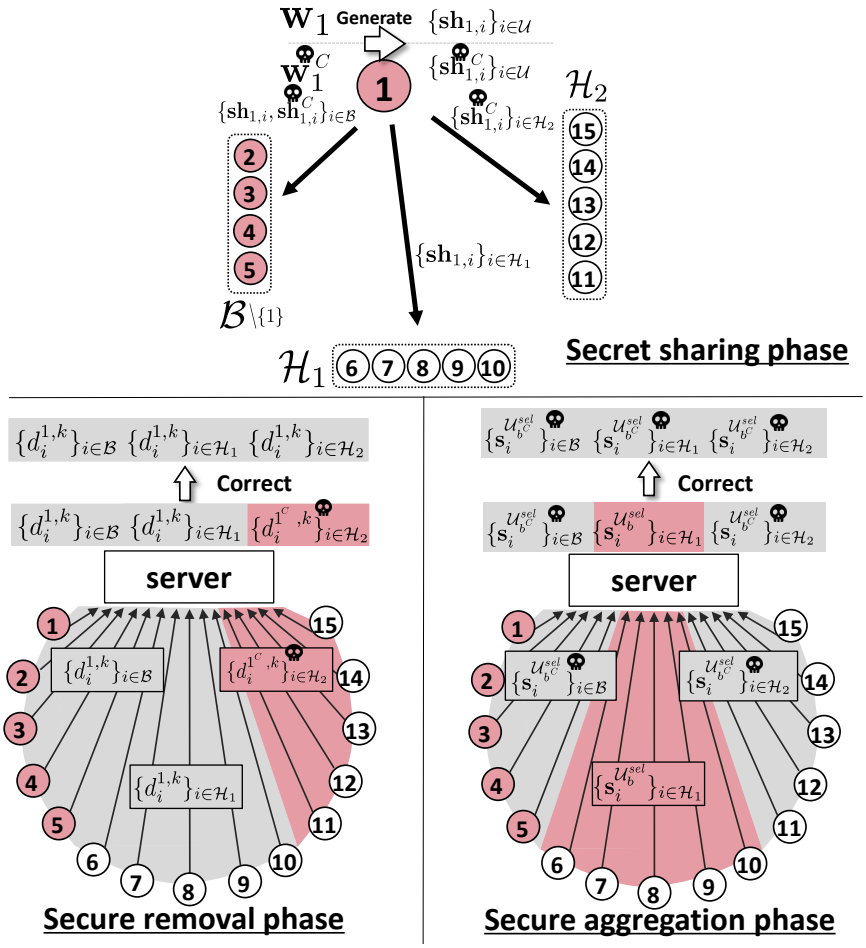

Figure 5.4: Colluding contaminated model injection attack under the condition $n = 15$, $\ell = 5$, $t = 2$, $r = 0$, $c = 2$

5.2 Communication Complexity Reduction

The goal of this section is to extend BREA to reduce communication complexity. Specifically, this section proposes the extension called BREA-SV after answering the question of how resilience against Byzantine attacks has been preserved when the share verification is omitted. The motivation for raising this question is that Reed-Solomon decoding is not enough to be against Byzantine attacks and the share verification is inevitable. According to this observation, the author designs BREA-SV so that some parts of share verification from users to the semi-honest server. This eliminates the need for users' downloading commitments.

5.2.1 BREA without Share Verification

The Reed-Solomon decoding detects contaminated shares as well as contaminated share distances and contaminated aggregated shares. This is because if a Byzantine user injects a contaminated share, share distances and aggregated shares which include this contaminated share is regarded as contaminated. However, BREA without share verification cannot remove the contaminated share distances and contaminated aggregated shares even if there is only one Byzantine user. For example, if a Byzantine user sends its contaminated shares to all the users, all the users send contaminated share distances and contaminated aggregated shares to the server, i.e., the number of errors is n.

The above attack suggests us a vulnerability in that shares of different local models can be sent by a Byzantine user if share verification is omitted. This sub-section designs an attack of colluding Byzantine users, which aims to inject the contaminated models of any Byzantine users, using the vulnerability called *colluding contaminated model injection attack*. The idea behind this attack is to create a situation where any Byzantine user uses shares of a *benign* model and those of a *contaminated* model at the secure removal phase and the secure aggregation phase, respectively. This attack succeeds if the number of Byzantine users satisfies the following condition: $\ell \geq \lceil n/3 \rceil$.

Colluding Contaminated Model Injection Attack

Fig. 5.4 illustrates the attack procedure, where $n = 15$, $\ell = 5$, $t = 2$, $r = 0$ and $m = 2$. Byzantine user b is user 1 and the other colluding Byzantine users are from user 2 to 5. In the secret sharing phase, user 1 prepares the benign model $\mathbf{w}_1$ and contaminated model $\mathbf{w}_1^C$. The Byzantine users carefully interact with honest users so that $\mathbf{w}_1$ is used in the secure removal phase to bypass the Multi-Krum scheme and $\mathbf{w}_1^C$ is used in the secure aggregation phase to inject $\mathbf{w}_1^C$ into aggregated shares of honest users. To this end, user 1 divides all the users in $\mathcal{H}$ into two disjoint groups $\mathcal{H}_1$ and $\mathcal{H}_2$ such that $|\mathcal{H}_1| \leq \ell$ and $|\mathcal{H}_2| \leq \ell$ hold. User 1 then generates shares $\mathbf{sh}_{1,i}$ and $\mathbf{sh}_{1,i}^C$ from $\mathbf{w}_1$ and $\mathbf{w}_1^C$, respectively. Here, $\mathbf{sh}_{1,i}$ and $\mathbf{sh}_{1,i}^C$ are sent to users in $i \in \mathcal{H}_1$ and $i \in \mathcal{H}_2$, respectively. For each user in $i \in \mathcal{B} \setminus \{1\}$ (i.e., the other Byzantine users), user 1 sends both $\mathbf{sh}_{1,i}$ and $\mathbf{sh}_{1,i}^C$.

In the secure removal phase, for the shares of user 1, honest users in $\mathcal{H}_1$ and Byzantine users compute and send pairwise share distances $\{d_i^{1,k}\}_{k\in\mathcal{U}} = \{\|\mathbf{sh}_{1,i} - \mathbf{sh}_{k,i}\|^2\}_{k\in\mathcal{U}}$ $(i \in \mathcal{H}_1 \cup \mathcal{B})$ and honest users in $\mathcal{H}_2$ compute and send $\{d_i^{1^C,k}\}_{k\in\mathcal{U}} = \|\mathbf{sh}_{1,i}^C - \mathbf{sh}_{k,i}\|^2\}_{k\in\mathcal{U}}$ $(i \in \mathcal{H}_2)$. As a result, the server obtains $\{\|\mathbf{w}_1 - \mathbf{w}_k\|^2\}_{k\in\mathcal{U}}$ since the Reed-Solomon code decoding algorithm removes $\{d_i^{1^C,k}\}_{k\in\mathcal{U},i\in\mathcal{H}_2}$.

In the secure aggregation phase, honest users in $\mathcal{H}_2$ and Byzantine users compute and send

aggregated shares $\mathbf{s}_i^{\mathcal{U}_{1^C}^{sel}} = \sum_{j\in\mathcal{U}^{sel}\setminus\{1\}} \mathbf{sh}_{j,i} + \mathbf{sh}_{1,j}^C$ ($i \in \mathcal{H}_2 \cup \mathcal{B}$). The Reed-Solomon code decoding algorithm removes aggregated shares $\{\mathbf{s}_i^{\mathcal{U}_1^{sel}} = \sum_{j\in\mathcal{U}^{sel}\setminus\{1\}} \mathbf{sh}_{j,i} + \mathbf{sh}_{1,j}\}_{i\in\mathcal{H}_1}$, and as a result, the server obtains the aggregation result $\sum_{j\in\mathcal{U}^{sel}\setminus\{1\}} \mathbf{w}_j + \mathbf{w}_1^C$. Consequently, the contaminated model $\mathbf{w}_1^C$ is injected into the aggregation result.

Byzantine Resilience

The colluding contaminated model injection attack succeeds when $\ell \geq \lceil n/3 \rceil$ because Byzantine users can generate $\mathcal{H}_1$ and $\mathcal{H}_2$ such that $|\mathcal{H}_1| \leq \ell$ and $|\mathcal{H}_2| \leq \ell$ hold as described in the previous section. The maximum number of Byzantine users is reduced from $\ell \leq (n-1-\max\{c+2, r+2t\})/2$ to $\ell < \lceil n/3 \rceil$. Moreover, if $\ell < \lceil n/3 \rceil$, this attack can be detected, but not be corrected by the server, because the number of contaminated share distances and aggregated shares exceeds ℓ. Therefore, BREA without share verification provides a weaker level of resilience than BREA.

5.2.2 Communication Complexity Reduction of Share Verification: BREA-SV

To achieve the same Byzantine resilience as BREA, the author needs share verification. This section designs a share verification method that eliminates users' need for downloading commitments to achieve communication efficiency for fast learning. Section 5.2.2 describes a naive method for share verification in BREA with a large complexity in communication. Here, the author calls the naive method *naive BREA*, which is used as the reference for comparison. Section 5.2.2 designs the proposed share verification method which is used by BREA-SV.

Naive BREA

In naive BREA, a *semi-honest* server is responsible for broadcasting all users' commitments. Each user i sends its commitment to the server. The server sends all the other users' commitments to each user i using unicast channels. Since the server is semi-honest, the commitments are correctly delivered.

In naive BREA, each user i uploads $t|\mathbf{w}|$ bytes, and downloads $(n-1)t|\mathbf{w}|$ bytes because each commitment is $t|\mathbf{w}|$ bytes. Since model/share size $|\mathbf{w}|$ is from hundreds of K bytes to hundreds of M bytes, such download traffic becomes a bottleneck for fast learning if users' devices are accommodated by low-bandwidth networks like mobile networks.

BREA-SV

Overview Naive BREA has a high communication cost for each user to verify shares. Since the verification of each share is performed by the user who received the share, BREA requires each user to send a commitment to $n-1$ users who receive that user's shares.

In contrast, BREA-SV, the proposed method, splits the verification process between the server and the users, rather than having the user who receives the share perform all the verification processes. Since each share cannot be reconstructed from the corresponding commitment, users can delegate the verification process using the commitment to the server.

With this approach, each user only needs to send the commitment to the server once, instead of sending it to all other users. Consequently, the number of times each user's commitment needs to be transmitted is reduced to just once. This significantly lowers the communication cost for each user and achieves efficient share verification.

Rationale A candidate solution is that the share verification with users' commitments is performed by the server instead of individual users. Precisely, the author carefully considers where three terms are computed by either a user or a server: $g^{\mathbf{sh}_{i,j}}$, $g^{\mathbf{w}_i}$ and $g^{\sum_{k=1}^{t} \mathbf{r}_{ik} {\theta_j}^k}$ of (5.5). The author decides that a user computes the first and second terms because the server should not see $\mathbf{sh}_{i,j}$ and $\mathbf{w}_i$ and that the server computes the third term. This satisfies the privacy requirement that the server should not obtain information about users' local models. The server also multiplies $g^{\mathbf{w}_i}$ and $g^{\sum_{k=1}^{t} \mathbf{r}_{ik} {\theta_j}^k}$, and checks if (5.5) holds. This enables each user i to send its commitment of random vectors $(g^{\mathbf{r}_{i1}}, g^{\mathbf{r}_{i2}}, \ldots, g^{\mathbf{r}_{it}})$ to only the server, which avoids each user i from downloading the commitments from all the other users. In addition, the verification of (5.5) is performed using hash values of the computation results to reduce communication costs. Specifically, the user sends the hash value of the left-hand side of (5.5) to the server, and the server computes the hash value of the right-hand side of (5.5) and compares it with the hash value received from the user.

This solution, however, has a disadvantage that the server should compute $g^{\sum_{k=1}^{t} \mathbf{r}_{ik} {\theta_j}^k}$, which requires td exponentiations for verifying a single share. The author addresses this issue by computing firstly the exponential part of the right-hand side of (5.5), i.e., $\sum_{k=1}^{t} \mathbf{r}_{ik} {\theta_j}^k$. Our computation method of $g^{\sum_{k=1}^{t} \mathbf{r}_{ik} {\theta_j}^k}$ is as follows: Each user i sends random vectors themselves $\{\mathbf{r}_{ik}\}_{k\in[t]}$ to the server. The server computes $\sum_{k=1}^{t} \mathbf{r}_{ik} {\theta_j}^k$, and then computes $g^{\sum_{k=1}^{t} \mathbf{r}_{ik} {\theta_j}^k}$. As compared to a seminal work [44] that requires td exponentiations, BREA-SV requires d exponentiations for verifying a single share.

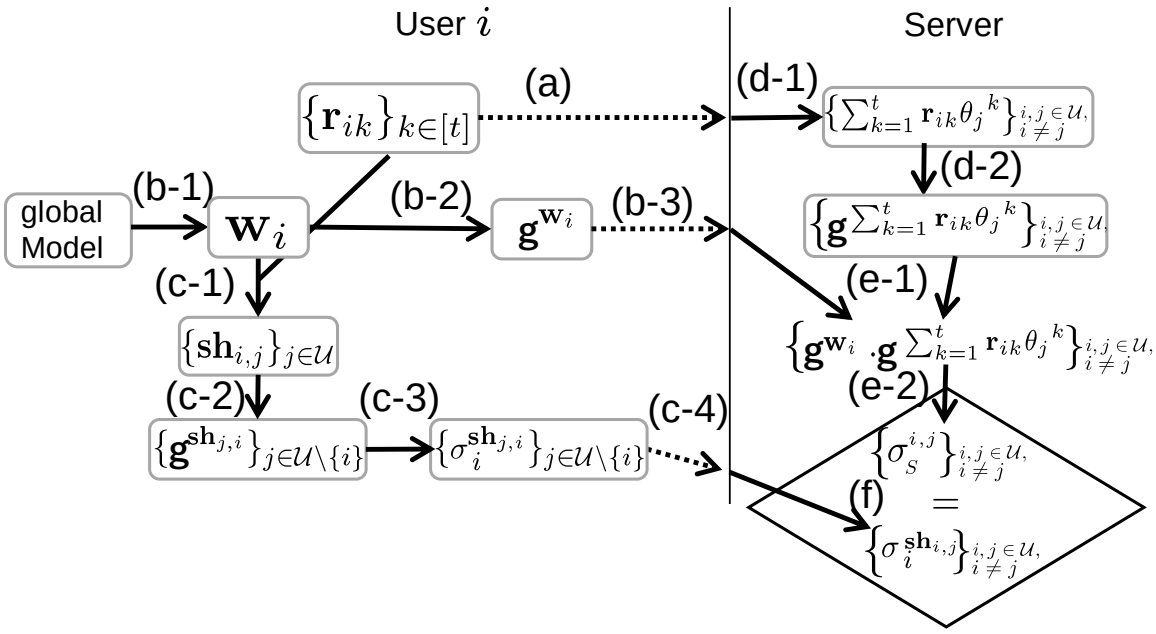

Figure 5.5: Share verification of BREA-SV

Protocol The rest of the subsection describes the share verification procedure of BREA-SV according to Fig. 5.5. Each user i and the server execute the share verification in the following way:

User i: Computing $\{\mathbf{r}_{ik}\}_{k\in[t]}$, $g^{\mathbf{w}_i}$, and $\{g^{\mathbf{sh}_{j,i}}\}_{j\in\mathcal{U}\setminus\{i\}}$

1. User i computes random vectors $\{\mathbf{r}_{ik}\}_{k\in[t]}$, and sends them to the server. ((a) in Fig. 5.5)
2. User i trains the shared model and generates its local model $\mathbf{w}_i$. Then, user i computes $g^{\mathbf{w}_i}$ and sends it to the server. ((b-1), (b-2) and (b-3) in Fig. 5.5)
3. User i generates shares $\{\mathbf{sh}_{j,i}\}_{j\in\mathcal{U}}$ and sends each share to user $j \in \mathcal{U}\setminus\{i\}$. After receiving the shares $\{\mathbf{sh}_{j,i}\}_{j\in\mathcal{U}\setminus\{i\}}$ of all the other users ((c-1) in Fig. 5.5), user i computes $\{g^{\mathbf{sh}_{j,i}}\}_{j\in\mathcal{U}\setminus\{i\}}$. To reduce communication costs by sending each masked share $g^{\mathbf{sh}_{j,i}}$, user i generates a hash value $\sigma_i^{\mathbf{sh}_{j,i}}$ for each $g^{\mathbf{sh}_{j,i}}$ (c-3) in Fig. 5.5), and sends it to the server((c-4) in Fig. 5.5).

Server: Computing right-hand side of (5.5) and checking (5.5)

1. The server recieves $\{\mathbf{r}_{ik}\}_{i\in\mathcal{U},k\in[t]}$ and computes $\{\sum_{k=1}^{t}\mathbf{r}_{ik}{\theta_j}^k\}_{i,j\in\mathcal{U},i\neq j}$. Then, the server computes $\{g^{\sum_{k=1}^{t}\mathbf{r}_{ik}{\theta_j}^k}\}_{i,j\in\mathcal{U},i\neq j}$. ((d-1) and (d-2) in Fig. 5.5.)
2. The server receives $\{g^{\mathbf{w}_i}\}_{i\in\mathcal{U}}$ and computes $\{g^{\mathbf{w}_i+\sum_{k=1}^{t}\mathbf{r}_{ik}{\theta_j}^k}\}_{i,j\in\mathcal{U},i\neq j}$, and generates hash values $\{\sigma_S^{i,j}\}_{i,j\in\mathcal{U},i\neq j}$ from the computed values. ((e-1) and (e-2) in Fig. 5.5)
3. The server receives $\{\sigma_i^{\mathbf{sh}_{j,i}}\}_{i,j\in\mathcal{U},i\neq j}$ and the server checks (5.5) for all the shares exchanged among the users. ((f) in Fig. 5.5)

BREA-SV achieves the privacy preservation described in section 3.3. This is because the server can know all the coefficients of (5.3) other than the local model of any user, but cannot know either the local model itself or any share of the local model. As a result, it cannot recover the local model of any user. Section 5.3 proves the Byzantine-resilience described in section 3.3.

Naive BREA VS BREA-SV

The author analyzes the computational and communication costs of the share verification, as summarized in Table 1.2. For all the other analyses of computation and communication, please refer to [25].

Computational Cost A computational cost is represented by the number of multiplications. To reduce the number of multiplications, an exponentiation operation is computed by *exponentiation of by squaring*. Exponentiation by squaring is a calculation method that reduces the number of multiplications by splitting an exponentiation calculation into smaller exponentiation calculations. This method is based on the fact that for a calculation of x^y, $x^y = (x^2)^{y/2}$ holds if y is even, and $x^y = x(x^2)^{(y-1)/2}$ holds if y is odd. This calculation is called recursively until y becomes 0 or 1, and it returns 1 when it becomes 0 or x when it becomes 1. As a result, the exponentiation of by squaring requires $O(\log y)$ multiplications for calculating x^y, and the naive exponentiation requires $O(y)$ multiplications.

In naive BREA, user i's computational cost consists of three parts: 1) generating $\mathbf{C}_{f_i(\theta,\mathbf{w}_i)}$, 2) generating $\{g^{\mathbf{sh}_{j,i}}\}_{j\in\mathcal{U}\setminus\{i\}}$, 3) verifying if (5.5) holds for all the received shares. Generating $\mathbf{C}_{f_i(\theta,\mathbf{w}_i)}$ and $\{g^{\mathbf{sh}_{j,i}}\}_{j\in\mathcal{U}\setminus\{i\}}$ have a computational cost of $O(mn\log p)$ where p is a finite field size. Performing checking (5.5) has a computational cost of $O(mn^2\log n)$ [44]. Therefore, the computational cost is $O(mn\log p + mn^2\log n)$. On the other hand, the server performs no computations.

In BREA-SV, user i's computational cost consists of two parts: 1) generating $\{g^{\mathbf{sh}_{j,i}}\}_{j\in\mathcal{U}\setminus\{i\}}$, 2) generating $g^{\mathbf{w}_i}$. The former has a computational cost of $O(mn\log p)$ and the latter has that of $O(m\log p)$, therefore, user i's computational cost is $O(mn\log p)$. On the other hand, the server's computational cost consists of three parts: 1) computing (d-1), 2) computing (d-2), and 3) computing (e-1). Computing (d-1) for all the shares exchanged among users has a computational cost of $O(mn^2\log^2 n)$ [42]. Computing (d-2) for all the shares exchanged among users has a computational cost of $O(mn^2\log p)$. Computing (e-1) for all the shares exchanged among users has a computational cost of $O(mn^2)$. Therefore, the computational cost is $O(mn^2\log^2 n)$.

BREA-SV has a disadvantage that the server should sequentially compute $\{\sum_{k=1}^{t}\mathbf{r}_{ik}{\theta_j}^k\}_{i,j\in\mathcal{U},i\neq j}$

if the server has a single CPU, whereas each user j computes each $\{g^{\sum_{k=1}^{t} \mathbf{r}_{ik} {\theta_j}^k}\}_{i \in \mathcal{U} \setminus \{j\}}$ in parallel in naive BREA.

Communication Cost In naive BREA, each user sends its commitment to the server and receives $n-1$ commitments from the server. Therefore, the communication cost of each user is $O(ntm)$. On the other hand, the server receives n commitments in total from the users and sends $n-1$ commitments to each user. Therefore, the communication cost of the server is $O(n^2tm)$. In BREA-SV, each user sends its commitment to only the server. Therefore, the communication cost of each user is $O(tm)$. On the other hand, the server receives n commitments in total from the users. Therefore, the communication cost of the server is $O(ntm)$.

As the advantage, BREA-SV reduces the total traffic amount of each user and the server to $O(tm)$ and $O(ntm)$, respectively.

5.3 Security Analysis

The section proves that BREA-SV satisfies Byzantine resilience as described in section 3.3. the author introduces some notations to use for the proof. $\mathcal{U}_{\mathrm{SR}}$ and $\mathcal{U}_{\mathrm{SA}}$ denote sets of all the users that send their messages to the server in the secure removal phase and the secure aggregation phase, respectively. Here, let $\mathcal{H}_{\mathrm{SR}}$ and $\mathcal{H}_{\mathrm{SA}}$ be $\mathcal{H}_{\mathrm{SR}} = \mathcal{U}_{\mathrm{SR}} \setminus \mathcal{B}$ and $\mathcal{H}_{\mathrm{SA}} = \mathcal{U}_{\mathrm{SA}} \setminus \mathcal{B}$, respectively.

Lemma 1. *In the secret sharing phase of BREA-SV, any Byzantine user $b \in \mathcal{B}$ is only able to send the shares generated by a single local model, i.e., $\mathbf{w}_b$ or $\mathbf{w}_b^C$, to any honest user $j \in \mathcal{H}$.*

Proof. Suppose that any Byzantine user $b \in \mathcal{B}$ sends its commitment $(g^{\mathbf{w}_b}, \mathbf{r}_{b1}, \mathbf{r}_{b2}, \ldots, \mathbf{r}_{bt})$ to the server. At this time, if b sends a contaminated share $\mathbf{a}_{b,j}$ to any honest user $j \in \mathcal{H}$, the server detects the attack. This is because $g^{\mathbf{a}_{b,j}}$ computed by j is not equal to $g^{\mathbf{w}_b + \sum_{k=1}^{t} \mathbf{r}_{bk} {\theta_j}^k}$ computed by the server, therefore, (5.5) does not hold. Similarly, if sends its commitment $(g^{\mathbf{w}_b^C}, \mathbf{r}_{b1}, \mathbf{r}_{b2}, \ldots, \mathbf{r}_{bt})$ to the server and sends a contaminated share, which is different from $\mathbf{sh}_{b,j}^C$, to any honest user $j \in \mathcal{H}$, the server detects the attack. □

Theorem 2. *BREA-SV satisfies Byzantine resilience as described in section 3.3 as long as $\lfloor (n - (2t+1) - r)/2 \rfloor \geq \ell$.*

Proof. From Lemma 1, the author supposes that any user $i \in \mathcal{U}$ sends the shares $\mathbf{sh}_{i,j}$ generated by single local model $\mathbf{w}_i$ to each user $j \in \mathcal{H}$.

First, the author shows that BREA-SV takes $\{\|\mathbf{w}_j - \mathbf{w}_k\|^2\}_{j,k\in\mathcal{U}}$ as the inputs of Multi-Krum in the secure removal phase. In this phase, each $j \in \mathcal{H}_{\mathrm{SR}}$ computes and sends $d_j^{i,k}$ to the server, and the server uses a $[n, 2t+1, n-2t]_p$ Reed-Solomon code decoding with r erasures that can tolerate a maximum number of $\lfloor (n-(2t+1)-r)/2 \rfloor$ errors [45]. Therefore, the server certainly can decode $\{\|f_i(\theta, \mathbf{w}_i) - f_k(\theta, \mathbf{w}_k)\|^2\}_{i,k\in\mathcal{U}}$ regardless of the contents of the messages sent by the Byzantine users as long as $\lfloor (n-(2t+1)-r)/2 \rfloor \geq \ell$. Finally, the server obtains the pairwise model distances $\{\|\mathbf{w}_i - \mathbf{w}_k\|^2\}_{i,k\in\mathcal{U}}$.

Next, the author shows that BREA-SV outputs an aggregation result $\sum_{i\in\mathcal{U}^{sel}} \mathbf{w}_i$ in the secure aggregation phase. In this phase, each $j \in \mathcal{H}_{\mathrm{SA}}$ computes and sends $\mathbf{s}_j^{\mathcal{U}^{sel}}$ to the server, and the server uses a $[n, t+1, n-t]_p$ Reed-Solomon code decoding with r erasures that can tolerate a maximum number of $\lfloor (n-(t+1)-r)/2 \rfloor$ errors [45]. Therefore, the server certainly can decode $\sum_{j\in\mathcal{U}^{sel}} f_j(\theta, \mathbf{w}_j)$ regardless of the contents of the messages sent by the Byzantine users as long as $\lfloor (n-(t+1)-r)/2 \rfloor \geq \ell$. Finally, the server obtains the aggregation result $\sum_{i\in\mathcal{U}^{sel}} \mathbf{w}_i$. □

corollary 1. *BREA-SV is said to be (α, ℓ)-Byzantine-resilient [24, 65] as long as $(n - 1 - \max\{c+2, r+2t\})/2 \geq \ell$.*

Phase	Computation	$m = 100K$			$m = 300K$			$m = 500K$		
		$n = 10$	$n = 30$	$n = 50$	$n = 10$	$n = 30$	$n = 50$	$n = 10$	$n = 30$	$n = 50$
Training	(b-1)	16.08	16.08	16.08	48.20	48.20	48.20	80.35	80.35	80.35
Secret sharing	(c)	0.402	1.444	2.846	1.219	4.392	8.886	1.949	7.720	14.51
	(d)	4.124	43.32	142.3	12.19	131.8	456.8	19.99	216.6	725.6
	(e)	0.363	2.805	7.350	0.872	7.509	21.20	1.401	12.45	39.30
	(f)	0.318	3.248	10.79	0.963	9.863	32.70	1.604	16.17	53.62
Secure removal and Secure aggregation	Total	0.316	2.968	9.062	0.984	9.442	27.38	1.722	15.59	46.31

Table 5.2: Summary of major computation time of share verification in BREA-SV (sec). The symbols in the second column correspond to the symbols in Fig. 5.5 and a formula number: (b-1) is training. (c) is computations of (c-1), (c-2) and (c-3). (d) is computations of (d-1) and (d-2). (e) is computation of (e-1) and (e-2). (f) is checking if (5.5) holds for all received shares by each user in naive BREA.

5.4 Performance Analysis

The author measures the execution time of one round of federated learning which we call the protocol running time. The protocol running time is defined as the time from when each user starts learning with its local data to the time when the server finishes updating a global model. The protocol running time is the sum of the computation time and the communication time. Since local models of users

are computed in parallel, the computation time is the sum of each user's computation time and the server's computation time. In contrast, the communication time is the time for each user's sending and receiving shares, commitments, share distances, and aggregated shares to the server.

The protocol running time is computed by simulating each protocol behavior using the measured values of computation time and communication time.

5.4.1 Measurement Method

Overview

The author measures federated learning time of naive BREA and BREA-SV in environments where modern CPUs and those with a GPU device are used for users' devices and the server, respectively, and where broadband and mobile networks are used to accommodate the devices, in the two steps: In the first step, the author measures computation time of time-consuming processes such as training, share verification on a modern computer. In the second step, the author measures learning time using simulation with changing values of the following parameters:

- n: the number of users. The number of users is a variable for evaluating the scalability of a federated learning system.
- m: the size of vectors. For mobile applications, the author assumes using models such as SqueezeNet [61] and MobileNetV3 [62], which range from a few hundred KB to a few MB in size.
- e: the number of a server's CPU cores. the author assumes that each user has one CPU core.
- a: the end-to-end throughput (bps) between each user and a server. The author assumes the throughput is determined by the bandwidth of access networks: broadband and mobile networks.

Computation Time Measurement

The author implements the code in C/C++ and runs the code on a Linux workstation with Intel Core i9 19-10900X CPU (3.70GHz), NVIDIA GeForce RTX 3090 GPU, and 64GB of RAM. The author measures the computation time of processes of BREA-SV listed in Table 5.2 for the pairs of n and m. Here, the author uses a simple neural network model consisting of an input layer, a hidden layer, and an output layer, and the numbers of neurons in the three layers are 90, $m/100$, and 10, respectively. In addition, each parameter in the neural network model is a 32-bit entry. The field size, p, is set as a prime with 32 bits.

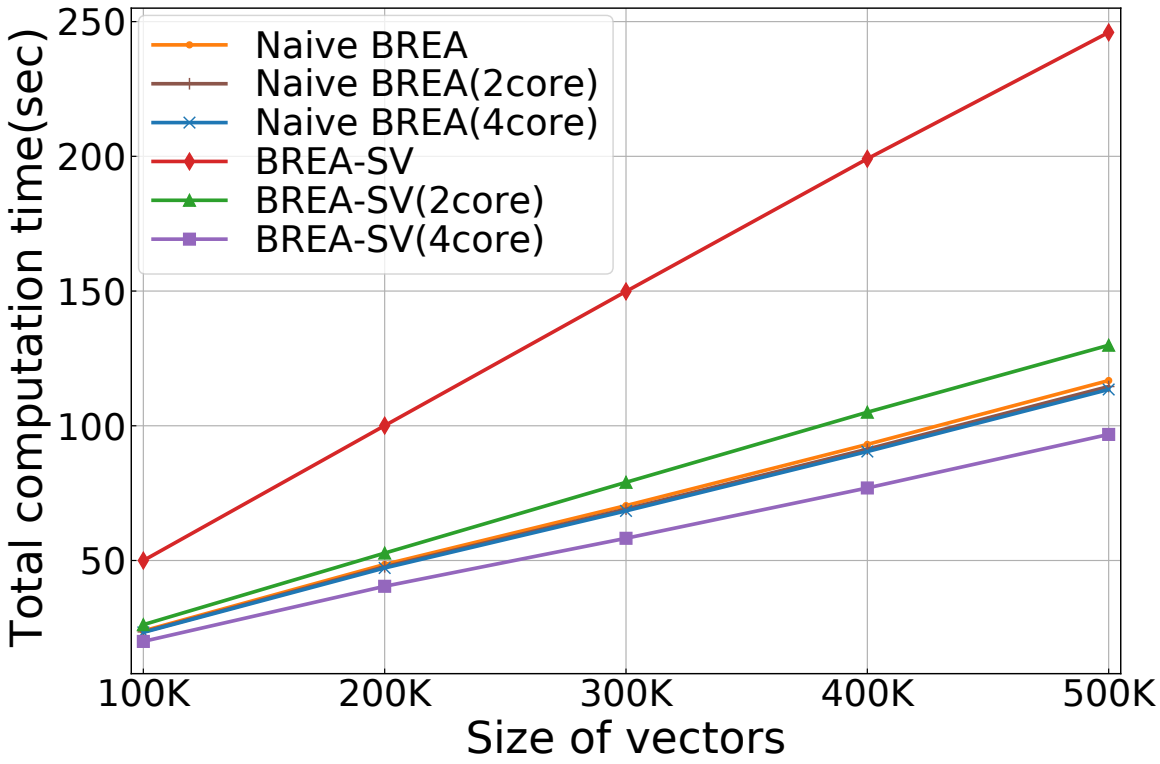

Figure 5.6: Total computation time of naive BREA and BREA-SV with the increase in m.

Simulation Conditions

The author simulates the behaviors of n users and the server, using the measurement values of processes in both cases that the GPU device is used, and that this device is not used. Unless otherwise specified, the default setting is that $n = 30$, $t = 0.1n$, $m = 100K$, $e = 1$, and the server does not use the GPU device. The author assumes that all the users have the same computing and communication capabilities. The author considers two access networks such that the end-to-end throughput between each user and the server is 1Gbps and 10Mbps, which represent the bandwidths for broadband and mobile networks, respectively.

5.4.2 Learning Time

Federated learning time is a sum of the computation time and communication time of one iteration if it is assumed that computation and communication do not overlap. Here, one iteration means the time from when all the users start training the shared model to when the server obtains the aggregation result. This subsection evaluates these three types of time in a stepwise manner.

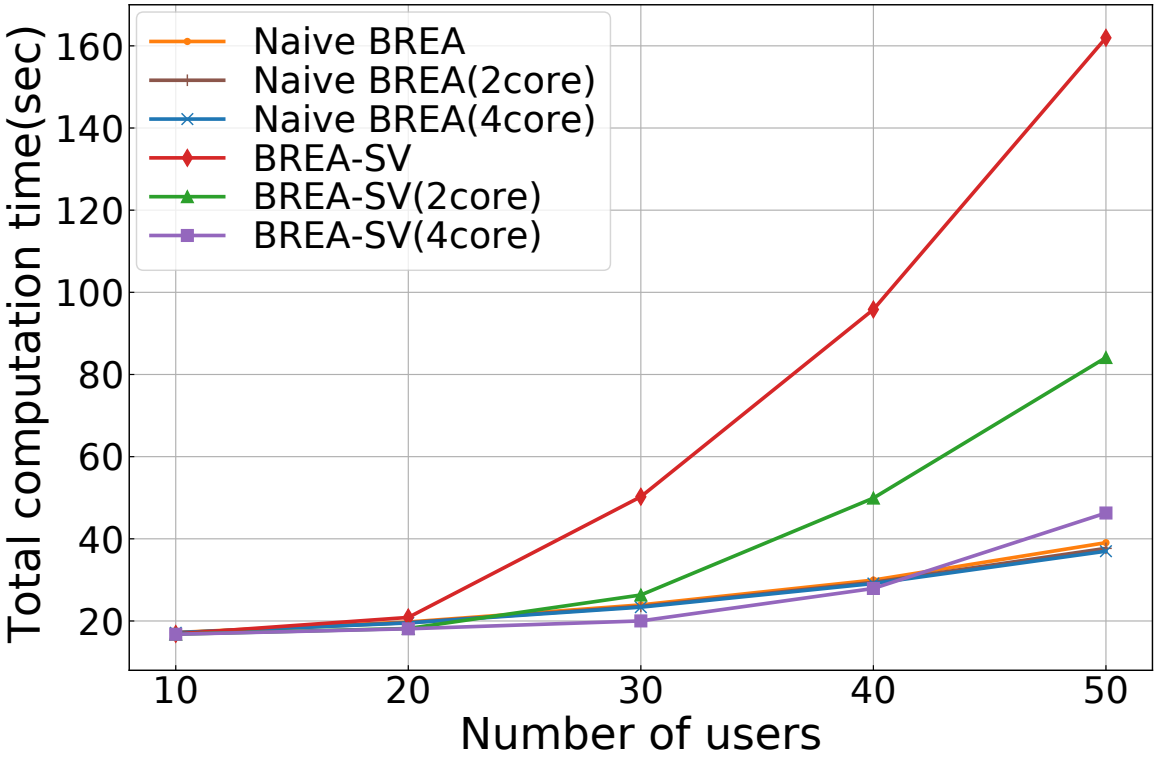

Figure 5.7: Total computation time of naive BREA and BREA-SV with the increase in n. In this figure, the computation time is plotted for cases where the server has 1, 2, and 4 cores.

Computation Time

The author first evaluates computation time of naive BREA, T_N^{comp}, and BREA-SV, T_{SV}^{comp}. As shown in Fig. 5.6 and Fig. 5.7, the computation time increases lineally and quadratically in m and n, respectively. This is due to the computational costs as described in section 5.2.2. Fig. 5.6 and Fig.5.7 also show that the computation time of BREA-SV is larger than that of naive BREA. This is because the computation of (5.5) is performed in parallel by users in naive BREA whereas all the computation is performed by the server in BREA-SV. However, BREA-SV has an advantage in terms of the efficiency of the computation of share verification. Specifically, the computations of (d-1) and (d-2) are performed in parallel with model training, and the computations of (e-1) and (e-2) are performed in parallel with the computations of (c-1) and (c-2) by each user. Based on these observations, the author roughly models the computation time difference of naive BREA from BREA-SV as follows:

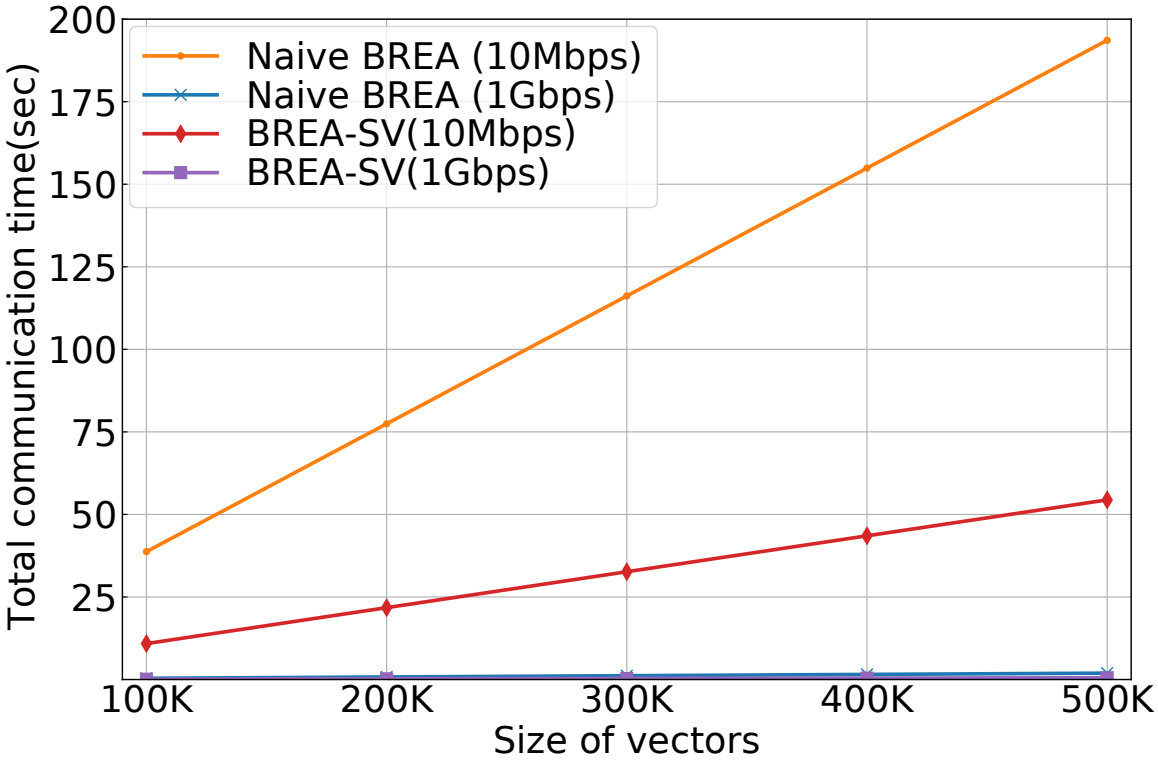

Figure 5.8: Total communication time of naive BREA and BREA-SV with the increase in m.

$$T_N^{comp} - T_{SV}^{comp} := (T^{tr} + T_N^{(5.5)}) - (\max\{T^{tr}, \frac{T_{SV}^{(\mathrm{d})}}{e}\} + \max\{T^{(c)}, \frac{T_{SV}^{(\mathrm{e})}}{e}\}), \tag{5.6}$$

where T^{tr}, $T_N^{(5.5)}$, $(T_{SV}^{(\mathrm{d})})/e$, $T^{(c)}$, $(T_{SV}^{(\mathrm{e})})/e$ and are training time, the share verification time by each user in naive BREA, the total computation time of (d-1) and (d-2) by the server in BREA-SV, the total computation time of (c-1) and (c-2) by each user, and total computation time of (e-1) and (e-2) by the server in BREA-SV, respectively.

Fig. 5.6 and Fig. 5.7 show that the computation time of BREA-SV decreases as the number of CPU cores. Specifically, when the number of cores is e, the computation time decreases to $\frac{1}{e}$.

Communication Time

The author then evaluates communication time of naive BREA, T_N^{comm}, and BREA-SV, T_{SV}^{comm}. The total communication time of naive BREA and BREA-SV increases linearly increase by the increase in m, and increases quadratically by the increase in n, as shown in Fig. 5.8 and Fig 5.9, respectively.

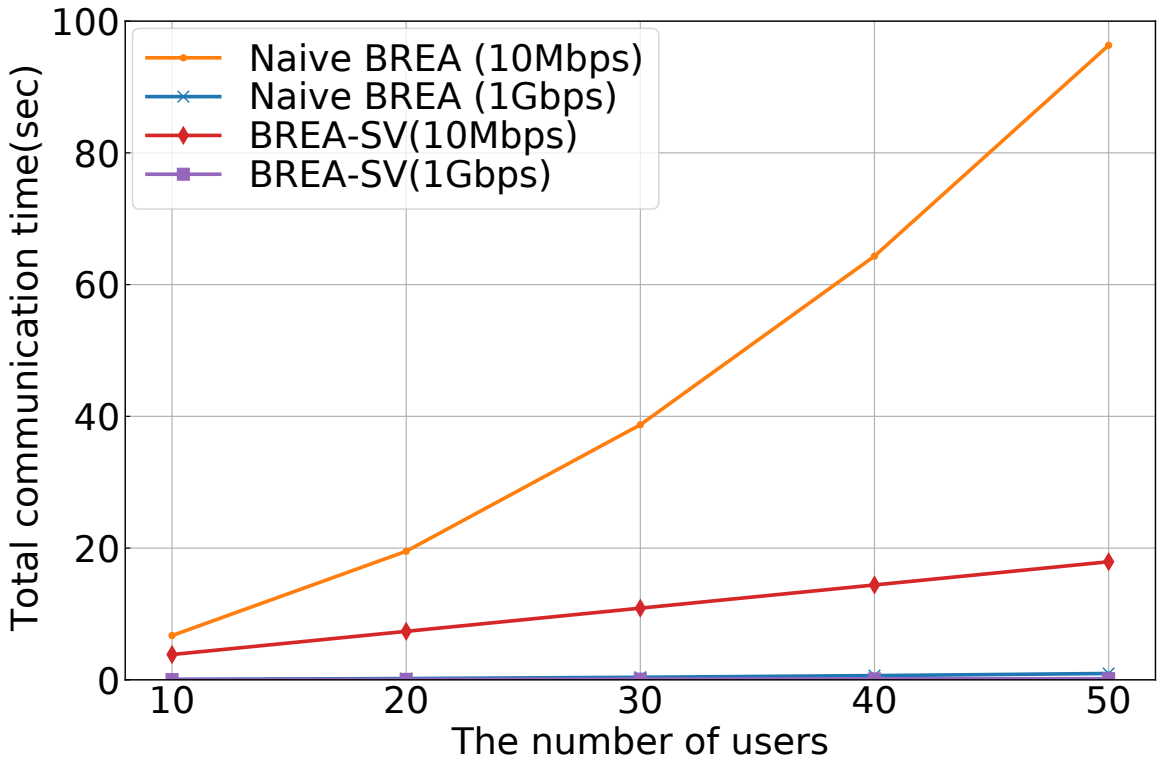

Figure 5.9: Total communication time of naive BREA and BREA-SV with the increase in n.

This is due to the communication costs as described in section 5.2.2. Fig. 5.8 and Fig. 5.9 also show that T_N^{comm} is always longer than T_{SV}^{comm}. This is because naive BREA always has more total traffic volume than BREA-SV. The traffic volume difference of naive BREA from BREA-SV is $(n-1)t|\mathbf{w}|$, and therefore, the communication time difference is defined as follows:

$$T_N^{comm} - T_{SV}^{comm} := \frac{(n-1)t|\mathbf{w}|}{\frac{a}{8}}. \tag{5.7}$$

The higher the throughput, the smaller the communication time difference.

Learning Time

1) Learning Time without a GPU device

First, the author evaluates learning times for BREA-SV and naive BREA in the case that the server does not use a GPU device. Since learning time in naive BREA and BREA-SV is the sum of computation time and communication time, if $(T_N^{comp} - T_{SV}^{comp}) + (T_N^{comm} - T_{SV}^{comm}) < 0$, naive BREA is faster. Also, if the inequality is reversed, BREA-SV is faster.

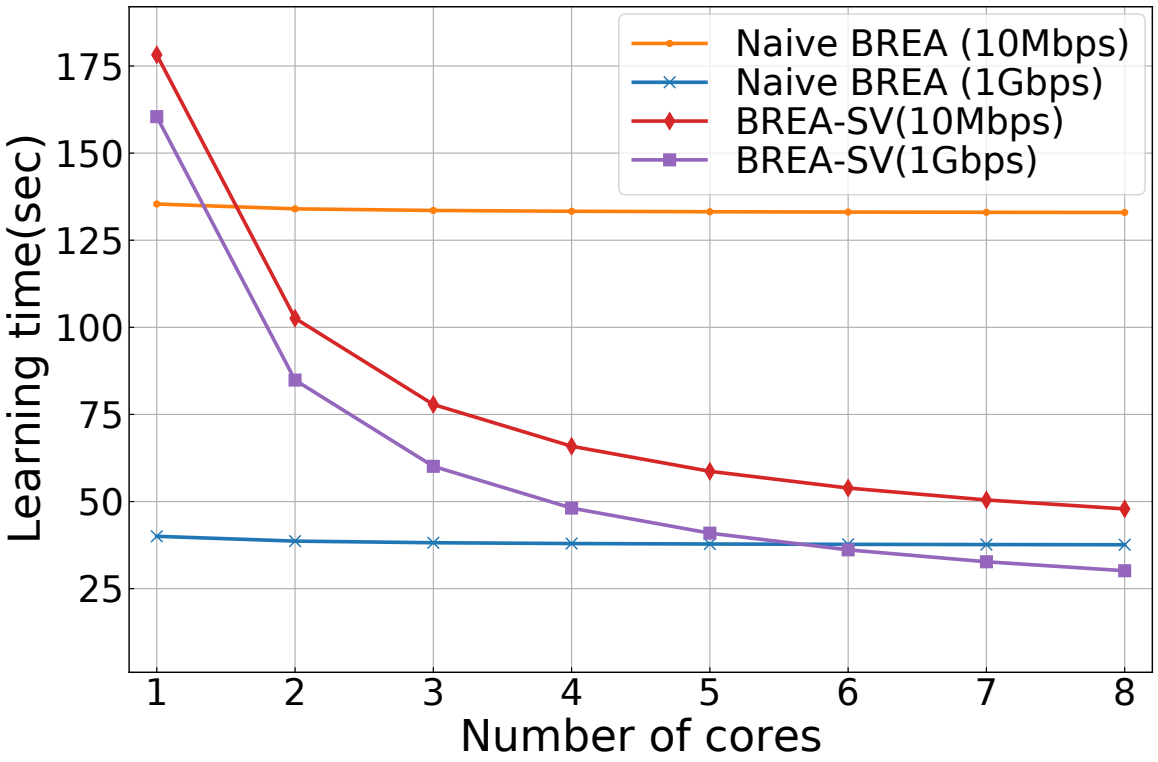

Figure 5.10: Learning time of naive BREA and BREA-SV with the increase in e.

Once n and m are fixed, which is faster, naive BREA or BREA-SV, is determined by the end-to-end throughout and the number of cores of the server. If the bandwidth is abundant and the server has only one CPU core ($e = 1$), the learning time of naive BREA is faster than that of BREA-SV. This is because $T_N^{comp} - T_{SV}^{comp}$ is a big negative value and $T_N^{comm} - T_{SV}^{comm}$ is a small positive value. However, the bandwidth is not abundant, learning time difference of naive BREA from BREA-SV dramatically decreases, even if $e = 1$. This is because $T_N^{comm} - T_{SV}^{comm}$ is a bigger negative value, the narrower the bandwidth. In addition, as the number of CPU cores of the server increases, $T_N^{comp} - T_{SV}^{comp}$ dramatically decreases.

Fig. 5.10 shows the learning time of naive BREA and BREA-SV with the increase in e in a setting where $m = 100K$ and $n = 50$. If $e = 1$, the learning time of naive BREA is faster than that of BREA-SV. On the other hand, if $e \geq 2$ and $a = 10$M or $e \geq 5, a = 1$G, the learning time of BREA-SV is faster than that of naive BREA.

2) Leaning Time with multiple cores and a GPU device

Although the previous sub-section shows the BREA-SV shortens the learning time, the learning

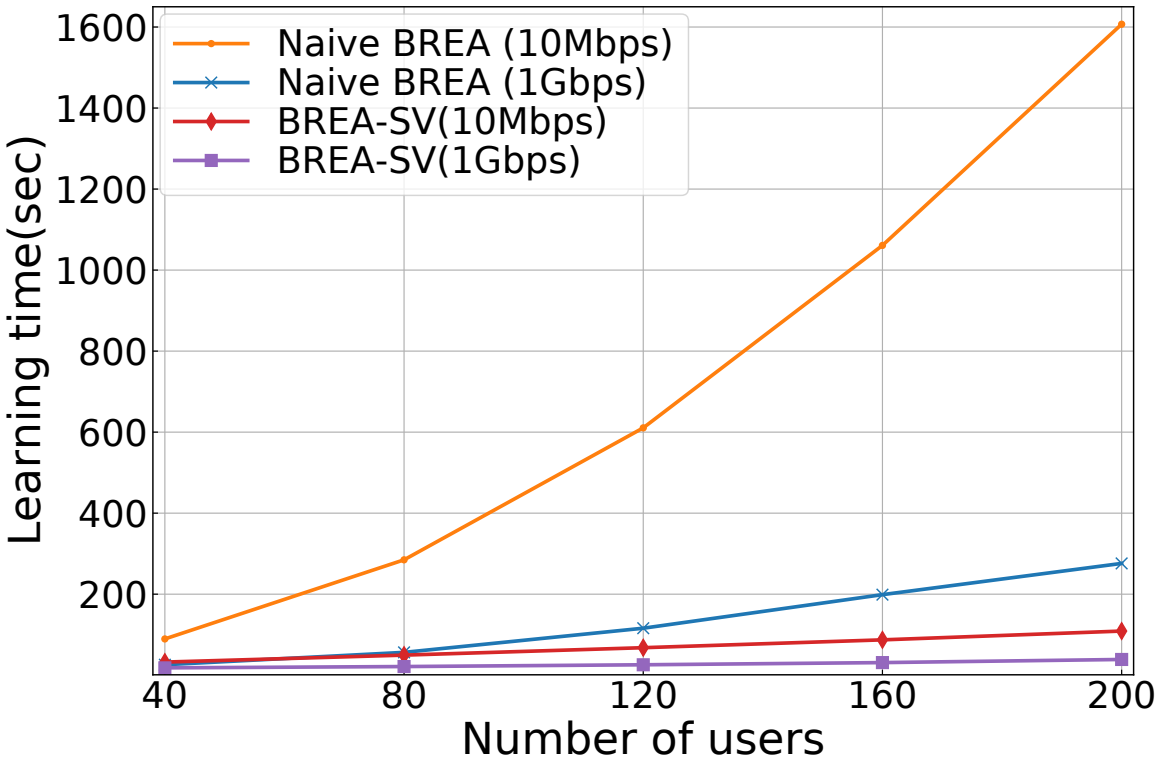

Figure 5.11: Learning time of naive BREA and BREA-SV with the increase in n. The server is equipped with GPU.

time is still long for many users when the computational power of the server is not rich. For example, it takes about 175 seconds for $n = 40$, $e = 1$, and $a = 10$M. If the server can offload the computations of the share verification to a GPU device, BREA-SV can significantly reduce the computation time. This is because the GPU device can compute each user's share in parallel, and each element of each share (vector) in parallel.

Fig. 5.11 shows the learning time of naive BREA and BREA-SV with the increase in n when the server is equipped with GPU. Although the learning time for larger users, i.e., larger n values, the learning time of BREA-SV almost linearly increases. In the case that $n = 200$ and $a = 10$M, the learning time of BREA-SV is about 100 seconds and BREA-SV is 15× faster than naive BREA. This validates that BREA-SV has the scalability to a large number of users in mobile networks.

5.5 Conclusion

This chapter addressed the large communication complexity BREA, caused by broadcasting of users' commitments in the two steps. In the first step, the author clarified that the weaker resilience is preserved due to the error-correcting capability of the Reed-Solomon decoding, when the share verification is omitted. Thus, in the second step, the author designed BREA-SV to reduce communication cost while preserving the same level of resilience against Byzantine attacks. BREA-SV offloads the computation of non-private information pieces to a semi-honest server so that the server does not obtain information about users' training data. In addition, BREA-SV reduces the total learning time by carefully computing multiplications of the share verification. These efforts are successful at achieving faster learning time than BREA.

The future directions of BREA-SV are summarized below: One promising direction is reducing the sizes of commitments by preparing commitments for aggregation of individual local models' vectors similar to the study of Tayyebeh et al [51]. A research issue is how to prevent adversaries from generating contaminated aggregated vectors of which commitment does not pass the share verification. Another promising direction is adding resilience against Byzantine servers whereas most studies assume semi-honest servers. Candidate solutions include leveraging TEE and adding Byzantine fault tolerance to multiple servers.

Chapter 6

Conclusion

In this thesis, the author designed fast model aggregation protocols for Byzantine-resilient and secure aggregation in federated learning. Although Byzantine-resilient and secure aggregation for federated learning enhances the security of federated learning, seminal works have heavy computation and communication costs. The author designed a secure aggregation protocol and a Byzantine-resilient secure aggregation protocol that improve the bottlenecks of computation and communication without compromising privacy strength and Byzantine-resilient strength. The author believes that the author's theoretical reduction in computation and communication costs contributes to the acceleration of federated learning, making federated learning more feasible.

In Chapter 4, the author improves the performance of existing secure aggregation protocols. The existing secure aggregation protocols sacrifice either computation cost or communication cost. A naïve secure aggregation protocol called SecAgg achieves low communication costs by having users send random seeds instead of random masks. However, it requires that a server incurs a substantial computation cost to reconstruct the random masks from the dropout users' random seeds. The state-of-the-art secure aggregation protocol called LightSecAgg reduces computation costs by having users send the random masks themselves, thus avoiding the need for the server's mask reconstruction. However, the exchange of random masks incurs high communication costs. To address this issue, the author designed a fast secure aggregation protocol called BalancedSecAgg, which balances computation and communication costs by complementing SecAgg and LightSecAgg. In BalancedSecAgg, some random masks are sent as seeds instead of the masks themselves to reduce communication costs. Additionally, users perform some of the random mask reconstructions instead of the server, thereby reducing the server's computation cost.

In Chapter 5, the author improves the performance of an existing Byzantine-resilient secure

aggregation protocol. The existing secure aggregation protocol called BREA requires a high communication cost duto to the exchange of commitments for share verification. In BREA, users broadcast commitments to each other to verify shares mutually. To address this issue, the author offloaded share verification to the server, reducing the number of commitment transfers and, consequently, the communication costs. Additionally, to handle the increased computation costs resulting from the server performing centralized share verification, they designed a share verification algorithm that reduces the number of multiplications.

Future directions for improving performance are as follows. The first is to design a method to reduce the communication cost of share exchange while maintaining security for Byzantine-resistant secure aggregation. Many studies adopt the model-split approach explained in the Related Work section, which reduces the size of each share at the cost of limiting the number of allowable semi-honest users, dropout users, and Byzantine users. To address this issue, the author may modify Shamir's secret sharing algorithm to generate some shares as random masks, similar to BalancedSecAgg. Specifically, for t semi-honest users, each user 1) generates t random masks, 2) generates a polynomial of degree $t + 1$ through Lagrange polynomial using these random masks and its local model, and 3) generates the remaining $n - (t + 1)$ shares by evaluating the polynomial, where n is the number of all users. The shares generated by this share generation maintain the same security as those generated by Shamir's secret sharing. Additionally, each user can send random masks as random seeds, reducing the communication cost of share exchange. The second is to focus on optimizations in implementation. Processes that compute independently for each element of a vector like a model or share, and processes that compute independently for each user's vector, might be optimized using parallel computing technology. Examples of the former include the generation of shares from local models by each user in BREA-SV, and the latter includes the server's share verification. Parallelizing these cryptographic processes using SIMD instructions may reduce computation time significantly. The author believes that federated learning can be further applied to real-world scenarios.

Bibliography

[1] T. Wuest, D. Weimer, C. Irgens, and K.-D. Thoben, "Machine learning in manufacturing: advantages, challenges, and applications," *Production & Manufacturing Research*, vol. 4, no. 1, pp. 23–45, 2016.

[2] B. Li, P. Qi, B. Liu, S. Di, J. Liu, J. Pei, J. Yi, and B. Zhou, "Trustworthy ai: From principles to practices," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–46, 2023.

[3] L. Caruccio, D. Desiato, G. Polese, G. Tortora, and N. Zannone, "A decision-support framework for data anonymization with application to machine learning processes," *Information Sciences*, vol. 613, pp. 1–32, 2022.

[4] N. Ponomareva, H. Hazimeh, A. Kurakin, Z. Xu, C. Denison, H. B. McMahan, S. Vassilvitskii, S. Chien, and A. G. Thakurta, "How to dp-fy ml: A practical guide to machine learning with differential privacy," *Journal of Artificial Intelligence Research*, vol. 77, pp. 1113–1201, 2023.

[5] P. Kairouz, S. Oh, and P. Viswanath, "Extremal mechanisms for local differential privacy," *Advances in neural information processing systems*, vol. 27, 2014.

[6] P. Mohassel and Y. Zhang, "Secureml: A system for scalable privacy-preserving machine learning," in *2017 IEEE symposium on security and privacy (SP)*, pp. 19–38, IEEE, 2017.

[7] J. So, B. Guler, and S. Avestimehr, "A scalable approach for privacy-preserving collaborative machine learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 8054–8066, 2020.

[8] Z. Gu, H. Jamjoom, D. Su, H. Huang, J. Zhang, T. Ma, D. Pendarakis, and I. Molloy, "Reaching data confidentiality and model accountability on the caltrain," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 336–348, IEEE, 2019.

[9] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Tensorscone: A secure tensorflow framework using intel sgx," *arXiv preprint arXiv:1902.04413*, 2019.

[10] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," *arXiv preprint arXiv:1806.03287*, 2018.

[11] V. Costan and S. Devadas, "Intel sgx explained," *Cryptology ePrint Archive*, 2016.

[12] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution.," *Hasp@ isca*, vol. 10, no. 1, 2013.

[13] S. Gueron, "A memory encryption engine suitable for general purpose processors," *Cryptology ePrint Archive*, 2016.

[14] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure:{SGX} cache attacks are practical," in *11th USENIX workshop on offensive technologies (WOOT 17)*, 2017.

[15] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, pp. 1–6, 2017.

[16] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2421–2434, 2017.

[17] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Trans. Intell. Syst. Technol.*, Jan. 2019.

[18] J. Konečnỳ, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *arXiv preprint arXiv:1610.02527*, 2016.

[19] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proceedings of ACM CCS*, 2015.

[20] Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi, "Beyond inferring class representatives: User-level privacy leakage from federated learning," in *Proceedings of IEEE INFOCOM*, 2019.

[21] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," *Advances in neural information processing systems*, vol. 32, 2019.

[22] J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients-how easy is it to break privacy in federated learning?," *Advances in neural information processing systems*, vol. 33, pp. 16937–16947, 2020.

[23] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[24] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," *Advances in neural information processing systems*, vol. 30, 2017.

[25] J. So, B. Güler, and A. S. Avestimehr, "Byzantine-resilient secure federated learning," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 7, pp. 2168–2181, 2020.

[26] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," in *International conference on artificial intelligence and statistics*, pp. 2938–2948, PMLR, 2020.

[27] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving google keyboard query suggestions," *arXiv preprint arXiv:1812.02903*, 2018.

[28] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for federated learning on user-held data," *arXiv preprint arXiv:1611.04482*, 2016.

[29] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1175–1191, 2017.

[30] H. Masuda, K. Kita, Y. Koizumi, J. Takemasa, and T. Hasegawa, "Model fragmentation, shuffle and aggregation to mitigate model inversion in federated learning," in *2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pp. 1–6, IEEE, 2021.

[31] Y. Zhao and H. Sun, "Information theoretic secure aggregation with user dropouts," *IEEE Transactions on Information Theory*, vol. 68, no. 11, pp. 7471–7484, 2022.

[32] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly) logarithmic overhead," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1253–1269, 2020.

[33] J. So, B. Güler, and A. S. Avestimehr, "Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning," *IEEE Journal on Selected Areas in Information Theory*, vol. 2, no. 1, pp. 479–489, 2021.

[34] G. Xu, H. Li, S. Liu, K. Yang, and X. Lin, "Verifynet: Secure and verifiable federated learning," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 911–926, 2019.

[35] X. Guo, Z. Liu, J. Li, J. Gao, B. Hou, C. Dong, and T. Baker, "V eri fl: Communication-efficient and fast verifiable aggregation for federated learning," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1736–1751, 2020.

[36] I. Ergun, H. U. Sami, and B. Guler, "Sparsified secure aggregation for privacy-preserving federated learning," *arXiv preprint arXiv:2112.12872*, 2021.

[37] S. Kadhe, N. Rajaraman, O. O. Koyluoglu, and K. Ramchandran, "Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning," *arXiv preprint arXiv:2009.11248*, 2020.

[38] B. Choi, J.-y. Sohn, D.-J. Han, and J. Moon, "Communication-computation efficient secure aggregation for federated learning," *arXiv preprint arXiv:2012.05433*, 2020.

[39] R. Schlegel, S. Kumar, E. Rosnes, and A. G. i Amat, "Codedpaddedfl and codedsecagg: Straggler mitigation and secure aggregation in federated learning," *IEEE Transactions on Communications*, 2023.

[40] Y. Ma, J. Woods, S. Angel, A. Polychroniadou, and T. Rabin, "Flamingo: Multi-round single-server secure aggregation with applications to private federated learning," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 477–496, IEEE, 2023.

[41] K. Cui, X. Feng, L. Wang, H. Wu, X. Zhang, and B. Düdder, "Chu-ko-nu: A reliable, efficient, and anonymously authentication-enabled realization for multi-round secure aggregation in federated learning," *arXiv preprint arXiv:2402.15111*, 2024.

[42] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[43] A. Roy Chowdhury, C. Guo, S. Jha, and L. van der Maaten, "Eiffel: Ensuring integrity for federated learning," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2535–2549, 2022.

[44] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pp. 427–438, IEEE, 1987.

[45] S. Gao, "A new algorithm for decoding reed-solomon codes," in *Communications, information and network security*, pp. 55–68, Springer, 2003.

[46] E. van Kempen, Q. Li, G. A. Marson, and C. Soriente, "Lisa: Lightweight single-server secure aggregation with a public source of randomness," *arXiv preprint arXiv:2308.02208*, 2023.

[47] J. So, C. He, C.-S. Yang, S. Li, Q. Yu, R. E Ali, B. Guler, and S. Avestimehr, "Lightsecagg: a lightweight and versatile design for secure aggregation in federated learning," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 694–720, 2022.

[48] T. Jahani-Nezhad, M. A. Maddah-Ali, S. Li, and G. Caire, "Swiftagg: Communication-efficient and dropout-resistant secure aggregation for federated learning with worst-case security guarantees," in *2022 IEEE International Symposium on Information Theory (ISIT)*, pp. 103–108, IEEE, 2022.

[49] T. Jahani-Nezhad, M. A. Maddah-Ali, S. Li, and G. Caire, "Swiftagg+: Achieving asymptotically optimal communication loads in secure aggregation for federated learning," *IEEE Journal on Selected Areas in Communications*, vol. 41, no. 4, pp. 977–989, 2023.

[50] H. Lycklama, L. Burkhalter, A. Viand, N. Küchler, and A. Hithnawi, "Rofl: Robustness of secure federated learning," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 453–476, IEEE, 2023.

[51] T. Jahani-Nezhad, M. A. Maddah-Ali, and G. Caire, "Byzantine-resistant secure aggregation for federated learning based on coded computing and vector commitment," *arXiv e-prints*, pp. arXiv–2302, 2023.

[52] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 1215–1225, PMLR, 2019.

[53] R. J. McEliece and D. V. Sarwate, "On sharing secrets and reed-solomon codes," *Communications of the ACM*, vol. 24, no. 9, pp. 583–584, 1981.

[54] D. J. C. Shu Lin, *Error control coding: fundamentals and applications*. Pearson/Prentice Hall, 2004.

[55] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of ACM CCS*, 2017.

[56] W. Diffe, "New direction in cryptography," *IEEE Trans. Information. Theory*, vol. 22, pp. 472–492, 1976.

[57] M. Blum and S. Micali, "How to generate cryptographically strong sequences of pseudo random bits," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pp. 227–240, 2019.

[58] A. C. Yao, "Theory and application of trapdoor functions," in *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*, pp. 80–91, IEEE, 1982.

[59] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 531–545, Springer, 2000.

[60] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2," *University of Tennessee, Tech. Rep. CS-08-627*, vol. 23, 2008.

[61] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[62] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 1314–1324, 2019.

[63] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, B. McMahan, *et al.*, "Towards federated learning at scale: System design," *Proceedings of machine learning and systems*, vol. 1, pp. 374–388, 2019.

[64] D. Minovski, N. Ögren, K. Mitra, and C. Åhlund, "Throughput prediction using machine learning in lte and 5g networks," *IEEE Transactions on Mobile Computing*, vol. 22, no. 3, pp. 1825–1840, 2021.

[65] P. Blanchard, E. M. E. Mhamdi, R. Guerraoui, and J. Stainer, "Byzantine-tolerant machine learning," *arXiv preprint arXiv:1703.02757*, 2017.