Karlsruher Institut für Technologie

KASTEL

Leveraging Multi-Party Computation for Privacy-Preserving Machine Learning

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Yufan Jiang

aus Hangzhou

Tag der mündlichen Prüfung: 11. May 2026
1. Referent: Prof. Dr. Jörn Müller-Quade
2. Referent: Associate Professor Tianwei Zhang, Ph.D

Acknowledgments

First and foremost, I am deeply grateful to Jörn Müller-Quade for taking me on as his doctoral student and supporting me throughout my studies. I still remember how he sent detailed emails to elaborate on his ideas. And I truly enjoy exploring those creative and interesting concepts with him. To me, Jörn is far more than just a supervisor.

I would like to express my sincere gratitude to my wife, Huilin Zhang, for her constant support over all these years. We first met when we were pursuing our master's degrees. She stood by me unconditionally and provided tremendous effort to our life. We often imagined how our life would change once I completed my doctoral studies Now this new chapter has finally begun.

A special thank you goes to Yong Li. He supported me throughout the first three years of my PhD and shared the latest research insights with me. I am also grateful that he trusted his intuition, which resulted in our very first joint paper.

Next, I would like to thank Tianxiang Dai for his great support with our publications. We have worked on numerous projects together. Without him, my doctoral journey would not have been so enjoyable and smooth. He generously shared his experience and programming skills without reservation. It has been a great pleasure to work with him.

I have engaged in many insightful discussions on the universal composability framework with Christian Martin and Jeremias Mechler. Beyond proofreading my thesis, Christian and I also exchanged ideas extensively across various topics. These conversations have greatly assisted me in completing numerous research projects.

Together with Saskia Bayreuther, Robin Berger, Felix Dörre and Eva Hetzel, I have been working on the same project for two years. I truly appreciate our many valuable discussions and our joint efforts in writing the paper.

Carmen Manietta and Holger Hellmuth have provided ongoing support for my work ever since I joined the team. I cannot imagine how things would have gone without their help. I also wish to thank Andy Rupp for the numerous productive meetings we held together.

I would also like to express my gratitude to my colleagues Laurin Benz, Robert Brede, Dennis Faust, Valerie Fetzer, Clemens Fruböse, Michael Klooß, Astrid Ottenhues, Johannes Ottenhues, Markus Raiber and Marcel Tiepelt for their constant kindness and support.

Finally, I would like to thank all those involved in my doctoral journey for their contributions. I especially thank Tianwei Zhang for traveling from Singapore to Germany and serving as my second reviewer.

Abstract

Machine Learning (ML) is no longer merely a research topic. At the time of writing this thesis, ML has become a well-studied and mature field, enabling the development of real-world industrial applications based on trained models, such as face recognition, fraud detection, recommendation systems, and more. In parallel, Generative AI, with models such as ChatGPT proposed by OpenAI, has gained significant attention, rapidly transforming the way people approach and perform everyday tasks.

To train an effective machine learning model, a well-structured and comprehensive dataset is an essential prerequisite. In real-world applications, large companies such as Google and Amazon often collect and own extensive training datasets from their end users, enabling them to train models independently. However, even for these large companies, datasets are typically collected within the scope of their own business domains and can thus be further improved by integrating with datasets from other sources. However, due to strict privacy regulations such as the General Data Protection Regulation (GDPR), companies are prohibited from directly sharing their data with others, and in some cases, business considerations may further discourage them from doing so. As a result, the following research question arises: *Can different entities collaboratively train a machine learning model without exposing their private datasets?*

Privacy-preserving machine learning (PPML) provides cryptographic mechanisms that enable model training without revealing the training data. Within the domain of PPML, different entities can collaboratively train a machine learning model from scratch or aggregate their local training results in a federated learning (FL) scenario. To achieve this goal, we apply secure multi-party computation (MPC) techniques, which enable distrustful parties to jointly evaluate functions without revealing their private inputs.

In this thesis, we investigate how privacy-preserving machine learning can be achieved using MPC protocols. Specifically, we propose and experiment with MPC protocols that are more efficient compared to existing ones. Our first contribution is a four-party secret-sharing scheme called $\mathcal{X}$-sharing, along with a set of four-party protocols built upon this scheme. We explore how four-party neuron network training can be accelerated using this new sharing method and compare its performance against existing approaches. Our second contribution is the development of new protocols for two-party training of gradient boosting decision trees (GBDT). We analyze the underlying modular protocols required for private GBDT and propose efficient two-party protocols to improve training efficiency. Our third contribution is a maliciously secure aggregation protocol designed for federated learning, which provides protection against poisoning attacks. The aggregation protocol is designed for a two-server setting, where clients efficiently share their gradient updates

with the servers, supporting them in generating message authentication codes (MACs). We prove the security of the proposed protocols within the universally composable (UC) framework.

Zusammenfassung

Maschinelles Lernen (ML) ist längst nicht mehr nur ein Forschungsthema. Zum Zeitpunkt der Abfassung dieser Arbeit ist ML schon ein gut erforschtes und ausgereiftes Gebiet, das die Entwicklung realer industrieller Anwendungen auf der Grundlage trainierter Modelle ermöglicht, wie in den Bereichen Gesichtserkennung, Betrugserkennung, Empfehlungssysteme und mehr. Parallel dazu hat die generative KI mit Modellen wie ChatGPT, das von OpenAI entwickelt wurde, erhebliche Aufmerksamkeit erlangt und verändert rasant die Art und Weise, wie Menschen alltägliche Aufgaben angehen und ausführen.

Um ein leistungsfähiges ML-Modell zu trainieren, ist ein gut strukturiertes und umfassendes Datensatz eine wesentliche Voraussetzung. In realen Anwendungen sammeln und besitzen große Unternehmen wie Google und Amazon oft umfangreiche Trainingsdaten ihrer Endnutzer, was es ihnen ermöglicht, Modelle eigenständig zu trainieren. Selbst bei diesen großen Unternehmen werden Datensätze jedoch in der Regel im Rahmen ihrer jeweiligen Geschäftsbereiche erhoben und können daher durch die Integration externer Datensätze weiter verbessert werden. Aufgrund strenger Datenschutzvorschriften wie der Datenschutz-Grundverordnung (DSGVO) ist es Unternehmen jedoch untersagt, ihre Daten direkt mit anderen zu teilen. In manchen Fällen verhindern auch geschäftliche Interessen eine Datenweitergabe. Daraus ergibt sich die folgende Forschungsfrage: *Können verschiedene Parteien gemeinsam ein maschinelles Lernmodell trainieren, ohne ihre privaten Datensätze offenzulegen?*

Datenschutzwahrendes maschinelles Lernen (Privacy-Preserving Machine Learning, PPML) stellt kryptographische Mechanismen bereit, die das Trainieren von Modellen ermöglichen, ohne die Trainingsdaten offenzulegen. Im Bereich von PPML können verschiedene Parteien gemeinsam ein maschinelles Lernmodellvon Grund auf trainieren oder ihre lokalen Trainingsergebnisse im Rahmen eines föderierten Lernens (FL) zusammenführen. Zur Erreichung dieses Ziels verwenden wir Techniken der sicheren Mehrparteienberechnung (Secure Multi-Party Computation, MPC), die es einander misstrauenden Parteien ermöglichen, gemeinsam Funktionen auszuwerten, ohne ihre privaten Eingabedaten offenzulegen.

In dieser Arbeit untersuchen wir, wie PPML mithilfe von MPC Protokollen realisiert werden kann. Konkret schlagen wir MPC-Protokolle vor und evaluieren sie experimentell im Hinblick auf ihre Effizienz gegenüber bestehenden Ansätzen. Unser erster Beitrag besteht in der Entwicklung eines Vier-Parteien-Secret-Sharing-Schemas, genannt $\mathcal{X}$-Sharing, sowie eine Reihe darauf basierender Protokolle. Wir untersuchen, wie das Training neuronaler Netze im Vier-Parteien-Setting durch dieses neue Verfahren beschleunigt werden kann, und vergleichen dessen Leistung mit bestehenden Ansätzen. Unser zweiter Beitrag

besteht in der Entwicklung neuer Protokolle für das Zwei-Parteien-Training von Gradient Boosting Decision Trees (GBDT). Wir analysieren die zugrunde liegenden modularen Protokolle, die für privates GBDT erforderlich sind, und schlagen effiziente Zwei-Parteien-Protokolle zur Verbesserung der Trainingseffizienz vor. Unser dritter Beitrag ist ein gegen bösartige Teilnehmer sicheres Aggregationsprotokoll für föderiertes Lernen, das vor Vergiftungsangriffen schützt. Das Protokoll ist für ein Zwei-Server-Modell konzipiert, bei dem Clients ihre Gradienten effizient mit den Servern teilen und diese bei der Generierung von Message Authentication Codes (MACs) unterstützen. Wir beweisen die Sicherheit der vorgeschlagenen Protokolle im Rahmen des Universell Komponierbaren (UC) Sicherheitsmodells.

Contents

List of Figures

List of Tables

Part I.

Background

1. Introduction

Machine learning (ML) has become one of the most prominent topics in 2025. Nowadays, large language models (LLMs) like ChatGPT [131] significantly influence the way people work and provide valuable assistance with everyday tasks, such as troubleshooting and information retrieval. On the one hand, a well-structured and comprehensive database is essential for developing an effective machine learning model, as both the quality and quantity of data directly influence the model's performance [66]. On the other hand, the importance of privacy and data security has also been increasingly recognized, with more than 70% of countries and regions worldwide having enacted legislation addressing these concerns [162]. The most secure way to keep data is to store it locally and never share it with anyone. However, this approach also poses a challenge to data usability when it is shared across several entities.

Secure multi-party computation (MPC) is designed to enable multiple parties to jointly compute a function without revealing their private inputs. Several related works [73, 88, 101, 105, 119, 120, 156, 163, 164, 169] have explored the application of MPC techniques to real-world, high-throughput scenarios, particularly in privacy-preserving machine learning (PPML). Starting from the two-party computation (2PC) case, various frameworks [49, 73, 85, 120, 137] have been proposed, offering trade-offs between security and performance. Recent works [4, 119] extend 2PC to three-party computation (3PC) by designing protocols that tolerate at most one corrupted party among the participants. The adversarial threshold is now relaxed from a dishonest-majority setting, where the adversary may corrupt more than half of the participating parties, to an honest-majority setting, where a strict majority of the parties (i.e. more than half) are assumed to behave honestly. The relaxed adversarial assumptions lead to more efficient protocols for 3PC compared to those for 2PC [40], raising the following scientific research question: *Can four-party computation (4PC) be non-trivially faster than 3PC?* In Chapter 4, we provide an affirmative answer to this question and demonstrate the application of our novel 4PC protocols to both the training and inference of convolutional neural networks (CNNs) [67, 93, 152]. Our contribution in Chapter 4 can be summarized as follows:

- We propose a new 4PC $\mathcal{X}$-sharing scheme (Section 4.2.1), and we show that in the honest-majority setting, 4PC protocols, particularly the matrix multiplication protocol, can be implemented more efficiently than existing solutions. We also show that our $\mathcal{X}$-sharing scheme is compatible with the free-truncation technique proposed in SecureML [120]. In addition, we adapt the edabits proposed in [54] to $\mathcal{X}$-dabit (Section 4.4) and construct novel share conversion and comparison protocols based on it.

References for Chapter 4

Chapter 4 was written based on the following publications.

Publication:

Tianxiang Dai, Li Duan, Yufan Jiang, Yong Li, Fei Mei, and Yulian Sun. "Force: Highly Efficient Four-Party Privacy-Preserving Machine Learning on GPU". in: *Secure IT Systems: 28th Nordic Conference, NordSec 2023, Oslo, Norway, November 16–17, 2023, Proceedings*. Oslo, Norway: Springer-Verlag, 2023, pp. 330–349. ISBN: 978-3-031-47747-8. DOI: 10.1007/978-3-031-47748-5_18. URL: https://doi.org/10.1007/978-3-031-47748-5_18

ePrint:

Tianxiang Dai, Li Duan, Yufan Jiang, Yong Li, Fei Mei, and Yulian Sun. *Force: Highly Efficient Four-Party Privacy-Preserving Machine Learning on GPU*. Cryptology ePrint Archive, Paper 2023/493. 2023. URL: https://eprint.iacr.org/2023/493

Contribution: Equal.

- We implement our framework Force [42] on top of Piranha [169] and conduct fair comparisons between different systems under the same setting. We evaluate existing solutions including Force, with different models and dataset sizes. An overview of the evaluation is provided in Section 4.7.

Apart from CNNs, gradient boosting decision trees (GBDT) [57] and its variant XGBoost [34] are widely used in real applications, such as online marketing [104, 116], risk management [157], fraud detection [143] and recommendation systems [146]. While the above techniques can be directly applied to enable private GBDT training, efficiency remains the primary concern. Recent works [36, 56, 108, 173] propose various solutions to improve efficiency, particularly focusing on a single module known as secure bucket aggregation, which is widely regarded as the most time-consuming component of private GBDT training. In Chapter 5, we propose a pure MPC-based secure bucket aggregation protocol, which outperforms other related works in the *online* computation. Our contribution in the first part of Chapter 5 can be summarized as follows:

- We present a novel two-party bucket aggregation protocol Π_{KeyBuc} (Section 5.2.5). Our protocol does not rely on additive homomorphic encryption as used in previous works [36, 108], but achieves improved communication efficiency while maintaining provable security.

Function Secret Sharing (FSS) schemes [18, 19, 20, 21] are designed to optimize the *online* phase of computation. These schemes rely on a *preprocessing* stage, during which a trusted dealer distributes correlated randomness to the parties and thus significantly accelerate the *online* computation by constructing FSS computation gates. In the remainder of Chapter 5,

References for Chapter 5

Portions of Chapter 5 were written based on the following publications.

Publication:

Tianxiang Dai, Yufan Jiang, Yong Li, and Fei Mei. " NodeGuard: A Highly Efficient Two-Party Computation Framework for Training Large-Scale Gradient Boosting Decision Tree ". In: *2024 IEEE Security and Privacy Workshops (SPW)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 95–103. DOI: 10.1109/SPW63631.2024.00015. URL: https://doi.ieeecomputersociety.org/10.1109/SPW63631.2024.00015

ePrint:

Tianxiang Dai, Yufan Jiang, Yong Li, and Fei Mei. *NodeGuard: A Highly Efficient Two-Party Computation Framework for Training Large-Scale Gradient Boosting Decision Tree*. Cryptology ePrint Archive, Paper 2024/535. 2024. URL: https://eprint.iacr.org/2024/535

Contribution: Equal.

we examine how to implement the modular protocols required for private GBDT training using FSS-based constructions. Our contribution can be summarized as follows:

- We model the FSS gates proposed in [18, 19, 20, 21] as an ideal functionality $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ within the Universal Composability (UC) framework [29], so that $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ can be called as sub-protocols.
- We use $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ to construct a silent bucket aggregation protocol Π_{SiBuc} in Section 5.4.1 and a silent argmax protocol Π_{SiArg} in Section 5.5.1. We then adapt the pure secret-sharing-based node split approach to an FSS-based approach, as described in Section 5.4.3. Combining all above techniques, we propose our FSS-based GBDT training algorithm in Section 5.6.

Federated Learning (FL) [115] was introduced to train a global machine learning model across multiple decentralized data sources while preserving data privacy, as local datasets are not shared with other participants. However, a wide range of research has shown that simply receiving gradients may allow the central server to infer sensitive information from local datasets, enabling various inference attacks [7, 47, 70, 71, 125]. At the same time, FL is also vulnerable to poisoning attacks. Malicious participants may inject corrupted updates into the training process to degrade the performance of the global model [14, 149, 154], or embed backdoors that can be exploited at inference time [63, 129, 155, 165]. To address the above issues, we follow state-of-the-art works [37, 110, 141] that employ *secure aggregation*

Reference for Chapter 5

Portions of Chapter 5 were written based on the following publication.

Publication:

Yufan Jiang, Fei Mei, Tianxiang Dai, and Yong Li. "SiGBDT: Large-Scale Gradient Boosting Decision Tree Training via Function Secret Sharing". In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 274–288. ISBN: 9798400704826. DOI: 10.1145/3634737.3657024. URL: https://doi.org/10.1145/3634737.3657024

Contribution: Main.

References for Chapter 6

Chapter 6 was written based on the following publications.

Publication:

Yufan Jiang, Maryam Zarezadeh, Tianxiang Dai, and Stefan Köpsell. "AlphaFL: Secure Aggregation with Malicious2 Security for Federated Learning against Dishonest Majority". In: *Proceedings on Privacy Enhancing Technologies* 2025 (4), pp. 348–368. DOI: https://doi.org/10.56553/popets-2025-0134

ePrint:

Yufan Jiang, Maryam Zarezadeh, Tianxiang Dai, and Stefan Köpsell. *AlphaFL: Secure Aggregation with Malicious^2 Security for Federated Learning against Dishonest Majority*. Cryptology ePrint Archive, Paper 2025/1289. 2025. URL: https://eprint.iacr.org/2025/1289

Contribution: Main.

and *norm-based defenses* [1], while shifting the focus toward achieving *malicious security* from cryptographic point of view. In Chapter 6, we construct a two-server framework, similar to the distributed server setting proposed in [127, 141]. Our contribution can be summarized as follows:

- We investigate how to securely and efficiently enable the servers to perform the SPD$\mathbb{Z}_{2^k}$ scheme in collaboration with clients, that are supposed to have limited computational resources. By applying the SPD$\mathbb{Z}_{2^k}$ scheme, clients are not required to participate in any computation beyond the input phase. We propose an input

[1] In this thesis, we focus on achieving malicious security from a cryptographic perspective, while treating the improvement of poisoning defense performance as an orthogonal research direction.

commitment protocol Π^{InCom} in Section 6.2.1 for clients to efficiently shares their input and help servers to generate information-theoretically secure message authentication code (MAC) shares. We provide detailed mathematical proofs to show that the probability of successfully introducing errors into MAC shares is negligible.

- We identify a subtlety in modeling functionalities for SPD$\mathbb{Z}_{2^k}$ scheme, which we elaborate on in Section 6.3.3. In a nutshell, a functionality might not be able to compute the correct MAC shares. Since the global MAC key is primally determined by the preprocessing functionality, the only option for other functionality to obtain the global MAC key is to reconstruct it by receiving input shares and input MAC shares, which could lead to inconsistencies. Following the approach of [39, 46], we construct a wrapper functionality $\mathcal{F}^{\mathsf{Wrap}}$ as described in Fig. A.7.

2. Related Work

2.1. Privacy-Preserving Machine Learning on Convolutional Neural Network with Graphics Processing Unit

In a two-party computation (2PC) setting, SecureML applies multi-party computation (MPC) techniques to privately train a neural network (NN). It adopts a mixed-protocol structure of ABY [49], using correlated randomness generated during a preprocessing phase to accelerate the online computation. Follow-up works such as miniONN [105] secureNN [163], Falcon [164], Cheetah [73] and [68, 81, 97, 117, 140, 183] continue to adopt the mixed-protocol approach, while focusing on optimizing modular operations using various techniques. Later, ABY3 [119] investigates three-party computation (3PC) in an honest-majority setting to accelerate PPML using a *replicated* secret sharing scheme. The semi-honest and malicious security foundations of the above protocols are addressed in both [4] and [60]. Follow-up work Blaze [138] achieves *fairness*, while Swift [89] provides the *guarantee of delivery* property in the 3PC setting under the honest majority. Recent four-party computation (4PC) frameworks [24, 32, 45, 89, 90, 101] continue with a similar approach but with different focuses. [101] builds a PPML framework based on a 4PC engine with semi-honest security, providing `Python` interfaces for real-world applications. Other works [24, 32, 45, 89, 90] aim to propose provably maliciously secure 4PC protocols, often incorporating additional desirable properties such as fairness or guarantee of delivery.

While Graphics Processing Units (GPUs) have been shown to significantly accelerate machine learning training, cryptography researchers have also begun exploring their use to accelerate the execution of cryptographic protocols. Finally, these two lines meet at PPML. CrypTen [88] retains an ABY-style cryptographic core while offering PyTorch-like [136] interfaces to support machine learning practitioners. While CrypTen still uses an *n-out-of-n* secret sharing scheme, CryptGPU [156] implements a replicated secret sharing scheme in the 3PC setting. Specifically, both frameworks accelerate local computations using GPUs. Apart from that, Piranha [169] introduces a generic secret-sharing-based MPC protocol designed for GPUs, implemented in `C++`. It proposes novel engineering optimizations that enable training large-scale models such as VGG [152] using MPC, which was previously infeasible with CryptGPU or CrypTen.

For more details, we also refer the reader to the survey [126].

2.2. Privacy-Preserving Machine Learning on Gradient Boosting Decision Tree

By applying oblivious transfer (OT) [124] and Yao's garbled circuits [178], Lindell and Pinkas [103] securely train a single decision tree model on a horizontally partitioned dataset over two parties. As follow-up work, Abspoel et al. [1] and De Hoogh et al. [48] continue this line of research. Recent works [36, 56, 108, 171] focus on privacy-preserving gradient boosting decision trees (GBDT) training, assuming that the dataset is vertically distributed. They show that optimizing the secure bucket aggregation protocol significantly accelerates the private GBDT training process. SecureBoost [36] applies the Paillier encryption scheme [133] to securely compute the sum of selected vector entries, thereby enabling secure bucket aggregation. The proposed method is further optimized by Fu et al. [58] and Chen et al. [35] from communication and computational complexity perspectives. Pivot [171] retains the underlying structure of SecureBoost [36], but introduces MPC protocols to address the leakage of intermediate results. Recently, Squirrel [108] investigates how privacy-preserving GBDT training can be accelerated using a lattice-based Homomorphic Encryption (HE) scheme, specifically learning with errors (LWE) and its ring variant (RLWE). While the above works employ HE for secure bucket aggregation, Xie et al. [173] and Fang et al. [56] instead rely on two-party multiplication protocols to implement this module. At the same time, works such as [94, 98] integrate secure hardware to support privacy-preserving GBDT training. Without employing any cryptographic protocols, FederBoost [158] chooses to apply differential privacy (DP) [52, 53] as the sole mechanism for protecting data privacy.

2.3. Privacy-Preserving Federated Learning and Poisoning Attacks

Federated learning (FL) [115] is first proposed by Google to train machine learning models across multiple data sources without centralizing data. FL is designed to safeguard data privacy, which however often fails in practice. Recent works [16, 17, 37, 110, 141] show that simply receiving model gradients can allow an adversary to infer sensitive information from clients' local datasets, even though the data itself is never revealed in plaintext. A wide range of research has examined and explored various inference attacks that could compromise the privacy of FL systems [7, 47, 70, 71, 125]. Apart from privacy issue, FL is also vulnerable to poisoning attacks. A malicious adversary in FL can compromise the global model in two ways. First, it can inject harmful data into its own local training dataset [87, 150, 151, 159], as its private data is not subject to verification by other participants. Second, it may maliciously submit altered gradient updates to the central server [6, 55, 113, 166], which lacks the means to detect such behavior.

Without considering data privacy, several mitigation methods against poisoning attacks have been explored [99, 107, 122, 123, 177, 184]. One such approach is to compare sub-

mitted gradient updates with benign ones and filter out outliers [5, 30, 50, 59, 113, 114]. Clustering [33, 127, 176] and anomaly detection [3, 80] are also considered effective means for identifying and filtering malicious updates. Frameworks such as Krum [15] and Median/Mean [179] propose Byzantine-robust FL aggregation rules and prove that the global model still converges when the corresponding aggregation rules are applied. Another approach is gradient clipping and noising, which smooths model updates and mitigates update discrepancies [100, 172]. These aggregation rules are later attacked and partially mitigated by Fang et al. [55].

As an independent research direction orthogonal to the above works, secure aggregation (particularly in a single-server setting) has been extensively studied to preserve user privacy [17]. Subsequent works extend these approaches to additionally guarantee output integrity [65, 142, 167, 186]. SecAgg [17] combines masking, Shamir's secret sharing [147] and symmetric encryption to protect local models from unauthorized access. VerifyNet [174] and VeriFL [64] build on top of SecAgg [17], incorporating additional verifiability features to ensure the accuracy of aggregation. SecAgg+ [13], SVFL [109], and Flamingo [112] attempt to improve efficiency by using masking techniques. E-SeaFL [11] applies authenticated homomorphic vector commitments to generate a proof of correct aggregation. Huang et al. [72] and Yu et al. [180] achieve result correctness verification by involving a trusted third party. Meanwhile, Pasquini et al. [135] introduce several attacks in the presence of a malicious server.

Recently, several works have proposed secure aggregation protocols that achieve poisoning resilience against malicious adversaries in FL. Prio [38], Prio+ [2] and Eiffel [37] use secret-shared non-interactive proofs (SNIP) to validate clients' input, thereby filtering out potentially malicious gradient updates. RoFL [110] provides input privacy and an enforcement of norm-based defenses by applying expensive non-interactive zero-knowledge proofs (NIZK), specifically Bulletproofs [23]. Instead, Acorn [12] proposes using range proofs, while Flag [8] enhances security against adaptive adversaries. In a distributed-server setting, MLGuard [86], FLGuard [128] and SafeFL [61] design MPC protocols among computation servers to detect malicious inputs. Elsa [141] breaks traditional server-client application model by accelerating online computation through offloading oblivious transfer correlations and Beaver triples to clients. Prior works also apply techniques such as DP [78, 106, 161, 170, 185], trusted execution environments (TEE) [82, 118, 121, 139], homormorphic encryption (HE) [26, 79, 134, 148, 181], zero knowledge proofs (ZKPs) [51, 130, 168] and hybrid approaches [25, 153, 160, 175] to counter corrupted actors in FL.

Part II.

Main Protocols

3. Preliminaries

This chapter is taken verbatim from the publications [40, 43, 75, 77].

3.1. Notations

In this thesis, we use boldface letters (e.g., $\mathbf{x}$) to denote vectors, where $\mathbf{x} = (x_0, \ldots, x_{n-1})$. We use 0-based indexing throughout this thesis unless stated otherwise. In particular, whenever we write $i \in [n]$, we mean $i \in \{0, \ldots, n-1\}$. We use P_i to denote the i-th party and P_c to denote a corrupted party. The i-th element of $\mathbf{x}$ is denoted by x_i, and we may also write $\mathbf{x}[i]$ to refer to the same element. If a party defines $\tilde{\mathbf{x}} = (\mathbf{x}, x_t)$, this denotes the vector obtained by appending the element x_t to $\mathbf{x}$. We write the dot product of vectors $\mathbf{x}$ and $\mathbf{y}$ as $\mathbf{x} \cdot \mathbf{y}$. We write the element-wise multiplication of vectors $\mathbf{x}$ and $\mathbf{y}$ as $\mathbf{x} \circ \mathbf{y}$. We write the element-wise AND of two Boolean vectors $\mathbf{x}$ and $\mathbf{y}$ as $\mathbf{x} * \mathbf{y}$. We also write the scalar product of a scalar x and a vector $\mathbf{y}$ as $x \circ \mathbf{y}$.

3.2. Fixed-Point Computation

Let p denote the number of fractional bits in the fixed-point representation. Let $\lfloor \cdot \rfloor$ denote the floor function. We define $x \in \mathbb{Z}_{2^k}$ as the fixed-point representation of $\tilde{x} \in \mathbb{R}$ satisfying the condition $x = \lfloor \tilde{x} \cdot 2^p \rfloor$, where x is an k-bit integer using two's complement. Thus, x consists of an integer part for $k - p$ bits and a fraction part for p bits. Addition and subtraction can be directly performed on $\mathbb{Z}_{2^k}$. However, the result of multiplication has to be divided by 2^p to keep the shifted magnitude consistent [120].

3.3. Secret Sharing

3.3.1. n-out-of-n Sharing

Definition 3.3.1 (*n-out-of-n Sharing*)**.** An n-out-of-n sharing of a value $x \in \mathbb{Z}_{2^k}$ over $\mathbb{Z}_{2^k}$ consists of shares

$$(x_0, x_1, \ldots, x_{n-1}) \in \mathbb{Z}_{2^k}^n ,$$

such that

$$\sum_{i=0}^{n-1} x_i \equiv x \pmod{2^k}.$$

Each party P_i holds the corresponding local share x_i.

We denote this sharing scheme as $[\cdot]$ in 2PC case and $[\cdot]_{\mathsf{non}}$ in nPC case (e.g. $[\cdot]_{3o3}$). We let $[\cdot]_i$ denote the local share of P_i. For the rest of this section, we use $[\cdot]$ as an example.

Addition (Subtraction). Suppose parties hold $[x]$ and $[y]$. To compute $[z] = [x + y]$, each P_i locally sets

$$[z]_i = [x]_i + [y]_i \bmod 2^k.$$

Subtraction is handled analogously.

Scalar Multiplication. Let $c \in \mathbb{Z}_{2^k}$ be a public constant known to all parties. Then the scalar multiplication $[z] = [cx]$ can be computed locally by P_i as

$$[z]_i = c \cdot [x]_i \bmod 2^k.$$

Multiplication. Unlike addition (subtraction) and scalar multiplication, multiplying two shared values $[x]$ and $[y]$ requires parties to communicate with each other. Given a Beaver triple [10, 49, 85] ($[a]$, $[b]$, $[c]$) satisfying $c = ab \bmod 2^k$, parties can reconstruct

$$e = x - a \bmod 2^k, \qquad f = y - b \bmod 2^k,$$

then compute their result shares as

$$[z] = [c] + e[b] + f[a] + ef.$$

Division. To divide $[x]$ by $[y]$, parties have to first compute the reciprocal $[1/y]$ of $[y]$ then multiply $[1/y]$ by $[x]$. As in previous works [56, 108], we use the Goldschmidt division algorithm [62] to compute an approximate reciprocal. We adopt the same parameters as Catrina and Saxena [31] for the secure normalization protocol when computing the initial approximation. As shown in their work, refining the jointly computed initial approximation using only two Goldschmidt iterations yields a relative error bounded by $|e_2| < (0.08578)^4$, which is approximately 2^{-14} in magnitude for the final result (ignoring truncation effects).

Boolean secret sharing. Let $x \in \mathbb{Z}_2$. A Boolean sharing of x, denoted by $\langle x \rangle$, consists of two shares $\langle x \rangle_1, \langle x \rangle_2 \in \mathbb{Z}_2$ such that $x = \langle x \rangle_1 \oplus \langle x \rangle_2$.

3.3.2. Zero Sharing

Definition 3.3.2 (*Zero Sharing*)**.** An n-party *zero sharing* over $\mathbb{Z}_{2^k}$ consists of shares

$$(r_0, r_1, \ldots, r_{n-1}) \in \mathbb{Z}_{2^k}^n,$$

such that

$$\sum_{i=0}^{n-1} r_i \equiv 0 \pmod{2^k}.$$

Each party P_i holds the corresponding local share r_i.

Such a *zero sharing* can be obtained by evaluating a keyed pseudorandom function (PRF) with pre-shared keys [4, 156].

3.3.3. Replicated Sharing

Definition 3.3.3 (Replicated Sharing in 3PC)**.** Let $x \in \mathbb{Z}_{2^k}$. A *replicated sharing* of x among three parties consists of shares $x_0, x_1, x_2 \in \mathbb{Z}_{2^k}$ such that

$$x = x_0 + x_1 + x_2 \bmod 2^k,$$

where each party P_i holds the pair (x_i, x_{i+1}).

We denote such a sharing by $[x]_{\mathsf{RS}}$. Addition and subtraction of two replicated shares $[x]_{\mathsf{RS}}$ and $[y]_{\mathsf{RS}}$ can be locally computed by parties. In contrast, multiplication of replicated shares requires interaction. Given $[x]_{\mathsf{RS}}$ and $[y]_{\mathsf{RS}}$, each party P_i can locally compute

$$z_i = x_i y_i + x_{i+1} y_i + x_i y_{i+1} \bmod 2^k,$$

which yields a 3-out-of-3 additive sharing $[xy]_{3\text{o}3}$, i.e.,

$$xy = z_0 + z_1 + z_2 \bmod 2^k.$$

To convert this 3-out-of-3 sharing into a replicated sharing $[xy]_{\mathsf{RS}}$, the parties execute a *reshare* protocol. Specifically, they jointly generate a three-party *Zero Sharing* $(r_0, r_1, r_2) \in \mathbb{Z}_{2^k}$ such that

$$r_0 + r_1 + r_2 \equiv 0 \pmod{2^k}.$$

Then P_i sends the masked value $z_i + r_i$ to party P_{i+1}. The above procedure is formally specified in [4, 119].

More generally, we call a secret sharing scheme *replicated* if at least one local share component is held by more than one party.

3.3.4. SPD$\mathbb{Z}_{2^k}$

In SPD$\mathbb{Z}_{2^k}$ [39], an information-theoretic MAC scheme is introduced. Each party P_j holds an additive share $\alpha^j \in \mathbb{Z}_{2^s}$ of a secret *global* MAC key $\alpha = \alpha^0 + \alpha^1$ (over $\mathbb{Z}$). An authenticated secret value $x \in \mathbb{Z}_{2^k}$ is shared between parties (in 2PC), if each party P_j holds a share $x^j \in \mathbb{Z}_{2^{k+s}}$, such that

$$x' = x^0 + x^1 \bmod 2^{k+s} \quad \text{and} \quad x \equiv x' \pmod{2^k}.$$

In addition, each party holds a MAC share $m^j \in \mathbb{Z}_{2^{k+s}}$ such that

$$m = m^0 + m^1 \bmod 2^{k+s} \quad \text{and} \quad m = \alpha \cdot x' \bmod 2^{k+s}.$$

Since α is a global MAC key, we abbreviate each local share as (x^j, m^j) and denote such an authenticated sharing by $[\![x]\!]$.

For a Boolean shared value $x \in \mathbb{Z}_2$, we write $[\![x]\!]_2$, where

$$[\![x]\!]_2^j := (x^j, m_x^j) \in \mathbb{Z}_{2^{1+s}} \times \mathbb{Z}_{2^{1+s}}.$$

Addition and multiplication over $\mathbb{Z}_{2^{1+s}}$ correspond to XOR and AND over $\mathbb{Z}_2$, respectively. As observed in [46], for Boolean multiplication it suffices to ensure correctness modulo 2, such as

$$z = x \cdot y \bmod 2,$$

and we may **not** have

$$z' = x' \cdot y' \bmod 2^{1+s}.$$

3.4. Function Secret Sharing

We follow the definition of *function secret sharing* (FSS) in [18, 20]. Unlike the traditional secret sharing schemes where a secret input x is shared, FSS applies the sharing scheme on a function $(\!|f|\!)$ and lets parties hold a masked value $\hat{x}$. FSS shares a function $f(\cdot)$ into two *function keys* $(\!|f|\!)_0$ and $(\!|f|\!)_1$ satisfying $f(x) = (\!|f|\!)_0(x) + (\!|f|\!)_1(x)$. We define FSS schemes and gates as follows:

Definition 3.4.1 (*FSS Syntax*)**.** A two-party function secret sharing (FSS) scheme is a pair of algorithms $(\mathsf{KeyGen}, \mathsf{Eval})$ satisfying the following conditions:

- $\mathsf{KeyGen}(\lambda, \hat{f})$ is a probabilistic polynomial-time key generation algorithm, which on input λ (security parameter) and $\hat{f} \in \{0,1\}^*$ (description of a function f) outputs a pair of keys (k_0, k_1). We assume that $\mathbb{G}^{\mathsf{in}}$ (input group) and $\mathbb{G}^{\mathsf{out}}$ (output group) are already included in $\hat{f}$.
- $\mathsf{Eval}(i, k_i, x)$ is a polynomial-time evaluation algorithm, which on input $i \in \{0,1\}$ (party index), k_i (i-th function key defining $(\!|f|\!)_i : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$) and $x \in \mathbb{G}^{\mathsf{in}}$ outputs $(\!|f|\!)_i(x) \in \mathbb{G}^{\mathsf{out}}$ (i-th share of $f(x)$).

Definition 3.4.2 (*FSS Correctness and Security*)**.** A pair of algorithms (KeyGen, Eval) as defined in Definition 3.4.1 is an FSS scheme for a family of function $\mathcal{F}$ if the following conditions hold:

- **Correctness:** For all $\hat{f}$ describing $f : \mathbb{G}^{\text{in}} \to \mathbb{G}^{\text{out}} \in \mathcal{F}$ and every $x \in \mathbb{G}^{\text{in}}$, if $(k_0, k_1) \leftarrow \mathsf{KeyGen}(\lambda, \hat{f})$ then $\Pr[\mathsf{Eval}(0, k_0, x) + \mathsf{Eval}(1, k_1, x) = f(x)] = 1$.
- **Security:** For each $i \in \{0, 1\}$ there is a PPT algorithm simulator $\mathcal{S}_i$, such that for every sequence $(\hat{f}_j)_{j\in\mathbb{N}}$ of polynomial-size function descriptions from $\mathcal{F}$ and polynomial-size input sequence x_j for f_j, the outputs of the following experiments Real and Ideal are computationally indistinguishable:
  - Real_j: $(k_0, k_1) \leftarrow \mathsf{Gen}(\lambda, \hat{f}_j)$; Output k_i.
  - Ideal_j: Output $\mathcal{S}_i(\lambda)$.

Definition 3.4.3 (*Offset function family and FSS gates*)**.** Let $\mathcal{G} = \{g : \mathbb{G}^{\text{in}} \to \mathbb{G}^{\text{out}}\}$ be a computation gate. The family of offset functions $\hat{\mathcal{G}}$ of $\mathcal{G}$ is given by:

$$\hat{\mathcal{G}} := \{g^{\{r^{\text{in}}, r^{\text{out}}\}} : \mathbb{G}^{\text{in}} \to \mathbb{G}^{\text{out}} \mid g : \mathbb{G}^{\text{in}} \to \mathbb{G}^{\text{out}} \in \mathcal{G}\}$$
$$\text{where } r^{\text{in}} \in \mathbb{G}^{\text{in}},\ r^{\text{out}} \in \mathbb{G}^{\text{out}} \text{ and}$$
$$g^{\{r^{\text{in}}, r^{\text{out}}\}}(x) := g(x - r^{\text{in}}) + r^{\text{out}},$$

and $g^{\{r^{\text{in}}, r^{\text{out}}\}}$ is parameterized with descriptions $\{r^{\text{in}}, r^{\text{out}}\}$. Finally, we use FSS gate for $\mathcal{G}$ to denote an FSS scheme for the corresponding offset family $\hat{\mathcal{G}}$.

Definition 3.4.4 (*Distributed Comparison Function*)**.** A special interval function $f^{<}_{\alpha,\beta}$ also referred to as a comparison function, outputs β if $x < \alpha$ and 0 otherwise. We refer to an FSS scheme for comparison function as distributed comparison function (DCF). Similarly, function $f^{\leq}_{\alpha,\beta}$ outputs β if $x \leq \alpha$ and 0 otherwise. In both cases, the minimal default leakage $\mathsf{leak}(\hat{f}) = (\mathbb{G}^{\text{in}}, \mathbb{G}^{\text{out}})$ is allowed.

3.5. Universally Composable Security

In this thesis, we consider two types of adversaries: semi-honest and malicious. A semi-honest adversary follows the protocol specification but attempts to infer additional private information from the received messages. In contrast, a malicious adversary may arbitrarily deviate from the protocol in an effort to compromise privacy or correctness. Let $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the output of an environment machine $\mathcal{Z}$ interacting with the adversary $\mathcal{A}$ executing the protocol Π in the real world. Let $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the output of $\mathcal{Z}$ interacting with a simulator $\mathcal{S}$ connected to an ideal functionality $\mathcal{F}$ in the ideal world:

Definition 3.5.1 (Universally Composable (UC) Security [28])**.** Let $\mathcal{F}$ be a functionality and let Π be a protocol that computes $\mathcal{F}$. Protocol Π is said to **uc-realize $\mathcal{F}$ in the presence of static semi-honest/malicious adversaries** if for every non-uniform probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a non-uniform PPT adversary $\mathcal{S}$, such that for any PPT environment $\mathcal{Z}$:

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \overset{c}{\equiv} \mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}},$$

where $\overset{c}{\equiv}$ denotes the computational indistinguishability.

We follow the security definitions provided by the Universal Composability (UC) framework [28]. In the hybrid model, a *uc-secure* protocol can be abstracted by an ideal functionality and invoked as a subroutine within other protocols. We let sid denote the session identifier. All subroutine calls to ideal functionalities are made under the same session identifier sid as the main protocol.

Static Corruption. We consider *static corruption* throughout the entire thesis. In [28], static corruption is defined such that the identities of adversarially controlled parties are fixed before computation begins. In particular, a corruption message sent from the adversary to the ideal functionality is only considered if it is delivered in the immediately subsequent activation.

3.6. Malicious Clients in Federated Learning

We let C_c denote the compromised clients. In the scope of federated learning, an adversary $\mathcal{B}$ may control a subset of clients and thus manipulate their local updates $\{\mathbf{u}_i\}_{i\in c}$. We formally describe the adversarial goal as follows:

Definition 3.6.1 (Compromised Model [127])**.** Let $\mathcal{M}$ be the benign model and let $\mathcal{M}'$ denote the compromised model. Let D_c denote the trigger set, where for each $x \in D_c$ there is a manipulated output z' chosen by the adversary. The model is said to be successfully compromised by the adversary, if:

$$f(\mathcal{M}', x) = \begin{cases} z' \neq f(\mathcal{M}, x) & \forall x \in D_c, \\ f(\mathcal{M}, x) & \text{Otherwise.} \end{cases}$$

In the meantime, the model $\mathcal{M}'$ should be hard to be distinguished from the benign model $\mathcal{M}$.

3.7. L_2-Norm and L_∞-Norm

The Euclidean Norm of a vector $\mathbf{x} = \{x_0, ..., x_{n-1}\}$, or L_2-Norm, is defined as $L_2(\mathbf{x}) = \sqrt{x_0^2 + ... + x_{n-1}^2}$. Due to the computation complexity, if we have bound the L_2-Norm of $\mathbf{x}$ by β, we instead bound the squared L_2-Norm by β^2. However, when working with cryptographic primitives over finite rings, merely imposing an upper bound on the L_2-Norm is inadequate for controlling the individual component magnitudes, since overflow can cause values to wrap around the modulus. To overcome this, we supplement the L_2-Norm bound with an additional component-wise upper limit based on bit length. Using the L_∞-Norm, we define $x_{\text{max}} = \max_i |x_i|$, which is now bounded by 2^{w-1} for some $w \in \mathbb{N}$. This is equivalent to bound every element in $\mathbf{x}$ by 2^{w-1}.

3.8. Malicious^2 Security

In the traditional *security with abort* paradigm [102], input validity is, by definition, out of scope. However, in federated learning (FL), a malicious adversary may not only deviate from the protocol but also corrupt a subset of clients and inject malicious inputs to manipulate the final result, as discussed in Section 3.6. To capture both aspects, we introduce the term *Malicious^2 Security*, which denotes security against malicious behavior in the UC framework as well as resilience to model poisoning attacks in the FL setting.

4. A Four-party Computation Framework for Training Convolutional Neural Network

4.1. Convolutional Neural Network

The development of deep learning and convolutional neural networks (CNNs) has brought revolution to computer vision and many other fields [67, 93, 152]. A typical CNN model consists of stacked convolutional layers combined with activation functions, optional normalization layers and pooling layers, followed by fully connected layers. In this section, we briefly introduce each type of layer and describe the specific computations performed by each.

Feature Map Tensor. Let $m \in \mathbb{N}$ denote the batch size, $d \in \mathbb{N}$ the number of feature channels, and $h, w \in \mathbb{N}$ the spatial height and width. A feature map tensor is defined as

$$\mathbf{X} \in \mathbb{R}^{m \times d \times h \times w}.$$

Convolution Layer. Let $\mathbf{X} \in \mathbb{R}^{m \times d_{\text{in}} \times h \times w}$ be the input tensor, where d_{in} denotes the number of input channels. Let the convolution kernel tensor be

$$\mathbf{K} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}} \times k_h \times k_w},$$

where d_{out} is the number of output channels and k_h, k_w denote the kernel height and width.

For stride $s \in \mathbb{N}$ and padding $p \in \mathbb{N}$, the output tensor satisfies

$$\mathbf{Y} \in \mathbb{R}^{m \times d_{\text{out}} \times h' \times w'},$$

where

$$h' = \left\lfloor \frac{h + 2p - k_h}{s} \right\rfloor + 1, \qquad w' = \left\lfloor \frac{w + 2p - k_w}{s} \right\rfloor + 1.$$

Let $\mathbf{b} \in \mathbb{R}^{d_{\text{out}}}$ be an optional bias vector. The convolution layer defines the mapping

$$\text{Conv}_{\mathbf{K},\mathbf{b}} : \mathbb{R}^{m \times d_{\text{in}} \times h \times w} \to \mathbb{R}^{m \times d_{\text{out}} \times h' \times w'}$$

such that for all $n \in \{0, \ldots, m-1\}$, $i \in \{0, \ldots, d_{\text{out}} - 1\}$, $j \in \{0, \ldots, h' - 1\}$, $k \in \{0, \ldots, w' - 1\}$,

$$y_{n,i,j,k} = b_i + \sum_{c=0}^{d_{\text{in}}-1} \sum_{a=0}^{k_h-1} \sum_{t=0}^{k_w-1} K_{i,c,a,t} \, X_{n,c,\, sj+a-p,\, sk+t-p}.$$

Entries of $\mathbf{X}$ outside the padded spatial domain are treated as zero.

Activation Layer. The activation layer introduces non-linearity. In Force, we use the ReLU activation function

$$f(x) = \max(x, 0).$$

Pooling Layer. Given pooling window size $k_h \times k_w$ and stride s, the output spatial dimensions are

$$h' = \left\lfloor \frac{h - k_h}{s} \right\rfloor + 1, \qquad w' = \left\lfloor \frac{w - k_w}{s} \right\rfloor + 1.$$

Max pooling: The max pooling layer defines

$$\text{MaxPool} : \mathbb{R}^{m \times d \times h \times w} \to \mathbb{R}^{m \times d \times h' \times w'}$$

such that for all $n \in \{0, \ldots, m-1\}$, $i \in \{0, \ldots, d-1\}$, $j \in \{0, \ldots, h'-1\}$, $k \in \{0, \ldots, w'-1\}$,

$$y_{n,i,j,k} = \max_{\substack{0 \le a \le k_h - 1 \\ 0 \le b \le k_w - 1}} x_{n,i,\, sj+a,\, sk+b}.$$

Average pooling: The average pooling layer defines

$$\text{AvgPool} : \mathbb{R}^{m \times d \times h \times w} \to \mathbb{R}^{m \times d \times h' \times w'}$$

such that

$$y_{n,i,j,k} = \frac{1}{k_h k_w} \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} x_{n,i,\, sj+a,\, sk+b}.$$

Fully Connected Layer. A fully connected layer defines the affine mapping

$$\text{FC}_{\mathbf{W},\mathbf{b}} : \mathbb{R}^n \to \mathbb{R}^m.$$

Let $\mathbf{W} \in \mathbb{R}^{m \times n}$ be the weight matrix, $\mathbf{b} \in \mathbb{R}^m$ the bias vector, and $\mathbf{x} \in \mathbb{R}^n$ the input. The output $\mathbf{y} \in \mathbb{R}^m$ is

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}.$$

Optional Normalization Layer (Batch Normalization). Let $\mathbf{X} \in \mathbb{R}^{m\times d\times h\times w}$ with entries $x_{n,i,j,k}$. For each channel $i \in \{0,\dots,d-1\}$, define the mean

$$\mu_i = \frac{1}{mhw}\sum_{n=0}^{m-1}\sum_{j=0}^{h-1}\sum_{k=0}^{w-1} x_{n,i,j,k},$$

and variance

$$\sigma_i^2 = \frac{1}{mhw}\sum_{n=0}^{m-1}\sum_{j=0}^{h-1}\sum_{k=0}^{w-1}\left(x_{n,i,j,k}-\mu_i\right)^2.$$

The normalized tensor is

$$\hat{x}_{n,i,j,k} = \frac{x_{n,i,j,k}-\mu_i}{\sqrt{\sigma_i^2+\epsilon}},$$

where $\epsilon > 0$ is a constant for numerical stability.

The BN output $\mathbf{Y} \in \mathbb{R}^{m\times d\times h\times w}$ is

$$y_{n,i,j,k} = \gamma_i \hat{x}_{n,i,j,k} + \beta_i,$$

where $\gamma_i, \beta_i \in \mathbb{R}$ are learnable parameters.

Forward and Backward Propagation. Network training consists of forward propagation and backward propagation. During forward propagation, computations are performed sequentially, layer by layer, from the input layer to the output layer according to the network architecture. Given an input sample (or mini-batch), the network produces a prediction, after which a loss function $\mathcal{L}$ is evaluated to measure the discrepancy between the predicted output and the ground-truth target. During backward propagation, gradients of the loss with respect to all learnable parameters are computed using the chain rule. The network parameters are then updated to minimize the loss. For Force, we employ Stochastic Gradient Descent (SGD) as the optimization algorithm.

4.2. Four-Party Secret Sharing and Fixed-Point Computation

4.2.1. $\mathcal{X}$-Sharing

We formally define a four-party sharing type $\mathcal{X}$-Sharing in this section. Note that $\mathcal{X}$-Sharing works over both $\mathbb{Z}_{2^k}$ for arithmetically shared value and $\mathbb{Z}_2$ for boolean shared value. In addition, a secret value x can be shared in two share-modes:

Definition 4.2.1 ($\mathcal{X}$*-Sharing in AC mode*)**.** A secret value $x \in \mathbb{Z}_{2^k}$ is said to $\mathcal{X}$-shared among the parties in AC mode, if P_0 and P_1 hold $x_0 \in \mathbb{Z}_{2^k}$, P_2 and P_3 hold $x_1 \in \mathbb{Z}_{2^k}$, such that $x_0 + x_1 = x \bmod 2^k$.

Protocol Π_{chMo}

Private inputs: Parties hold $[x]_{\psi}$.
Outputs: $[x]_{\phi}$.
Preprocessing: P_i sends $(\mathsf{CMSGen}, \psi\mathrm{to}\phi, 0, \mathsf{P}_i, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$, receives $[\![r]\!]_{\psi\mathrm{to}\phi}^{\mathsf{P}_i}$ as output.
Protocol:

- If $\psi = \mathsf{AC}$ and $\phi = \mathsf{AB}$:
  1. P_0 computes $d_0 = [x]_{\psi}^{\mathsf{P}_0} + [\![r]\!]_{\psi\mathrm{to}\phi}^{\mathsf{P}_0} \bmod 2^k$, then sends d_0 to P_2.
  2. P_2 computes $d_1 = [x]_{\psi}^{\mathsf{P}_2} + [\![r]\!]_{\psi\mathrm{to}\phi}^{\mathsf{P}_2} \bmod 2^k$, then sends d_1 to P_0.
  3. Upon receiving d_0 and d_1, P_0 and P_2 set $[x]_{\phi}^{\mathsf{P}_0} = d_0 + d_1 \bmod 2^k$ and $[x]_{\phi}^{\mathsf{P}_2} = d_0 + d_1 \bmod 2^k$, respectively.
  4. P_1 and P_3 set $[x]_{\phi}^{\mathsf{P}_1} = [\![r]\!]_{\psi\mathrm{to}\phi}^{\mathsf{P}_1}$, $[x]_{\phi}^{\mathsf{P}_3} = [\![r]\!]_{\psi\mathrm{to}\phi}^{\mathsf{P}_3}$ respectively.
- Otherwise if $\psi = \mathsf{AB}$ and $\phi = \mathsf{AC}$, switch the role of P_2 with P_1, do the same as above.

Figure 4.1.: Four-Party Change Share-Mode Protocol

Definition 4.2.2 ($\mathcal{X}$*-Sharing in AB mode*)**.** A secret value $x \in \mathbb{Z}_{2^k}$ is said to $\mathcal{X}$-shared among the parties in AB mode, if P_0 and P_2 hold $x_0 \in \mathbb{Z}_{2^k}$, P_1 and P_3 hold $x_1 \in \mathbb{Z}_{2^k}$, such that $x_0 + x_1 = x \bmod 2^k$.

We denote the above sharing schemes as $[\cdot]_{\mathsf{AC}}$-Sharing and $[\cdot]_{\mathsf{AB}}$-Sharing. We further denote P_i's local share of $[x]_{\psi}$ as $[x]_{\psi}^{\mathsf{P}_i}$, where $\psi \in \{\mathsf{AC}, \mathsf{AB}\}$. We also define the *reconstruction partner* as follows:

Definition 4.2.3 (*Reconstruction Partner*)**.** Suppose that a secret value x is $\mathcal{X}$-shared in ψ mode. The reconstruction partner of P_i is the party with which P_i can reconstruct the secret value x regarding the share-mode ψ.

4.2.2. Share-Mode Conversion

In this section, we show that the parties can execute a *share-mode conversion* protocol Π_{chMo} to change share-modes. Suppose that the parties originally hold $[x]_{\mathsf{AC}}$ and are willing to convert $[x]_{\mathsf{AC}}$ to $[x]_{\mathsf{AB}}$. The local shares must be freshly computed, otherwise a party (e.g. P_1) which originally holds x_0 and then receives x_1 is able to reconstruct x.

We formally describe Π_{chMo} in Fig. 4.1. We first assume that the parties already hold a correlated randomness called *changing-mode sharing* (CMS), denoted as $[\![r]\!]_{\psi\mathrm{to}\phi}$:

Definition 4.2.4 (*Changing-Mode Sharing*)**.** A *changing-mode sharing* is defined as $[\![r]\!]_{\psi\mathrm{to}\phi} := (r_0, r_1, r) \in \mathbb{Z}_{2^k}^3$, such that

$$r_0 + r_1 + r = 0 \bmod 2^k.$$

If ψtoϕ = ACtoAB, P_0 holds r_0, P_1 holds r_1, P_2 and P_3 hold r. If ψtoϕ = ABtoAC, P_0 holds r_0, P_2 holds r_1, P_1 and P_3 hold r.

In Π_{chMo}, the generation of $[\![r]\!]_{\psi\text{to}\phi}$ is realized via an interaction between the parties and the four-party preprocessing ideal functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$, formally defined in Fig. 4.7. Suppose that ψ = AC and ϕ = AB. After executing Π_{chMo}, parties P_0 and P_2 (and similarly P_1 and P_3) obtain identical local output shares. Intuitively, the values (r_0, r_1) serve as masks for the local shares of P_0 and P_2, while the parties holding r directly adopt it as their refreshed local share under the target sharing ϕ. As shown in Fig. 4.1, P_0 and P_2 exchange their masked values d_0 and d_1, respectively, and compute their outputs as

$$[x]_{\mathsf{AB}}^{\mathsf{P}_0} = [x]_{\mathsf{AB}}^{\mathsf{P}_2} = d_0 + d_1 \bmod 2^k.$$

Meanwhile, P_1 and P_3 set their outputs to

$$[x]_{\mathsf{AB}}^{\mathsf{P}_1} = [x]_{\mathsf{AB}}^{\mathsf{P}_3} = r.$$

The correctness follows from

$$d_0 + d_1 + r = x_0 + x_1 + r_0 + r_1 + r = x \bmod 2^k,$$

which shows that the resulting shares reconstruct the original value x.

There exist multiple approaches to generate $[\![r]\!]_{\psi\text{to}\phi}$, depending on whether certain parties are allowed to learn other parties' local shares. We take $[\![r]\!]_{\mathsf{ACtoAB}}$ as an example. A naive way to generate $[\![r]\!]_{\mathsf{ACtoAB}}$ is to let P_0 hold k^0_{ACtoAB}, P_1 hold k^1_{ACtoAB} and P_2 (and P_3) hold $(k^0_{\mathsf{ACtoAB}}, k^1_{\mathsf{ACtoAB}})$, then derive randomness from keyed PRF. However, we notice that in this case (r_0, r_1) are publicly known by P_1 and P_3. Although the above approach does not compromise the security of Π_{chMo}, since P_0 and P_2 do not have to communicate with P_1 and P_3, we choose to hide the local shares (r_0, r_1) from the parties that hold r. We introduce our CMS generation protocol Π_{CMSGen} in Fig. 4.2.

4.2.3. Reshare

We also present a *reshare* protocol Π_{resh} in Fig. 4.3, which converts a 4-out-of-4 shared value $[x]_{\mathsf{4o4}}$ to an $\mathcal{X}$-shared value $[x]_\psi$, where $\psi \in \{\mathsf{AC}, \mathsf{AB}\}$. The key idea is to make use of a *Zero Sharing* similar as in ABY3 [119].

We define the *reshare partner* as follows:

Definition 4.2.5 (*Reshare Partner*)**.** Suppose that a secret value x is 4-out-of-4 shared among the parties and $[x]_\psi$ is desired. The *reshare partner* is the party with which P_i should exchange its masked local share to construct $[x]_\psi$:

- If ψ = AC, P_0 and P_1, P_2 and P_3 are reshare partners for each other, respectively.
- If ψ = AB, P_0 and P_2, P_1 and P_3 are reshare partners for each other, respectively.

Protocol Π_{CMSGen}

Public inputs: Parties hold ctr.
Outputs: $[\![r]\!]_{\psi \mathsf{to} \phi}$.
Protocol:

1. P_0 randomly choose k^0_{ACtoAB}, k^1_{ACtoAB}, k^0_{ABtoAC} and k^1_{ABtoAC}. It sends k^0_{ACtoAB} to P_1 and P_3, k^1_{ACtoAB} to P_2. It also sends k^0_{ABtoAC} to P_2 and P_3, k^1_{ABtoAC} to P_1
2. P_1 randomly choose k^2_{ACtoAB} and k^2_{ABtoAC}. It sends k^2_{ACtoAB} to P_2 and P_3, it also sends k^2_{ABtoAC} to P_2 and P_3.

- If $\psi \mathsf{to} \phi = \mathsf{ACtoAB}$:
  3. P_0 computes $r_0 = \mathsf{PRF}_{k^0_{\mathsf{ACtoAB}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^1_{\mathsf{ACtoAB}}}(\mathsf{ctr})$ then sets $[\![r]\!]^{\mathsf{P}_0}_{\mathsf{ACtoAB}} = r_0$.
  4. P_2 computes $r_1 = \mathsf{PRF}_{k^1_{\mathsf{ACtoAB}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^2_{\mathsf{ACtoAB}}}(\mathsf{ctr})$ then sets $[\![r]\!]^{\mathsf{P}_2}_{\mathsf{ACtoAB}} = r_1$.
  5. P_1 and P_3 compute:
     $r = \mathsf{PRF}_{k^2_{\mathsf{ACtoAB}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^0_{\mathsf{ACtoAB}}}(\mathsf{ctr})$ and set $[\![r]\!]^{\mathsf{P}_1}_{\mathsf{ACtoAB}} = [\![r]\!]^{\mathsf{P}_3}_{\mathsf{ACtoAB}} = r$, respectively.
- Otherwise if $\psi \mathsf{to} \phi = \mathsf{ABtoAC}$:
  3. P_0 computes $r_0 = \mathsf{PRF}_{k^0_{\mathsf{ABtoAC}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^1_{\mathsf{ABtoAC}}}(\mathsf{ctr})$ then sets $[\![r]\!]^{\mathsf{P}_0}_{\mathsf{ABtoAC}} = r_0$.
  4. P_1 computes $r_1 = \mathsf{PRF}_{k^1_{\mathsf{ABtoAC}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^2_{\mathsf{ABtoAC}}}(\mathsf{ctr})$ then sets $[\![r]\!]^{\mathsf{P}_1}_{\mathsf{ABtoAC}} = r_1$.
  5. P_2 and P_3 compute:
     $r = \mathsf{PRF}_{k^2_{\mathsf{ABtoAC}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^0_{\mathsf{ABtoAC}}}(\mathsf{ctr})$ and set $[\![r]\!]^{\mathsf{P}_2}_{\mathsf{ACtoAB}} = [\![r]\!]^{\mathsf{P}_3}_{\mathsf{ABtoAC}} = r$, respectively.

Figure 4.2.: Four-Party Changing-Mode Sharing Generation Protocol

Protocol Π_{resh}

Private inputs: Parties hold $[x]_{4o4}$.
Outputs: $[x]_\psi$ with $\psi \in \{\mathsf{AC}, \mathsf{AB}\}$.
Preprocessing: P_i sends $(\mathsf{ZeroSGen}, \mathsf{P}_i, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Pre}}_{\mathsf{4PC}}$, receives $[0]^{\mathsf{P}_i}_{4o4}$ as output.
Protocol:

1. We denote $\mathsf{P}^{\mathsf{P}_i}_j$ as P_i's reshare partner.
   - If $\psi = \mathsf{AC}$: P_0 and P_1 set $\mathsf{P}^{\mathsf{P}_0}_j = \mathsf{P}_1$ and $\mathsf{P}^{\mathsf{P}_1}_j = \mathsf{P}_0$, respectively. P_2 and P_3 set $\mathsf{P}^{\mathsf{P}_2}_j = \mathsf{P}_3$ and $\mathsf{P}^{\mathsf{P}_3}_j = \mathsf{P}_2$, respectively.
   - If $\psi = \mathsf{AB}$: P_0 and P_2 set $\mathsf{P}^{\mathsf{P}_0}_j = \mathsf{P}_2$ and $\mathsf{P}^{\mathsf{P}_2}_j = \mathsf{P}_0$, respectively. P_1 and P_3 set $\mathsf{P}^{\mathsf{P}_1}_j = \mathsf{P}_3$ and $\mathsf{P}^{\mathsf{P}_3}_j = \mathsf{P}_1$, respectively.
2. Each P_i computes $e^{\mathsf{P}_i} = [x]^{\mathsf{P}_i}_{4o4} + [0]^{\mathsf{P}_i}_{4o4} \bmod 2^k$ and sends e^{P_i} to its reshare partner P_j.
3. Upon receiving e^{P_j} from $\mathsf{P}^{\mathsf{P}_i}_j$, P_i sets $[x]^{\mathsf{P}_i}_\psi = e^{\mathsf{P}_i} + e^{\mathsf{P}_j} \bmod 2^k$.

Figure 4.3.: Four-Party Reshare Protocol

As shown in Π_{resh}, P_i only have to mask its own local share $[x]^{\mathsf{P}_i}_{4o4}$ with $[0]^{\mathsf{P}_i}_{4o4}$ and then exchange the masked share e^{P_i} with its reshare partner $\mathsf{P}^{\mathsf{P}_i}_j$ according to desired share-mode ψ. The correctness follows from

$$\sum_i e^{\mathsf{P}_i} = \sum_i (x_i + r_i) = x \bmod 2^k.$$

4.2.4. Linearity

We now discuss the linearity properties of $\mathcal{X}$-sharing in AC mode.

Let $x, y \in \mathbb{Z}_{2^k}$ be two secret values that are $\mathcal{X}$-shared in the same mode, denoted by $[x]_{\mathrm{AC}}$ and $[y]_{\mathrm{AC}}$. Then the parties can locally compute

$$[z]_{\mathrm{AC}} = [x]_{\mathrm{AC}} + [y]_{\mathrm{AC}},$$

where for each party P_i,

$$[z]_{\mathrm{AC}}^{\mathsf{P}_i} = [x]_{\mathrm{AC}}^{\mathsf{P}_i} + [y]_{\mathrm{AC}}^{\mathsf{P}_i} \bmod 2^k.$$

This yields a valid $\mathcal{X}$-sharing of $z = x+y \bmod 2^k$. Similarly, for public constants $e_0, e_1 \in \mathbb{Z}_{2^k}$, the parties can locally compute

$$[z]_{\mathrm{AC}} = e_0[x]_{\mathrm{AC}} + e_1[y]_{\mathrm{AC}},$$

by setting, for each P_i,

$$[z]_{\mathrm{AC}}^{\mathsf{P}_i} = e_0[x]_{\mathrm{AC}}^{\mathsf{P}_i} + e_1[y]_{\mathrm{AC}}^{\mathsf{P}_i} \bmod 2^k.$$

This produces a sharing of $z = e_0x + e_1y \bmod 2^k$.

However, suppose x and y are $\mathcal{X}$-shared in two different share-modes, i.e., the parties hold $[x]_{\mathrm{AC}}$ and $[y]_{\mathrm{AB}}$. In this case, a direct local addition yields a 4-out-of-4 additive sharing of

$$z = 2x + 2y \bmod 2^k.$$

More precisely, defining

$$[z]_{4\mathrm{o}4} = [x]_{\mathrm{AC}} + [y]_{\mathrm{AB}},$$

each party P_i locally computes

$$[z]_{4\mathrm{o}4}^{\mathsf{P}_i} = [x]_{\mathrm{AC}}^{\mathsf{P}_i} + [y]_{\mathrm{AB}}^{\mathsf{P}_i} \bmod 2^k.$$

Since each secret component is duplicated across two parties, the resulting reconstruction equals

$$\sum_{\mathsf{P}_i} [z]_{4\mathrm{o}4}^{\mathsf{P}_i} = 2x + 2y \bmod 2^k,$$

which corresponds to a 4-out-of-4 shared $[2x + 2y]_{4\mathrm{o}4}$.

To maintain $\mathcal{X}$-sharing for the computed value, the parties can execute a *Share-Mode Conversion* protocol Π_{chMo} (see Section 4.2.2), converting the share-mode of y (or x) from $[\cdot]_{\mathrm{AB}}$-sharing to $[\cdot]_{\mathrm{AC}}$-sharing. After conversion, both x and y are again $\mathcal{X}$-shared in the same mode, and local linear operations preserve the intended invariant.

Protocol Π_{Mult}

Private inputs: Parties hold $[x]_\psi$, $[y]_\phi$, where $\psi \neq \phi$.
Outputs: $[z]_\theta$ with $z = xy$.
Protocol:

1. Parties locally compute $[z]_{4o4}$, where $[z]_{4o4}^{\mathsf{P}_i} = [x]_\psi^{\mathsf{P}_i}[y]_\phi^{\mathsf{P}_i} \bmod 2^k$.
2. Parties execute Π_{resh} with $[z]_{4o4}$ as input, receives $[z]_\theta$ as output.

Figure 4.4.: Four-Party Multiplication Protocol with $\psi \neq \phi$

4.2.5. Multiplication with $\mathcal{X}$-Sharing

We now describe how to perform four-party multiplication with $\mathcal{X}$-sharing. Suppose the parties hold $[x]_{\mathsf{AC}}$ and $[y]_{\mathsf{AB}}$. The goal is to compute a 4-out-of-4 shared product $[z]_{4o4}$, where $z = x \cdot y \bmod 2^k$. Each party P_i can then locally compute

$$[z]_{4o4}^{\mathsf{P}_i} = [x]_{\mathsf{AC}}^{\mathsf{P}_i}[y]_{\mathsf{AB}}^{\mathsf{P}_i} \bmod 2^k.$$

The Correctness follows from

$$\begin{aligned} xy &= (x_0 + x_1)(y_0 + y_1) \\ &= x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1 \bmod 2^k. \end{aligned}$$

Hence, no communication is required to compute $[z]_{4o4}$. However, the resulting sharing is a 4-out-of-4 additive sharing and does not preserve the $\mathcal{X}$-sharing structure. If subsequent computation requires z to be $\mathcal{X}$-shared (or for efficiency considerations), the parties must execute the reshare protocol Π_{resh} to convert $[z]_{4o4}$ into a valid $\mathcal{X}$-sharing $[z]_\theta$. The full multiplication protocol Π_{Mult}, which invokes Π_{resh} as a subprotocol, is described in Fig. 4.4.

Sometimes, the parties are forced to compute the product of $[x]_\psi$ and $[y]_\phi$, where $\psi = \phi$. As a result, they are no longer able to apply the efficient multiplication as explained above. Instead, they must use Beaver triple generated in the preprocessing stage to support the multiplication in a 2PC way. Another option is that the parties first execute the protocol Π_{chMo} to adjust the share-mode of one secret value, and then apply the multiplication in the 4PC way. An obvious advantage of such a protocol construction is that parties can get rid of Beaver triple generation, but this requires an additional communication round in the online stage as trade-off. We formally construct such a protocol $\Pi_{\mathsf{Mult}}^{\mathsf{chMo}}$ in Fig. 4.5, where parties call $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMo}}$ described in Fig. A.16 as a subroutine protocol.

4.2.6. Truncation with $\mathcal{X}$-Sharing

Since we are using fixed-point numbers to represent both x and y, the parties must additionally truncate the multiplicative result $[z]_\theta$ to maintain p decimal bit precision.

Protocol $\Pi^{\mathsf{chMo}}_{\mathsf{Mult}}$

Private inputs: Parties hold $[x]_\psi$, $[y]_\psi$.
Outputs: $[z]_\theta$ with $z = xy$.
Protocol:

1. Parties send $(\mathsf{chMode}, [y]^{\mathsf{P}_c}_\psi, \mathsf{P}_c, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{chMo}}_{\mathsf{4PC}}$, receive $[y]_\phi$ as output, where $\phi \neq \psi$.

1. Parties locally compute $[z]_{\mathsf{4o4}}$, where $[z]^{\mathsf{P}_i}_{\mathsf{4o4}} = [x]^{\mathsf{P}_i}_\psi [y]^{\mathsf{P}_i}_\phi \bmod 2^k$.

2. Parties execute Π_{resh} with $[z]_{\mathsf{4o4}}$ as input, receives $[z]_\theta$ as output.

Figure 4.5.: Four-Party Multiplication Protocol with $\psi = \phi$

Note that $\mathcal{X}$-Sharing actually splits a secret value into two shares, resulting in a 2-out-of-2 sharing. This construction is compatible with the free truncation technique Π_{trunc} introduced by SecureML [120], which avoids additional communication overhead and extra round required by the truncation protocols Π_{trunc1} and Π_{trunc2} proposed by ABY3 [119].

4.2.7. Duality

Due to the specific property of $\mathcal{X}$-Sharing explained in Section 4.2.5, a preferred scenario is when the parties compute the product of two $\mathcal{X}$-shared values that have different share-modes. However, sometimes the parties must multiply two $\mathcal{X}$-shared values in the same share-mode. A common example is to compute the square of a secret value. The simplest approach is to always execute Π_{chMo} when necessary. A more efficient solution is for the parties to hold one shared value in both share-modes, which enables them to perform efficient 4PC computations everywhere (using $\mathcal{X}$-Sharing). Note that holding an $\mathcal{X}$-shared value in both share-modes does not leak any information to the parties, since the local shares of each $\mathcal{X}$-shared value are freshly chosen, such as

$$x = x_0 + x_1 \bmod 2^k \quad \text{and} \quad x = x'_0 + x'_1 \bmod 2^k.$$

4.3. Efficiency Analysis

In an MPC-based privacy-preserving machine learning framework, convolution operations implemented as dot products between secret-shared vectors are highly throughput-intensive and often constitute the most time-consuming component of the training phase [40]. In this section, we compare $\mathcal{X}$-Sharing with existing secret sharing schemes regarding the computation and communication overhead for the dot product (DotP) computation. A summary of our result is shown in Table 4.1. Force is implemented on top of the GPU framework Piranha [169], which already includes several state-of-the-art secret sharing schemes for 2PC, 3PC and 4PC. We denote those as P-Framework in Table 4.1.

Table 4.1.: Force Compared to the Existing Works Regarding Dot Product (in bits).

Setting	Framework	Preparation	Online		Local	with Trunc	
		Comm	Comm	Rounds	Mult	Comm	Rounds
2PC (S.H.)[1]	P-SecureML [120, 169]	TTP[2]	$4nk$	1	3	$4nk$	1
3PC (S.H.)	CryptGPU [156]	0	$2k$	1	3	$3k$	2
	P-Falcon [164, 169]	0	$2k$	1	3	$4k$	1
4PC (S.H.)	CrypTen [88]	TTP	$8nk$	1	2	$(8n+4)k$	2
	P-FantasticFour [45, 169]	0	$4k$	1	7	$6k$	2
	PrivPy [101]	0	$4k$	1	2	$4k$	1
4PC (M.)[3]	Trident [32]	$3k$	$4k$	1	3	$5k(4k)$	$2(1)$
	Swift [89]	$3k$	$3k$	2	3	$4k(3k)$	$2(1)$
	Tetrad [90]	$2k$	$4k(3k)$	$2(1)$	4	$4k(3k)$	$2(1)$
4PC (S.H.)	**Force**	**0**	**2k**	**1**	**1**	**2k**	**1**

In 2PC, P-SecureML [120, 169] requires Beaver triples to support DotP in the online stage. As a shortcut, P-SecureML assumes a trusted third party to generate and distribute the secret shares, instead of implementing the full share-generation protocol. Since local truncation is free for 2PC [120], the total communication overhead for the online stage comes solely from the multiplication, which costs $4nk$ bits.

In 3PC, both CryptGPU [156] and P-Falcon [164, 169] choose to apply the replicated sharing scheme, which is more communication-friendly in the honest-majority setting. While the multiplication in this case requires no communication, parties must exchange overall $2k$ bits to reconstruct the replicated share holdings. ABY3 [119] has shown that the local truncation technique [120] fails for the replicated sharing scheme in 3PC case. Thus, P-Falcon chooses to perform a three-party truncation protocol Π_{trunc2} [119] with the help of a pre-computed truncation pair $([r], [r'])$, where $r' = r/2^p$. Combined with resharing, the above protocol requires $4k$ bits communication in a single communication round. To eliminate the requirement for a truncation pair, CryptGPU [156] implements an alternative three-party truncation protocol Π_{Trunc1} [119], which incurs a total communication volume of $3k$ across two rounds totally.

In 4PC, existing frameworks use different sharing schemes to perform computations. CrypTen [88] chooses to apply a 4-out-of-4 sharing scheme. Regardless of the triple generation, each party still has to send/receive $8nk$ bits for the DotP protocol and $4k$ bits for the truncation protocol. Compared to CrypTen, both PrivPy [101] and P-FantasticFour [45, 169] use replicated sharing scheme over four shares, which reduces the communication overhead to $4k$ and $6k$, respectively. However, we notice that there is still potential for optimization in local computations. As shown in Table 4.1, P-FantasticFour requires each party to perform DotP seven times, while PrivPy requires two times.

Recently, multiple four-party computation (4PC) frameworks [32, 89, 90] have realized malicious security under an honest-majority assumption. By exploiting correlated ran-

[1] Semi-Honest
[2] Trusted Third-Party
[3] Malicious

domness generated in the preprocessing phase, these protocols significantly reduce the cost of online computation. Furthermore, the availability of batched multiplication gates enables amortized communication overhead, as reflected in Table 4.1. In contrast to semi-honest 4PC frameworks [88, 101], these designs achieve stronger security guarantees while maintaining comparable online communication complexity.

By proposing $\mathcal{X}$-Sharing, we introduce Force, which achieves significant reductions in both computational and communication complexity. First, note that our preprocessing stage requires no communication, as the parties can independently generate *Zero Sharing* by deriving randomness with a PRF. Moreover, Force is the only framework which applies the free two-party truncation protocol Π_{trunc} [120] as a non-2PC framework. For the online computation of DotP combined with truncation, the parties only have to exchange overall $2k$ bits in a single round. In addition, $\mathcal{X}$-Sharing reduces the local computation of DotP to a single multiplication per party, improving local computational efficiency as well. This is especially beneficial for large-scale PPML tasks when parties have limited local computational resources, such as few GPU cards.

4.4. Share Conversion with $\mathcal{X}$-Share

4.4.1. $\mathcal{X}$-dabit

For PPML tasks, while linear functions (such as multiplication, convolution etc.) can be efficiently computed using arithmetic share, non-linear functions (such as ReLU, max-pooling etc.) are typically more efficient when evaluated over boolean inputs [88, 119, 137, 156, 169]. Consequently, the parties must frequently convert between arithmetic and Boolean share representations. In this section, we introduce how the parties perform such share conversion using $\mathcal{X}$-Sharing.

As an important building block, we extend the edabit primitive introduced by Escudero *et al.* [54] to a new construction called $\mathcal{X}$-dabit. We now formally define $\mathcal{X}$-dabit as follows:

Definition 4.4.1 ($\mathcal{X}$-dabit)**.** An $\mathcal{X}$-dabit is defined as

$$\mathcal{X}\text{-dabit} := \big([b]_\psi,\ \langle \mathbf{r}\rangle_\phi = (\langle r_0\rangle_\phi, \langle r_1\rangle_\phi, \ldots, \langle r_{m-1}\rangle_\phi)\big),$$

where $r_j \in \mathbb{Z}_2$ and $\langle r_j\rangle_\phi^{\mathsf{P}_i} \in \mathbb{Z}_2$ for all $j \in \{0, \ldots, m-1\}$. The value $b \in \mathbb{Z}_{2^k}$ satisfies

$$b = \sum_{j=0}^{m-1} 2^j r_j \bmod 2^k,$$

and each party holds $[b]_\psi^{\mathsf{P}_i} \in \mathbb{Z}_{2^k}$.

Protocol Π_{BitToA}

Private inputs: Parties hold $\langle x \rangle_\phi$, where $x \in \mathbb{Z}_2$.
Outputs: $[x]_\psi$.
Preprocessing: P_i sends ($\mathcal{X}$-dabitGen, $\psi, \phi, m, \mathsf{P}_i$, sid) to $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$ with $m = 1$, receives $([b]_\psi^{\mathsf{P}_i}, \langle r_0 \rangle_\phi^{\mathsf{P}_i})$ as output.
Protocol:

1. Parties locally compute and then reveal $h = \langle x \rangle_\phi \oplus \langle r_0 \rangle_\phi$, where $h \in \mathbb{Z}_2$.
2. If $h = 0$, parties set $[x]_\psi = [b]_\psi$, otherwise $[x]_\psi = 1 - [b]_\psi$.

Figure 4.6.: Four-Party Bit-to-Arithmetic Protocol

Note that the share-mode ψ does not need to be identical with ϕ. We begin by generating an $\mathcal{X}$-dabit, where $\psi = \phi$. Following the protocols proposed by [54] in 2PC setting, each pair of parties can generate an identical (shared) edabit using the same derived randomness. Subsequently, the parties can execute the protocol Π_{chMo} to change the share-mode of either $[b]_\psi$ or $(\langle r_0 \rangle_\phi, \langle r_1 \rangle_\phi, ..., \langle r_{m-1} \rangle_\phi)$.

4.4.2. Bit to Arithmetic

We first consider the one bit case, where the goal is to convert a Boolean share $\langle x \rangle_\phi$ into an arithmetic share $[x]_\psi$, for $x \in \mathbb{Z}_2$. This conversion is handled by a Bit2A protocol Π_{BitToA} described in Fig. 4.6. Since $\langle x \rangle_\phi$ is a Boolean-shared bit, the parties generate only a simple $\mathcal{X}$-dabit $([b]_\psi, \langle r_0 \rangle_\phi)$ with $m = 1$. During the online stage, the parties simply reconstruct the masked result $h = \langle x \rangle_\phi \oplus \langle r_0 \rangle_\phi$, then locally "unmask" the revealed value h using the arithmetic share $[b]_\psi$. Such a bit-to-arithmetic conversion is necessary, when a precomputed protocol outputs a Boolean share but the subsequent computation requires an arithmetic share. A common use case for the protocol Π_{BitToA} arises when the parties have to jointly evaluate a non-linear function. We take the ReLU function as an example. To determine the sign of a secretly shared value x, the parties first execute a comparison protocol Π_{Comp} (will be explained in Section 4.4.3), which outputs a Boolean-shared result $\langle z \rangle_\phi$ indicating whether $x \geq 0$. At this point, the parties hold an arithmetic share $[x]_\phi$ together with a Boolean share $\langle z \rangle_\phi$. To evaluate the ReLU function, they have to compute the product of these two shared values, which requires converting the Boolean share $\langle z \rangle_\phi$ into an arithmetic share $[z]_\psi$ using Π_{BitToA}. Once the conversion is complete, the multiplication can be performed in the arithmetic domain.

4.4.3. Secure Comparison via Bit Extraction

In this section, we introduce how the comparison protocol Π_{Comp} is executed via extracting the most significant bit (MSB) using $\mathcal{X}$-Sharing.

Suppose that the parties have to compare an arithmetically shared value $[x]$ against $[y]$ ($[x]$ against 0 for ReLU). They first compute the difference $[z] = [x] - [y]$ and then extract the MSB of $[z]$ by jointly evaluating a *Prefix Adder Circuit* (PPA) [88, 119, 137, 156, 169]. Following the implemented protocol in Piranha [169], the parties generate edabits $([b], \langle \mathbf{r} \rangle)$ in the preprocessing stage. In the online stage, the parties use the arithmetic part of the generated edabits to mask and reconstruct $x' = [x] - [b]$. Then they use the bit decomposition of x' (denoted as $\mathbf{x}'$) to compute the shared propagator $\langle \mathbf{p} \rangle = \langle \mathbf{r} \rangle \oplus \mathbf{x}'$ and the shared generator $\langle \mathbf{g} \rangle = \langle \mathbf{r} \rangle * \mathbf{x}'$, where $*$ denotes bit-wise **AND** of two vectors. Now holding both $\langle \mathbf{p} \rangle$ and $\langle \mathbf{g} \rangle$ allows parties to jointly evaluate PPA.

We note that the above computation is non-trivial using $\mathcal{X}$-Sharing, since now we are able to manage the share-modes of $\mathbf{p}$ and $\mathbf{g}$ during the iterative computations on PPA to achieve better efficiency. For each iteration, the parties are required to compute three monomials: $\langle \mathbf{p} \rangle * \langle \mathbf{g} \rangle$, $\langle \mathbf{p} \rangle * \langle \mathbf{p} \rangle$ and $\langle \mathbf{g} \rangle \oplus \langle \mathbf{g} \rangle$. The ideal situation is when all AND operations can be executed using the efficient 4PC protocol as described in Section 4.2.5. However, since $\langle \mathbf{p} \rangle$ must be updated at each iteration, holding the pair $(\langle \mathbf{p} \rangle_\psi, \langle \mathbf{p} \rangle_\phi)$ with $\psi \neq \phi$ does not bring any help [4]. Thus, we simply let the parties hold $\langle \mathbf{p} \rangle_\psi$ and $\langle \mathbf{g} \rangle_\phi$ such that $\psi \neq \phi$, allowing approximately 50% of the AND operations to be executed efficiently using the 4PC protocol.

4.5. Security Proof

4.5.1. Four-party Preprocessing Functionality

We formally define the four-party preprocessing functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$ in Fig. 4.7.

4.5.2. Security of Π_{CMSGen}

Theorem 4.5.1. If PRF is a pseudorandom function, then Protocol Π_{CMSGen} described in Fig. 4.2 uc-realizes the functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{CMSGen}}$ described in Fig. A.14 and Fig. A.15 in the presence of a semi-honest adversary who can corrupt $\mathsf{P}_i \in \{\mathsf{P}_0, \mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3\}$, with static corruption.

We defer the detailed proof to Appendix A.3.1.

[4] Updating both $\langle \mathbf{p} \rangle_\psi$ and $\langle \mathbf{p} \rangle_\phi$ will results in additional communication and computation overhead, which is equivalent to execute the protocol Π_{chMo}.

Functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$

The functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$ has all the same features as $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{CMSGen}}$, with additional command:
Zero Sharing Generation:

- Wait to receive k_{Zero} and k'_{Zero} from the adversary.
- Upon receiving (**ZeroSGen**, ctr, P_i, sid) from $\mathsf{P}_i \in \mathcal{P}$:
  - Compute $[0]_{\mathsf{Zero}}^{\mathsf{P}_c} = \mathsf{PRF}_{k_{\mathsf{Zero}}}(\mathsf{ctr}) - \mathsf{PRF}_{k'_{\mathsf{Zero}}}(\mathsf{ctr})$.
  - Sample $[0]_{\mathsf{Zero}}^{\mathsf{P}_j}$ for all $\mathsf{P}_j \in \mathcal{P}$ and $j \neq c$, such that $\sum_j [0]_{\mathsf{Zero}}^{\mathsf{P}_j} \equiv 0 \pmod{2^k}$.
  - Send $[0]_{\mathsf{Zero}}^{\mathsf{P}_i}$ to P_i.

$\mathcal{X}$-dabit **Generation:**

- Upon receiving ($\mathcal{X}$-dabitGen, $\psi, \phi, m, \mathsf{P}_i$, sid) from $\mathsf{P}_i \in \mathcal{P}$,
  - Wait to receive $([b]_{\psi}^{\mathsf{P}_c}, \langle \mathbf{r} \rangle_{\phi}^{\mathsf{P}_c})$ from the adversary, where $[b]_{\psi}^{\mathsf{P}_c} \in \mathbb{Z}_{2^k}$ and $\langle \mathbf{r} \rangle_{\phi}^{\mathsf{P}_c} \in \mathbb{Z}_2^m$.
  - Let $[b]_{\psi}^{\mathsf{P}_c} = b^c$, $\langle \mathbf{r} \rangle_{\phi}^{\mathsf{P}_c} = (r_0^c, ..., r_{m-1}^c)$, sample $b^{c-1} \xleftarrow{\$} \mathbb{Z}_{2^k}$ and $(r_0^{c-1}, ..., r_{m-1}^{c-1}) \xleftarrow{\$} \mathbb{Z}_2^m$, such that $b^c + b^{c-1} = \sum_{j=0}^{m-1} 2^j (r_j^c + r_j^{c-1}) \bmod 2^k$.
  - Assign (b^c, b^{c-1}) and $((r_0^c, ..., r_{m-1}^c), (r_0^{c-1}, ..., r_{m-1}^{c-1}))$ appropriately to each party's output regarding ψ and ϕ. Send $([b]_{\psi}^{\mathsf{P}_i}, \langle \mathbf{r} \rangle_{\phi}^{\mathsf{P}_i})$ to P_i.

Figure 4.7.: Four-Party Preprocessing Functionality

4.5.3. Security of Π_{chMo}

Theorem 4.5.2. The protocol Π_{chMo} described in Fig. 4.1 uc-realizes the functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMo}}$ described in Fig. A.16 in the $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$-hybrid model, in the presence of a semi-honest adversary who can corrupt $\mathsf{P}_i \in \{\mathsf{P}_0, \mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3\}$, with static corruption.

We defer the detailed proof to Appendix A.3.2.

4.5.4. Security of Π_{Mult}

Theorem 4.5.3. The protocol Π_{Mult} described in Fig. 4.1 uc-realizes the functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{chMo}}$ described in Fig. A.16 in the $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{Pre}}$-hybrid model, in the presence of a semi-honest adversary who can corrupt $\mathsf{P}_i \in \{\mathsf{P}_0, \mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3\}$, with static corruption.

We defer the detailed proof to Appendix A.3.3.

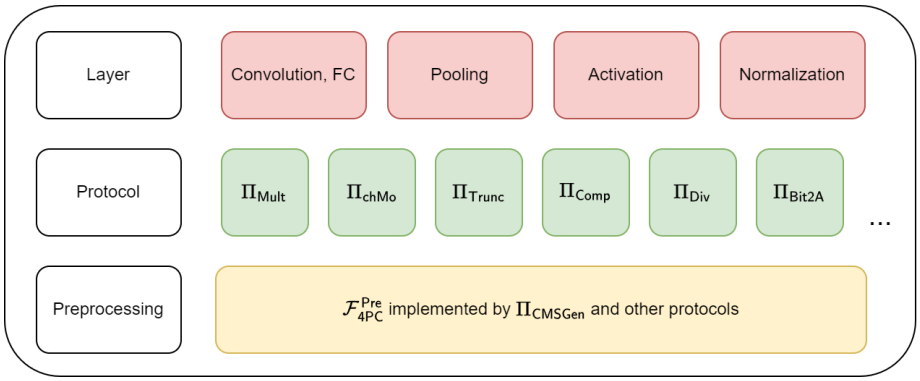


Figure 4.8.: Force Architecture

4.6. Force Architecture

We construct efficient 4PC protocols as the core building blocks of Force. The overall architecture is illustrated in Fig. 4.8. As introduced in Section 4.1, we begin at the top level with the fundamental computational layers of a convolutional neural network (CNN). To support privacy-preserving evaluation of these layers in the 4PC setting, we design a set of modular protocols as placed in the middle layer of Fig. 4.8. The bottom layer of the architecture represents the preprocessing stage, where parties invoke the abstracted functionality $\mathcal{F}_{\text{4PC}}^{\text{Pre}}$ (described in Fig 4.7) to generate the necessary correlated randomness for the online phase.

4.7. Evaluation

We conduct in-depth experiments to compare our proposed solution against state-of-the-art privacy-preserving machine learning (PPML) frameworks. We build Force on top of Piranha [169] [5], at commit *bd9c8c4*, and we implement the framework in `C++`. In particular, we implement all modular protocols using our novel $\mathcal{X}$-Sharing scheme.

4.7.1. Evaluation Setup

Testbed Environment. We run our evaluations on 4 cloud servers. Our server is running `Ubuntu 18.04.6 LTS` with `CUDA 10.1.243`. The servers come with 2 CPUs, `Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz`, and 12 × 128GB of RAM. Each of our servers is equipped with one GPU, `NVIDIA Tesla P100- PCIE` with 16GB of video RAM (VRAM). We consider two types of network environments, both simulated by the `tc` tool [6]:

- **LAN**: 10Gbps bandwidth + 0.2*ms* round-trip latency;

[5] `https://github.com/ucbrise/piranha/`
[6] `https://man7.org/linux/man-pages/man8/tc.8.html`

- **WAN**: 100Mbps bandwidth + 40ms round-trip latency.

Baseline. We select several state-of-the-art systems with **semi-honest security** as baselines, as summarized in Table 4.1. For 2PC, Cheetah [73] is the most recent PPML work using homomorphic encryption (HE) and correlated oblivious transfer (cOT) on CPUs, which is fundamentally different from Force. We run it on the same server as a baseline of CPU-based PPML. SecureML [120] is the only two-party system supporting both private inference and training, which is improved by Piranha [169] via porting it to GPU. We refer to its GPU version as P-SecureML. For both 3PC and 4PC systems, we only consider the **honest-majority** setting. Falcon [164] is the fastest three-party system on CPU. Piranha [169] ports the **semi-honest** version of Falcon to GPU with a huge boost. We mark it as P-Falcon. CryptGPU [156] is another three-party system on GPU, similar to P-Falcon. We deploy it using the latest `Github` source code [7], at commit $2ff57b2$. We include both systems as baselines. As for four-party systems, CrypTen [88] is the only one with **semi-honest security** in an **honest-majority** setting by design. We deploy it using their latest `Github` source code [8], at commit $efe8eda$. Yet, Piranha [169] reimplements the **semi-honest** version of FantasticFour [45] on GPU. We include this simplified version in our evaluation and refer to it as P-FantasticFour.

We run all the evaluations using 20 bits fixed-point precision. Computations are performed over the 64-bit ring $\mathbb{Z}_{2^{64}}$, except Cheetah [73], which supports a maximum of 44-bit. All the experiments are repeated multiple times with a batch size of BatchSize = 1, as some systems do not support large batch sizes. Then we calculate the benchmark result by averaging all the results except the first run, to mitigate the influence of system initialization and runtime randomness.

Models and Datasets. We consider three datasets in different sizes for our evaluations:

- Small dataset: `CIFAR10` [91] provides 60,000 pieces 32×32 RGB images in 10 classes.
- Medium dataset: `TinyImageNet` [95] (`Tiny` for short) provides 100,000 pieces 64×64 RGB images in 200 classes.
- Large dataset: `ImageNet` [144] provides 1,000,000 pieces 224×224 RGB images in 1,000 classes.

These datasets are evaluated in three neural networks models of different depths:

- Shallow model: `AlexNet` [92] is an 8-layer convolutional network.
- Medium model: `VGG16` [152] is an 16-layer convolutional network.
- Deep model: `ResNet152` [67] is a 152-layer convolutional network.

[7] `https://github.com/jeffreysijuntan/CryptGPU`
[8] `https://github.com/facebookresearch/CrypTen`

Table 4.2.: Summary of the evaluated neural network models across different datasets. The architectures used for the small and medium datasets may vary slightly, as detailed in Section 4.7.1.

Model	Number of layers						Number of Parameters		
	Conv	ReLU	FC	Pool	BN	Total	CIFAR10	Tiny	ImageNet
AlexNet	5	7	3	3	0	18	3.9M	6.1M	35.9M
VGG16	13	15	3	6	0	37	14.9M	15.3M	54.5M
ResNet152	155	151	1	2	155	464	58.1M	58.6M	60.2M

Table 4.3.: Inference Accuracy Comparison Between Force and PyTorch

Inference		CIFAR10	Tiny	ImageNet
AlexNet	PyTorch	69.65%	26.38%	22.84%
	Force	69.69%	26.39%	22.84%
VGG16	PyTorch	88.31%	54.90%	56.41%
	Force	88.34%	54.89%	56.42%
ResNet152	PyTorch	83.99%	65.14%	67.36%
	Force	83.98%	65.15%	67.36%

We strive to preserve the original model architectures as closely as possible to their respective publications. However, due to variations in input sizes across different datasets and performance considerations, we make slight structural adjustments similar to those in CryptGPU [156] and Falcon [164]. The numbers of layers and parameters are summarized in Table 4.2.

4.7.2. Accuracy Verification

To measure accuracy, we run both inference and training with Force. We first train the models on all the datasets with PyTorch to obtain pre-trained models. Starting from those pre-trained models, we perform accuracy evaluation with Force. We run all evaluations with 26-bit fixed-point precision, as recommanded by Piranha. For inference, we use the entire validation set of CIFAR10 and randomly selected subsets of the validation sets of Tiny and ImageNet, ensuring that each actual inference dataset contains 10,000 images. The result is shown in Table 4.3. Force provides almost the same accuracy as the plaintext PyTorch, with a negligible relative error below 0.1% across all models and datasets. For

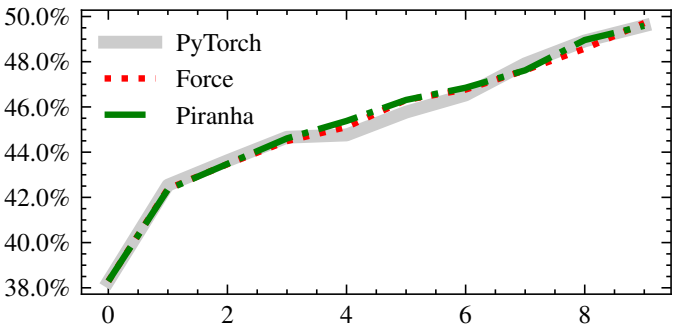


Figure 4.9.: Validation Accuracy over 9 Training Epochs for AlexNet on CIFAR10

Table 4.4.: Running Time (Seconds) of a Single Inference Pass in **LAN** (BatchSize = 1)

	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	0.41	1.48	7.89	0.55	2.19	9.44	2.50	15.70	31.46
CryptGPU	1.15	2.91	35.58	1.14	3.83	38.21	2.42	12.74	49.54
P-Falcon	0.29	0.89	5.18	0.35	1.37	6.24	1.12	10.03	20.39
CrypTen	1.05	3.48	26.04	1.25	5.20	29.10	4.59	32.75	62.58
P-FantasticFour	0.72	2.20	12.81	0.87	3.40	15.59	2.72	24.03	49.74
Force	**0.12**	**0.35**	**2.54**	**0.14**	**0.54**	**3.01**	**0.43**	**3.26**	**9.70**
Cheetah	2.67	80.43	66.96	19.74	325.30	263.87	383.97	4026.87	3226.62
PyTorch	0.0008	0.0017	0.0264	0.0009	0.0017	0.0266	0.0009	0.0017	0.0268

Table 4.5.: Running Time (Seconds) of a Single Inference Pass in **WAN** (BatchSize = 1)

	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	12.20	57.54	239.81	21.64	121.05	241.73	179.19	1126.83	1907.65
CryptGPU	18.41	44.17	807.32	19.46	65.15	846.11	48.53	359.46	1387.28
P-Falcon	2.85	11.08	91.06	3.80	28.27	119.33	30.79	370.70	730.97
CrypTen	34.37	103.26	721.67	43.47	256.77	876.92	397.49	2203.29	4649.98
P-FantasticFour	7.60	41.39	218.33	13.00	125.80	368.82	135.93	1489.91	2853.29
Force	**2.60**	**6.75**	**75.28**	**2.94**	**14.21**	**85.85**	**13.59**	**155.15**	**324.17**
Cheetah	12.64	233.34	220.16	52.59	908.09	711.77	827.68	11012.47	8101.88

training, we use the entire training set of `CIFAR10` and the full validation set for validation. Starting from a pre-trained model, we train `AlexNet` on `CIFAR10` using Piranha, Force and `PyTorch` for 9 epochs. We plot the validation accuracy curves in Fig. 4.9. After 9 epochs, `PyTorch` achieves 49.59% accuracy, while Force obtains 49.71%, which is even 0.12% higher. We also plot the validation accuracy curve of Piranha in Fig. 4.9, which indicates that Force has no accuracy loss compared to the original Piranha implementation. Note that the three Piranha-based systems (P-SecureML, P-FantasticFour AND P-FantasticFour) exhibit identical accuracy performance, so we only plot a single curve for them.

4.7.3. End-to-End Inference Running Time

We show the inference running time required for all datasets and models described in Section 4.7.1 in Table 4.4 and Table 4.5.

In the **LAN** setting, CPU-based Cheetah is slower than all the other GPU-based systems. We notice that among GPU-based systems, the `C++`-implemented frameworks perform much better than `Python`-implemented frameworks. This performance gap is likely due to inherent differences in language efficiency. We also point out that the four-party system P-FantasticFour is outperformed by the three-party system P-Falcon across all test cases. Upon analyzing the communication and computational overhead detailed in Table 4.1, we realize that simply applying the replicated sharing scheme to the 4PC framework is suboptimal. Among those `C++`-implemented, Force outperforms the other three Piranha-based systems (P-SecureML, P-Falcon and P-FantasticFour) with the acceleration brought by our novel $\mathcal{X}$-Sharing.

Table 4.6.: Communication Volume (MB) of a Single Inference Pass (BatchSize = 1)

	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	65.93	381.39	1178.82	130.16	849.01	2082.17	1186.00	8361.98	15718.33
CryptGPU	2.32	53.59	236.17	13.32	214.12	677.61	226.08	2622.02	7376.14
P-Falcon	3.72	84.48	168.85	20.83	337.62	680.50	350.09	4134.47	8441.19
CrypTen	74.67	579.78	1409.07	178.98	1641.77	3034.04	2005.10	18069.92	27607.43
P-FantasticFour	7.01	159.50	300.42	39.24	637.43	1218.99	659.45	7805.96	15150.84
Force	**1.49**	**33.76**	**79.95**	**8.38**	**134.93**	**316.65**	**140.95**	**1652.33**	**3907.41**
Cheetah	40.10	951.35	773.51	249.24	3792.40	3091.30	4493.92	46450.00	37876.50

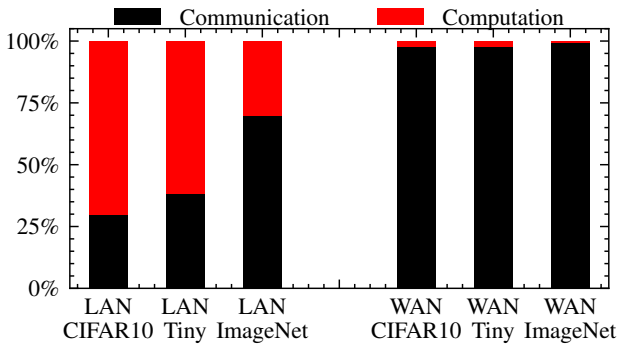
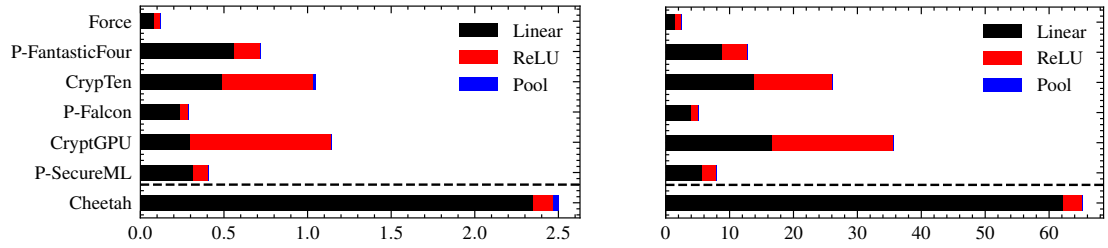


Figure 4.10.: Ratio of Communication Time to Computation Time for Force Inference on ResNet152 (BatchSize = 1)

In the **WAN** setting, Cheetah (implemented in `C++`) outperforms the `Python`-implemented systems (CryptGPU and CrypTen) on deep networks such as `ResNet152`, although it still runs slower than the other `C++`-based systems (P-SecureML, P-Falcon, P-FantasticFour, and Force). We will elaborate this fact more in depth in Section 4.7.5. Overall, the performance rankings of the frameworks (excluding Cheetah) remain consistent with the results observed in the **LAN** setting.

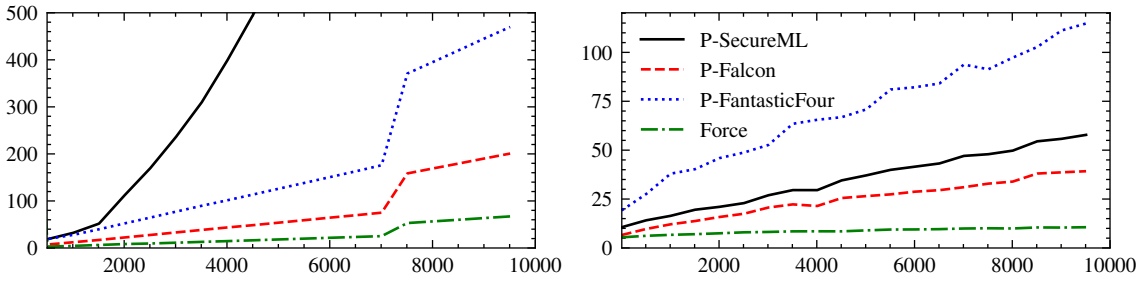
4.7.4. Inference Communication Cost

Cheetah claims that one of its main contributions is reducing the communication volume and thereby decreasing the overall communication time. Force does much better than that. More precisely, we have the minimal communication volume and the lowest communication time across all datasets and models when performing inference with BatchSize = 1. We show the actual communication volume in Table 4.6. However, we notice that the communication cost remains high, particularly for large datasets and in the **WAN** setting. To illustrate this, we plot the ratio of communication time to computation time for Force in Fig. 4.10, using several representative examples. We observe that the communication ratio increases with dataset size, which correlates with the growth in model parameter size. In high-latency environments such as the **WAN** setting, communication becomes the dominant contributor to overall runtime.



(a) `CIFAR10` + `AlexNet` **(b)** `CIFAR10` + `ResNet152`

Figure 4.11.: Running time of different operations during a single inference pass in a LAN setting (BatchSize = 1). The x-axis represents time in seconds.



(a) Matrix Multiplication **(b)** ReLU

Figure 4.12.: Micro-benchmark of matmul and ReLU in four Piranha-based systems. The x-axis represents the data dimension, and the y-axis represents time (milliseconds). For matmul, we multiply an $x \times x$ matrix by an $x \times 1$ vector.

4.7.5. Linear vs. Non-Linear Operations

We further investigate how effectively $\mathcal{X}$-Sharing accelerates common computational tasks. Our focus is on both linear operations (e.g., convolution and fully connected layers) and non-linear operations (e.g., ReLU). Throughout this section, we refer to convolution, matrix multiplication, and batch normalization as "linear operation". Fig. 4.11 shows the runtime of different operations during a single inference pass in the **LAN** setting with BatchSize = 1. Due to the significant runtime gap between Cheetah and all other systems, the results for datasets other than `CIFAR10+AlexNet` and `CIFAR10+ResNet152` are barely visible in the bar chart. Therefore, we present only these two representative cases. In addition, we conduct micro-benchmarks of matrix multiplication and ReLU across four Piranha-based systems. We evaluate the performance of matmul and ReLU on inputs of varying sizes, recording the average execution time in Fig. 4.12.

As shown in Fig. 4.11, Cheetah is extremely slow in linear operation, but performs well in non-linear operation compared to CryptGPU and CrypTen. This is primarily due to the underlying cryptographic mechanisms and the implementation language. Cheetah applies HE and cOT instead of secret sharing, resulting in significantly higher computational overhead in exchange for fewer communication rounds. Unlike CPUs, which are designed for general-purpose computation, GPUs are specialized for data-intensive tasks. It is not

Table 4.7.: Running Time (Seconds) of a Single Training Pass in **LAN** (BatchSize = 1)

	CIFAR10			Tiny			ImageNet		
	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152	AlexNet	VGG16	ResNet152
P-SecureML	1.62	4.55	29.20	7.53	5.99	27.81	7.41	28.82	65.51
CryptGPU	2.27	5.49	40.24	3.23	8.06	41.37	9.10	38.86	53.28
P-Falcon	0.75	2.44	12.08	0.96	3.04	13.55	4.13	16.14	35.78
CrypTen	13.48	40.86	27.68	18.39	50.34	33.35	FAIL	FAIL	74.07
P-FantasticFour	1.65	4.99	25.65	2.17	6.64	29.96	9.69	37.10	79.78
Force	**0.35**	**1.23**	**6.40**	**0.51**	**1.59**	**7.53**	**2.89**	**8.57**	**22.77**
PyTorch	0.0031	0.0067	0.0659	0.0027	0.0049	0.0637	0.0034	0.0077	0.0683

surprising that the CPU-FHE-based Cheetah is slower than all other GPU-MPC-based systems in linear operations. Among four Piranha-based systems accelerated by GPU and written in C++, Force completely outperforms the other three by applying the $\mathcal{X}$-Sharing scheme. According to Fig. 4.12a, both P-Falcon and Force scale linearly as the matrix size increases. However, Force is approximately 3× faster than P-Falcon, which matches our analysis in Table 4.1.

4.7.6. End-to-End Training Time

Under the same experimental setting as in Section 4.7.1, we report Force 's total running time for a training pass in the **LAN** setting across all datasets and models in Table 4.7. Since Cheetah does not support private training, it is omitted here. Force completely outperforms all the baseline systems in every evaluation. As expected, the Python-implemented systems (CryptGPU and CrypTen) continue to perform worse than the C++-implemented frameworks. The performance gap between CryptGPU and CrypTen widens compared to the inference results shown in Table 4.4. For instance, CryptGPU can be up to 7.4× faster than CrypTen when training CIFAR10 on VGG16. Nonetheless, CryptGPU remains 4.5× slower than Force. Compared to the other three Piranha-based systems (P-SecureML, P-Falcon and P-FantasticFour), Force is also faster regarding all experiments. On average, Force is 4.9× faster than P-SecureML, 1.8× faster than P-Falcon and 4× faster than P-FantasticFour.

5. Large-Scale Two-Party Gradient Boosting Decision Tree Training via Function Secret Sharing

5.1. Gradient Boosting Decision Tree

In this section, we introduce the training and inference algorithms for gradient boosting decision trees model (GBDT). We use the split metric of XGBoost as an example, but we note that our work is applicable in other tree variants such as the classification and regression tree (CART) [22]. We also note that portions of this section are written verbatim from [75].

5.1.1. GBDT Training

Let $\mathbf{X} \in \mathbb{R}^{N \times F}$ denote a dataset consisting of N samples and F features. A Gradient Boosted Decision Tree (GBDT) model constructs an ensemble of T decision trees $\{f_t\}_{t=1}^{T}$ sequentially, where t indexes each tree and D denotes the maximum depth constraint imposed on each tree.

For each sample $\mathbf{x}_i \in \mathbb{R}^F$, the prediction of the ensemble is defined as

$$\hat{y}_i = \sum_{t=1}^{T} f_t(\mathbf{x}_i). \tag{5.1}$$

In this work, we assume that all decision trees are perfect binary trees in order to characterize the maximum computational overhead. Under this assumption, each tree contains

$$n_{\text{non}} = 2^D - 1$$

internal (non-leaf) nodes and

$$n_{\text{leaf}} = 2^D$$

leaf nodes.

Split Candidate Construction. At each internal node, the learning algorithm aims to determine the optimal split feature and corresponding threshold value. Following [34, 83], we adopt an approximate split-finding strategy in which candidate split points are determined by partitioning each feature into B buckets according to its empirical distribution.

For a feature f, we denote by $\mathcal{I}^{f,b}$ the set of sample indices assigned to the b-th bucket. Consequently, each feature induces $(B-1)$ split candidates, resulting in a total of $F \cdot (B-1)$ candidates at each node.

Gradient Statistics. Let L denote a twice-differentiable loss function measuring the discrepancy between the true label y_i and the prediction $\bar{y}_{i,(t-1)}$ obtained from the previous boosting iteration. The initial prediction vector $\bar{\mathbf{y}}_0$ is set to a predefined constant.

At iteration t, the first- and second-order gradient statistics are computed as

$$g_i = \partial_{\bar{y}_{i,(t-1)}} L(y_i, \bar{y}_{i,(t-1)}), \qquad h_i = \partial^2_{\bar{y}_{i,(t-1)}} L(y_i, \bar{y}_{i,(t-1)}). \tag{5.2}$$

For each bucket (f, b), we aggregate the statistics as

$$G^{f,b} = \sum_{i \in \mathcal{I}^{f,b}} g_i, \qquad H^{f,b} = \sum_{i \in \mathcal{I}^{f,b}} h_i. \tag{5.3}$$

For the i-th split candidate of feature f, the buckets are partitioned into left and right subsets. The corresponding cumulative first-order statistics for the left and right subsets are defined as

$$G_L = \sum_{b \le i} G^{f,b}, \qquad G_R = \sum_{i < b \le B} G^{f,b},$$

with H_L and H_R defined analogously for the second-order statistics.

Split Evaluation. The quality of each candidate split is evaluated using the XGBoost gain function:

$$\text{Gain} = \frac{1}{2}\left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right) - \gamma, \tag{5.4}$$

where λ and γ are regularization parameters. The split maximizing Gain is selected.

Tree Expansion and Leaf Weight Computation. After selecting the optimal split, samples are partitioned into the left and right child nodes, and the procedure is applied recursively until the maximum depth D is reached.

For a leaf node with sample index set $\mathcal{I}^{\text{leaf}}$, the optimal leaf weight is computed as

$$\omega = -\frac{\sum_{i \in \mathcal{I}^{\text{leaf}}} g_i}{\sum_{i \in \mathcal{I}^{\text{leaf}}} h_i + \lambda}. \tag{5.5}$$

The final model is obtained as the ensemble of the T trees constructed in this manner.

5.1.2. Private GBDT Training

In the **vertical federated learning** (VFL) setting, we assume that the dataset $\mathbf{X}$ is vertically partitioned as

$$\mathbf{X} = \mathbf{X}_0 \,||\, \mathbf{X}_1,$$

where P_0 holds $\mathbf{X}_0 \in \mathbb{R}^{N \times F_0}$ and P_1 holds $\mathbf{X}_1 \in \mathbb{R}^{N \times F_1}$. Both parties share the same set of N samples but possess disjoint feature subsets. Additionally, without loss of generality, we assume that P_0 holds the label vector $\mathbf{Y} \in \mathbb{R}^N$.

Following prior work [56, 103, 108, 171], an indicator bit revealing the owner of the best split feature is disclosed during split selection. Subsequently, the index of the selected split candidate is revealed only to the corresponding feature owner, who records the split locally. The other party learns nothing beyond the indicator bit.

After a split is determined, the parties must securely update the sample sets associated with the two child nodes. Otherwise, revealing the updated sample assignments would leak the relative ordering of samples in plaintext. Existing approaches [56, 103, 108, 171] employ a secret-sharing (SS)-based method, where the shared node indicator vector $[\mathbf{u}_j]^{\mathsf{B}}$ is updated into $[\mathbf{u}_{2j}]^{\mathsf{B}}$ and $[\mathbf{u}_{2j+1}]^{\mathsf{B}}$ for the left and right child nodes, respectively.

Specifically, by invoking a secure select functionality with inputs $[\mathbf{g}_j]$ and $[\mathbf{u}_{2j}]^{\mathsf{B}}$, the parties obtain the updated shared gradient vector $[\mathbf{g}_{2j}]$ for the left child node. The same procedure applies to the Hessian vector $[\mathbf{h}_j]$. The gradient (and Hessian) shares for the right child node can then be computed locally as

$$[\mathbf{g}_{2j+1}] = [\mathbf{g}_j] - [\mathbf{g}_{2j}],$$

and analogously for $[\mathbf{h}_{2j+1}]$.

5.2. Secure Bucket Aggregation

5.2.1. The Bucket Aggregation Functionality.

Initially, a feature holder $\mathsf{P}_h \in \{\mathsf{P}_0, \mathsf{P}_1\}$ holds $\mathcal{I}^{f,b}$ for each bucket b of a feature f, where $i \in \mathcal{I}^{f,b}$ indicates that the sample x_i belongs to the bucket b. Alternatively, $\mathcal{I}^{f,b}$ can be interpreted as a vector $\mathbf{s}^{f,b}$, where $s_i^{f,b} = 1$ if $i \in \mathcal{I}^{f,b}$ and $s_i^{f,b} = 0$ otherwise. Then at any node j, parties hold a freshly updated $[\mathbf{g}_j]$. The bucket aggregation functionality $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{BucAgg}}$ outputs a secretly shared $z_j^{f,b} = \sum_{i \in \mathcal{I}^{f,b}} g_{j,i}$, which is the same as computing the dot product $z_j^{f,b} = \mathbf{g}_j \cdot \mathbf{s}^{f,b}$. We formally describe the functionality $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{BucAgg}}$ in Fig. 5.1.

Functionality $\mathcal{F}_{\text{2PC}}^{\text{BucAgg}}$

Internal state: **ready** $\in$ {true, false}.
Initialization: Set **ready** = false.
Compute:

- Upon receiving $\mathcal{I}^{f,b}$ from P_h for each bucket b of each feature f, check whether **ready** is set to **false**:
  - If yes, record all $\mathcal{I}^{f,b}$, set **ready** = true.
  - Otherwise, send (failed, P_h, sid) to P_h.
- Upon receiving (Comp, $[\mathbf{g}_j]_i$, P_i, sid) from each $\mathsf{P}_i \in \mathcal{P}$ at a node j, and $[z_j^{f,b}]_c$ for each bucket b of each feature f at node j from the adversary corrupting $\mathbf{P_c}$:
  1. Compute $\mathbf{g}_j = [\mathbf{g}_j]_h + [\mathbf{g}_j]_{1-h} \bmod 2^k$.
  2. For each bucket b of each feature f, compute $z_j^{f,b}$, where $z_j^{f,b} = \sum_{i \in \mathcal{I}^{f,b}} g_{j,i}$. Then compute $[z_j^{f,b}]_{1-c} = z_j^{f,b} - [z_j^{f,b}]_c \bmod 2^k$.
  3. For each bucket b of each feature f, send $([z_j^{f,b}]_i, \mathsf{P}_i, \text{sid})$ to $\mathsf{P}_i \in \mathcal{P}$.

Figure 5.1.: Two-Party Bucket Aggregation Functionality

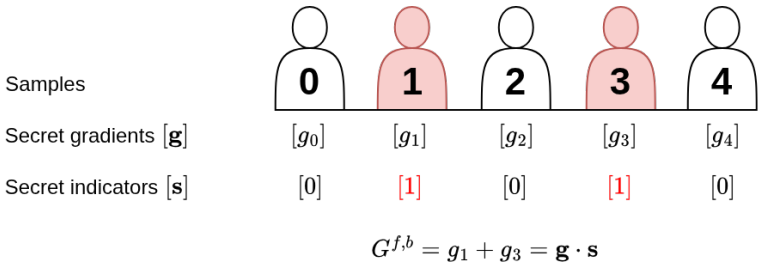


Figure 5.2.: Bucket Aggregation via Arithmetic Multiplication

5.2.2. Indicator-Based Solution

We first introduce the indicator-based solution adopted in [173]. As a running example, suppose that the total number of samples is $N = 5$, and that P_h holds $\mathcal{I}^{f,b} = \{1, 3\}$ for a particular bucket b of feature f. In the following, we focus exclusively on this bucket and mark the **corresponding samples** in red for illustration. The secret-shared gradient sum corresponding to this bucket is computed as

$$[G^{f,b}] = [g_1] + [g_3].$$

As shown in Fig. 5.2, the index set $\mathcal{I}^{f,b}$ can equivalently be represented as an indicator vector $\mathbf{s} \in \{0,1\}^N$, where

$$s_i = \begin{cases} 1, & \text{if } i \in \mathcal{I}^{f,b}, \\ 0, & \text{otherwise.} \end{cases}$$

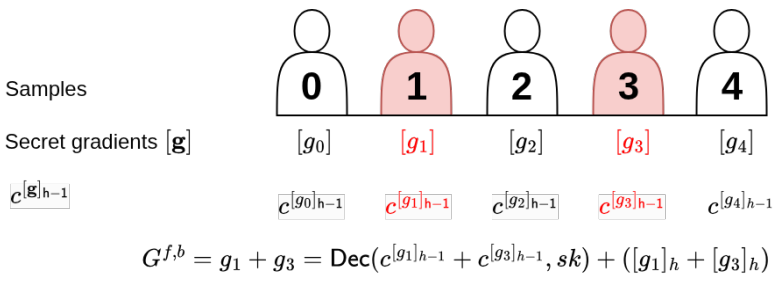


Figure 5.3.: Bucket Aggregation via Homomorphic Encryption

After P_h secretly shares the (arithmetic) indicator vector $\mathbf{s}$, the aggregated gradient $[G^{f,b}]$ is computed as the secure dot product of the secret-shared vectors $[\mathbf{s}]$ and $[\mathbf{g}]$. Consequently, the parties invoke a secure multiplication protocol to evaluate the dot product, which is realized using precomputed Beaver triples [10].

5.2.3. HE-based Solution

Squirrel [108] adopts a homomorphic encryption (HE)-based bucket aggregation protocol, as illustrated in Fig. 5.3. Specifically, P_{1-h} encrypts its local arithmetic shares under the public key pk and sends them to P_h:

$$c^{[\mathbf{g}]_{1-h}} = \mathsf{Enc}([\mathbf{g}]_{1-h}, \mathsf{pk}).$$

Exploiting the additive homomorphism of the encryption scheme, P_h can compute

$$c^{[g_1]_{1-h}+[g_3]_{1-h}} = c^{[g_1]_{1-h}} + c^{[g_3]_{1-h}},$$

which corresponds to an encryption of the aggregated share $[G^{f,b}]_{1-h}$. Meanwhile, P_h locally computes its own share as

$$[G^{f,b}]_h = [g_1]_h + [g_3]_h \bmod 2^k.$$

However, directly returning $c^{[g_1]_{1-h}+[g_3]_{1-h}}$ to P_{1-h} would leak information about the index set $\mathcal{I}^{f,b}$. Indeed, upon decryption, P_{1-h} would obtain $[g_1]_{1-h} + [g_3]_{1-h}$, thereby learning which gradients were aggregated. To prevent this leakage, Squirrel [108] applies a masking technique. Instead of returning the raw encrypted sum, P_h samples a random mask r and sends back an encryption of $[G^{f,b}]_{1-h} - r$, while setting its own local share to r. As a result, the two parties obtain a valid arithmetically shared $G^{f,b}$ without revealing $\mathcal{I}^{f,b}$.

5.2.4. Permutation-Based Solution

Finally, [56] proposes a permutation-based solution using MPC. The core idea is to let P_h securely permute the vector $[\mathbf{g}]$, such that after permutation, the parties can easily

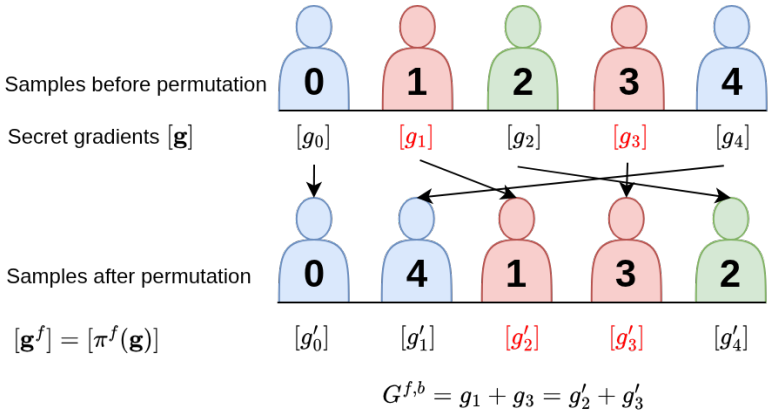


Figure 5.4.: Bucket Aggregation via Secure Permutation

determine $[G^{f,b}]$ for a bucket b by aggregating their local shares.[1] We let π^f denote the desired permutation as shown in Fig. 5.4. For illustration purposes, we mark samples in the same bucket using colors, e.g., $\mathbf{x}_2$ and $\mathbf{x}_4$ in red before the permutation. Thus, the computed (shared) result becomes $[\mathbf{g}^f] = [\pi^f(\mathbf{g})]$. Now similar to the indicator-based solution [173], the parties can generate correlated randomness in the preprocessing stage to accelerate the online computation of the secure permutation [56]. We formally define the so-called *permutation triple* as follows:

Definition 5.2.1 (*Permutation Triple*). A *permutation triple* is defined as $(\pi_1, [\mathbf{r}], [\mathbf{r}'])$, where π_1 is a permutation over $\{0, \ldots, m-1\}$, $\mathbf{r} \in \mathbb{Z}_{2^k}^m$, and $\mathbf{r}' = \pi_1(\mathbf{r})$.

As described in [56], we always let P_h hold the permutation π_1. In addition, $[\mathbf{r}]$ and $[\mathbf{r}']$ are secretly shared between P_h and P_{1-h}, where $\mathbf{r}'$ is the permuted vector $\mathbf{r}$ applying the permutation π_1. In the online stage, parties first reveal $\mathbf{g} - \mathbf{r}$ at P_h's side. Knowing both π_1 and π^f enables P_h to compute a fresh permutation π_2, such that $\pi_2(\pi_1(\cdot)) = \pi^f$. During the protocol execution, P_h sends π_2 to P_{1-h}, enabling P_{1-h} to compute $[\pi^f(\mathbf{g})]_{1-h} = \pi_2([\mathbf{r}']_{1-h})$. Note that sending π_2 to P_{1-h} does not leak any information of π^f to P_{1-h}, since π_1 remains hidden. From P_h's point of view, it sets $[\pi^f(\mathbf{g})]_h = \pi^f(\mathbf{g} - \mathbf{r}) + \pi_2([\mathbf{r}']_h)$. The correctness follows from:

$$\begin{aligned}
[\pi^f(\mathbf{g})]_h + [\pi^f(\mathbf{g})]_{1-h} &= \pi^f(\mathbf{g} - \mathbf{r}) + \pi_2([\mathbf{r}']_h) + \pi_2([\mathbf{r}']_{1-h}) \\
&= \pi^f(\mathbf{g} - \mathbf{r}) + \pi_2([\pi_1(\mathbf{r})]_h) + \pi_2([\pi_1(\mathbf{r})]_{1-h}) \\
&= \pi^f(\mathbf{g} - \mathbf{r}) + [\pi^f(\mathbf{r})]_h + [\pi^f(\mathbf{r})]_{1-h} \\
&= \pi^f(\mathbf{g}).
\end{aligned}$$

[1] The number of entries in each bucket is fixed and public. As shown in Fig. 5.4, the parties always compute $[G^{f,b}] = [g'_2] + [g'_3]$ after the permutation.

Protocol Π_{KeyBuc}

Private inputs: For each bucket b of each feature f, P_h holds $\mathcal{I}^{f,b}$, where $|\mathcal{I}^{f,b}| \leq N$, $1 \leq b \leq B$ and $1 \leq f \leq F_h$. At a node j, parties hold $[\mathbf{g}_j]$, where $\mathbf{g}_j = (g_{j,0}, ..., g_{j,N-1})$.

Outputs: For each bucket b of each feature f, parties output $[z_j^{f,b}]$, where $z_j^{f,b} = \sum_{i \in \mathcal{I}^{f,b}} g_{j,i}$.

Preprocessing:

- P_h sends $(\mathsf{KBucGen}, j, \mathsf{P}_h, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}$, for each bucket b of each feature f:
  - If $j = 1$, receives $\mathbf{r}^{f,b}$, where $\mathbf{r}^{f,b} = (r_0^{f,b}, ..., r_{N-1}^{f,b})$.
  - Receives $[m_j^{f,b}]_h$, where $m_j^{f,b} = \mathbf{k}_j \cdot \mathbf{r}^{f,b} \bmod 2^k$.
- P_{1-h} sends $(\mathsf{KBucGen}, j, \mathsf{P}_{1-h}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}$:
  - Receives $\mathbf{k}_j$, where $\mathbf{k}_j = (k_{j,0}, ...k_{j,N-1})$.
  - For each bucket b of each feature f receives $[m_j^{f,b}]_{1-h}$.

Protocol:

1. If $j = 1$, for each bucket b of each feature f:
   - P_h computes a secret vector $\mathbf{s}^{f,b} = (s_0^{f,b}, ..., s_{N-1}^{f,b})$, where $s_i^{f,b} = 1$ if $i \in \mathcal{I}^{f,b}$ and $s_i^{f,b} = 0$ otherwise.
   - P_h computes $\mathbf{v}^{f,b} = \mathbf{s}^{f,b} - \mathbf{r}^{f,b} \bmod 2^k$, then sends $\mathbf{v}^{f,b}$ to P_{1-h}.
2. P_{1-h} computes $\mathbf{q}_j = [\mathbf{g}_j]_{1-h} - \mathbf{k}_j \bmod 2^k$, sends $\mathbf{q}_j$ to P_h.
3. Parties locally output for each bucket b of each feature f:
   - P_h sets $[z_j^{f,b}]_h = \sum_{i \in \mathcal{I}^{f,b}} q_{j,i} + \sum_{i \in \mathcal{I}^{f,b}} [g_{j,i}]_h + [m_j^{f,b}]_h \bmod 2^k$.
   - P_{1-h} sets $[z_j^{f,b}]_{1-h} = \mathbf{k}_j \cdot \mathbf{v}^{f,b} + [m_j^{f,b}]_{1-h} \bmod 2^k$.

Figure 5.5.: Two-Party Keyed Bucket Aggregation Protocol

5.2.5. Keyed Bucket Aggregation

We now introduce an MPC-based bucket aggregation variant, which further improves the communication and computational efficiency compared to the previously discussed protocols above. In the following part, the resulting framework is denoted as NodeGuard. We formally describe the keyed bucket aggregation protocol Π_{KeyBuc} in Fig. 5.5.

Preprocessing. A key observation is that $\mathcal{I}^{f,b}$ remains constant throughout the entire GBDT training process. If we examine the indicator-based solution described in Section 5.2.2 more closely, we observe that the feature holder P_h must redundantly mask its secret indicator $\mathbf{s}^{f,b}$ at each tree node, which results in huge communication overhead. Although such masking is inevitable while executing the secure multiplication protocol, we propose a solution to eliminate this redundancy by introducing a new primitive *keyed bucket triple*, which can be generated in the preprocessing stage:

Definition 5.2.2 (*Keyed Bucket Triple*)**.** A *keyed bucket triple* is defined as $(\mathbf{r}^{f,b}, \mathbf{k}_j, [m_j^{f,b}])$, where $\mathbf{r}^{f,b}, \mathbf{k}_j \in \mathbb{Z}_{2^k}^m$ and $m_j^{f,b} = \mathbf{r}^{f,b} \cdot \mathbf{k}_j \bmod 2^k$.

In Π_{KeyBuc}, the parties invoke the functionality $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}$ to generate such triples. We formally describe $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}$ in Fig. A.1. At the initial node $j = 1$, we let P_h hold $\mathbf{r}^{f,b}$ to mask its secret $\mathbf{s}^{f,b}$, where $\mathbf{r}^{f,b} = (r_0^{f,b}, ..., r_{N-1}^{f,b})$. Then at any node j (including the initial node), we let P_{1-h} hold $\mathbf{k}_j$ to mask its secret share $[\mathbf{g}_j]_{1-h}$. We call $\mathbf{k}_j$ the key of the node j. In addition, the parties hold a secretly shared $[m_j^{f,b}]$, such that $m_j^{f,b} = \mathbf{r}^{f,b} \cdot \mathbf{k}_j \bmod 2^k$. The functionality $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}$ can be implemented by a trusted third party (TTP), which faithfully distributes required correlated randomness. In the absence of a TTP, the parties can instead execute a secure multiplication protocol, where P_h keeps $\mathbf{r}^{f,b}$ and P_{1-h} keeps $\mathbf{k}_j$ as output, respectively.

Online Stage. Again, we start at the initial node $j = 1$. For each bucket b of feature f, P_h masks its secret indicator $\mathbf{s}^{f,b}$ with $\mathbf{r}^{f,b}$ and sends the masked vector $\mathbf{v}^{f,b}$ to P_{1-h}. We note that this message is sent only once during the entire GBDT training process. Then, at any node j (including the initial node), P_{1-h} computes $\mathbf{q}_j = [\mathbf{g}_j]_{1-h} - \mathbf{k}_j \bmod 2^k$ and sends $\mathbf{q}_j$ to P_h. Upon receiving $\mathbf{q}_j$, P_h computes the sum of the corresponding $[g_{j,i}]_h$ and $q_{j,i}$ for all $i \in \mathcal{I}^{f,b}$. Meanwhile, P_{1-h} computes $\mathbf{k}_j \cdot \mathbf{v}^{f,b} \bmod 2^k$ without learning $\mathcal{I}^{f,b}$. Finally, both parties unmask their local shares by adding $[m_j^{f,b}]$. We show correctness as follows:

$$
\begin{aligned}
[z_j^{f,b}]_h + [z_j^{f,b}]_{1-h} &= \sum_{i \in \mathcal{I}^{f,b}} q_{j,i} + \sum_{i \in \mathcal{I}^{f,b}} [g_{j,i}]_h + [m_j^{f,b}]_h + \mathbf{k}_j \cdot \mathbf{v}^{f,b} + [m_j^{f,b}]_{1-h} \\
&= \mathbf{q}_j \cdot \mathbf{s}^{f,b} + [\mathbf{g}_j]_h \cdot \mathbf{s}^{f,b} + \mathbf{k}_j \cdot \mathbf{s}^{f,b} + (m_j^{f,b} - \mathbf{k}_j \cdot \mathbf{r}^{f,b}) \\
&= [\mathbf{g}_j]_h \cdot \mathbf{s}^{f,b} + (\mathbf{q}_j + \mathbf{k}_j) \cdot \mathbf{s}^{f,b} \\
&= \mathbf{g}_j \cdot \mathbf{s}^{f,b} \bmod 2^k.
\end{aligned}
$$

5.3. FSS Compute Functionality

We now present the idea for constructing a functionality that replaces FSS gates within a protocol execution. We show the FSS compute functionality $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ in Fig. 5.6. First, the parties initialize all wires by sending an initialization query to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$, which allows $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ to generate records in a list $\mathcal{L}$. The parties can then reconstruct the values for a wire by sending their respective shares to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. Again, this allows $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ to update the list $\mathcal{L}$ and record the reconstructed wire values. Finally, the parties can compute a function f by sending f along with the corresponding input wire IDs $\{\mathsf{wid}_j\}$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. Assuming that f takes m inputs, $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ retrieves the values of the m input wires from $\mathcal{L}$ and computes the output as $z = f(x_0, ..., x_{m-1})$. This reflects the fact that once an

$\mathcal{F}_{\text{FSS}}^{\text{Comp}}$

Internal list $\mathcal{L}$: Initialized as $\mathcal{L} = \emptyset$.
Compute:

- Upon receiving (FSSInit, wid, sid) from both $\mathsf{P}_i \in \mathcal{P}$:
  1. Check the list $\mathcal{L}$, continue if the record $(\mathbb{G}, \text{wid}, \text{sid}, \cdot)$ does not exist.
  2. Update $\mathcal{L} = \mathcal{L} \cup (\mathbb{G}, \text{wid}, \text{sid}, \cdot)$.
- Upon receiving (Reconst, wid, $[x]_i$, P_i, sid) from both $\mathsf{P}_i \in \mathcal{P}$:
  1. Check the list $\mathcal{L}$, continue if the record $(\mathbb{G}, \text{wid}, \text{sid}, \cdot)$ exists.
  2. Reconstruct $x = [x]_0 + [x]_1 \in \mathbb{G}$.
  3. Update $\mathcal{L} = \mathcal{L} \cup (\mathbb{G}, \text{wid}, \text{sid}, x)$.
- Upon receiving (Assign, wid, x, P_h, sid) from P_h:
  1. Check the list $\mathcal{L}$, continue if the record $(\mathbb{G}, \text{wid}, \text{sid}, \cdot)$ exists and wid is not assigned.
  2. Update $\mathcal{L} = \mathcal{L} \cup (\mathbb{G}, \text{wid}, \text{sid}, x)$.
  3. Send a notification to P_{1-h}.
- Let $f : \mathbb{G}_j^m \to \mathbb{G}^{\text{out}}$, upon receiving (Compute, f, $\{\text{wid}_j\}$, sid) from both $\mathsf{P}_i \in \mathcal{P}$:
  1. Check the list $\mathcal{L}$, continue if for $j \in [m]$ all records $(\mathbb{G}_j, \text{wid}_j, \text{sid}, x_j)$ exist and $f \in \{f^{\text{dReLU}}, f^{\text{Select}}, \text{xor}\}$.
  2. Wait for the adversary to input $[z]_c \in \mathbb{G}^{\text{out}}$.
  3. Compute $z = f(x_0, ..., x_{m-1}) \in \mathbb{G}^{\text{out}}$. Determine $[z]_{c-1} = z - [z]_c \in \mathbb{G}^{\text{out}}$.
  4. Send $[z]_i$ to P_i.

Figure 5.6.: Two-Party FSS Compute Functionality

input wire value is reconstructed, it can be reused across multiple function evaluations.[2] Such reusability relies on circuit-dependent correlated randomness. We refer to [18] for a detailed discussion.

For the remainder of this section, we let f^{dReLU} denote the **dReLU** function (also known as the comparison function), and f^{Select} denote the **select** function.[3] We now give the formal definitions.

Definition 5.3.1 (dReLU Function)**.** The function $f^{\text{dReLU}} : \mathbb{Z}_{2^k} \to \mathbb{Z}_2$ is defined as

$$f^{\text{dReLU}}(x) := \begin{cases} 1 & \text{if } x \geq 0, \\ 0 & \text{otherwise,} \end{cases}$$

where x is interpreted as a signed integer in two's complement representation over $\mathbb{Z}_{2^k}$.

Definition 5.3.2 (Select Function)**.** The function $f^{\text{Select}} : \mathbb{Z}_2 \times \mathbb{Z}_{2^k} \times \mathbb{Z}_{2^k} \to \mathbb{Z}_{2^k}$ is defined as

$$f^{\text{Select}}(b, x, y) := b \cdot x + (1 - b) \cdot y,$$

[2] For FSS gates, a wire offset (also known as a mask) can serve multiple gates incident to the same wire.
[3] In this thesis, f^{Select} always selects between an input x and 0.

Protocol Π_{SiBuc}

Private inputs: For each bucket b of each feature f, the feature owner P_h holds $\mathcal{I}^{f,b}$, where $|\mathcal{I}^{f,b}| \leq N$, $1 \leq b \leq B$ and $1 \leq f \leq F_h$. Note that $\mathsf{P}_h \in \{\mathsf{P}_0, \mathsf{P}_1\}$. At a node j, parties hold $[\mathbf{g}_j]$, where $\mathbf{g}_j = (g_{j,0}, ..., g_{j,N-1})$.
Public inputs: Public parameters.
Outputs: For each bucket b of each feature f, parties output $[z_j^{f,b}]$, where $z_j^{f,b} = \sum_{i \in \mathcal{I}^{f,b}} g_{j,i}$.
Initialize: Parties initialize all wires by sending corresponding wire IDs to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$.
Protocol:

1. If $j = 1$, for each bucket b of each feature f, P_h assigns $\mathbf{s}^{f,b}$ to the corresponding wire ID by sending $\mathbf{s}^{f,b}$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$, which sends a notification to P_{1-h}.
2. Then at the current node j, Parties reconstruct $\mathbf{g}_j$ by sending respective $[\mathbf{g}_j]_i$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$.
3. Parties send f^{Select} along with the corresponding wire IDs of $\mathbf{s}^{f,b}$ and $\mathbf{g}_j$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. They receive $[\mathbf{q}_j^{f,b}]$ as output, where $q_{j,i}^{f,b} = g_{j,i}$ if $s_i^{f,b} = 1$ and $q_{j,i}^{f,b} = 0$ otherwise.
4. Parties compute $[z_j^{f,b}] = \sum_i q_{j,i}^{f,b}$.

Figure 5.7.: Two-Party Silent Bucket Aggregation Protocol

where $b \in \mathbb{Z}_2$ and $x, y \in \mathbb{Z}_{2^k}$.

We naturally extend $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ to support all secure function evaluations that can be realized using FSS gates. In the context of SiGBDT, we restrict $f \in \{f^{\mathsf{dReLU}}, f^{\mathsf{Select}}, \mathsf{xor}\}$, as these operations suffice for protocol execution.

5.4. Protocol Building Blocks to Train GBDT

In this section, we build modular protocols using $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ explained above. We first introduce our novel silent bucket aggregation protocol Π_{SiBuc}, which applies f^{Select} at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ as backbone. We compare Π_{SiBuc} to state-of-the-art bucket aggregation protocols through a complexity comparison, and we show that Π_{SiBuc} achieves a better communication efficiency. We also briefly review the parallelized FSS argmax protocol Π_{PaArg} introduced in [145]. In addition, we propose our silent argmax protocol Π_{SiArg}, which leverages f^{Select} and f^{dReLU} at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ as subroutine protocols.

5.4.1. Silent Bucket Aggregation

We formally describe our protocol Π_{SiBuc} in Fig. 5.7. At the initial node $j = 1$, P_h assigns the secret indicator vector $\mathbf{s}^{f,b}$ to the corresponding input wire. This is the only point in the protocol where P_h injects the value $\mathbf{s}^{f,b}$ into the computation. This wire value will be reused

Table 5.1.: Communication rounds and overhead required in Π_{SiBuc} compared to other protocols.

Framework	Protocol	Rounds	Overhead
Xie. et al. [173]	Indicator	$T\, n_{\mathsf{non}}$	$4FB\, k\, Tn_{\mathsf{non}}N$
Fang et al. [56]	Permutation	$2T\, n_{\mathsf{non}}$	$2Fk\, Tn_{\mathsf{non}}N$
Squirrel [108]	LWE-HE	$2T\, n_{\mathsf{non}}$	$> 2\log_2 q\, Tn_{\mathsf{non}}N$
NodeGuard [43]	Sharing	$T\, n_{\mathsf{non}}$	$FBNk + 2TNk\, n_{\mathsf{non}}$
SiGBDT [75]	FSS	$T\, n_{\mathsf{non}}$	$FBN + 2TNk\, n_{\mathsf{non}}$

at all subsequent nodes. Then at any node j, the only communication overhead arises during the reconstruction of the gradient g_j by the parties. Compared to the keyed bucket aggregation protocol Π_{KeyBuc} from [43], Π_{SiBuc} requires less communication overhead because the "assign" request of the value $\mathbf{s}^{f,b}$ operates over $\mathbb{Z}_2$ rather than $\mathbb{Z}_{2^k}$.

Complexity Analysis. We now analyze the communication rounds and overhead required for each secure bucket aggregation protocol in Table 5.1. During the execution of Π_{SiBuc}, the parties exchange $FBN + 2TNk\, n_{\mathsf{non}}$ bits of information over Tn_{non} rounds, achieving the minimum number of communication rounds among all comparable frameworks. As mentioned in Section 5.4.1, the protocol Π_{SiBuc} already optimizes the communication of Π_{KeyBuc} from [43]. Therefore, we focus on the comparison of Π_{SiBuc} against the indicator-based solution Π_{IndiBuc} from [173], the HE-based solution Π_{HEBuc} from [108] and the permutation-based solution Π_{PermBuc} from [56]. Although Π_{IndiBuc} achieves the same minimum communication rounds as Π_{SiArg} and Π_{KeyBuc}, it requires a communication overhead of $4FB\, k\, Tn_{\mathsf{non}}N$ bits, which is the highest among all current solutions and strictly greater than that of Π_{SiBuc}. In certain cases, we observe that Π_{PermBuc} and Π_{HEBuc} can be more communication-efficient than Π_{SiBuc}. Compared to Π_{PermBuc}, the protocol Π_{SiBuc} performs better when $2\, k\, Tn_{\mathsf{non}}(F-1) > FB$, which roughly corresponds to the condition $2\, k\, Tn_{\mathsf{non}} > B$. Similarly, compared to Π_{HEBuc}, Π_{SiBuc} has the advantage if $2Tn_{\mathsf{non}} > \frac{FB}{\log_2 q - k}$. We adopt the choice of $q = 2^{109}$ for the security guarantee of LWE-based HE as in Squirrel [108]. And we follow the parameter settings suggested in [132] for training a large-scale GBDT model (e.g. $T = 100$, etc). Under these common settings, the proposed protocol Π_{SiBuc} typically achieves better efficiency in both communication and computation.

5.4.2. Silent Argmax

We assume that the parties hold a secretly shared array $[\mathbf{x}]$ of size n and they want to compute $[\mathbf{z}]^{\mathsf{B}} = \mathsf{argmax}([\mathbf{x}])$, where $z_i = 1$ if $x_i = \mathsf{max}(\mathbf{x})$ otherwise $z_i = 0$.[4] In Π_{PaArg} [145], the parties execute two sequential FSS-based protocols. First, they compute

[4] In this thesis, the computed argmax result is a boolean shared one hot encoding $[\mathbf{z}]^{\mathsf{B}}$ of size n, which differs from the traditional argmax output.

Protocol Π_{SiArg}

Private inputs: Parties holds $[\mathbf{x}]$.
Public inputs: Public parameters. $|\mathbf{x}| = n$, and we suppose that n is a power of two.
Outputs: Parties output $[\mathbf{z}]^{\mathsf{B}}$, where $z_i = 1$ if $x_i = \max(\mathbf{x})$ and $z_i = 0$ otherwise.
Initialize: Parties initialize all wires by sending corresponding wire IDs to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$.
Protocol:

- Parties set $[\mathbf{x}^{\mathsf{tmp}}] = [\mathbf{x}]$.
- For csize $\neq 1$, parties do:
  1. Set $[\mathbf{x}^{\mathsf{high}}] = ([x_0^{\mathsf{tmp}}], ..., [x_{\frac{\mathsf{csize}}{2}-1}^{\mathsf{tmp}}])$, $[\mathbf{x}^{\mathsf{low}}] = ([x_{\frac{\mathsf{csize}}{2}}^{\mathsf{tmp}}], ..., [x_{\mathsf{csize}-1}^{\mathsf{tmp}}])$, compute $[\mathbf{x}^{\mathsf{diff}}] = [\mathbf{x}^{\mathsf{high}}] - [\mathbf{x}^{\mathsf{low}}]$.
  2. Parties reconstruct $\mathbf{x}^{\mathsf{diff}}$ by sending respective $[\mathbf{x}^{\mathsf{diff}}]_i$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$.
  3. Parties send f^{dReLU} along with the corresponding wire IDs of $\mathbf{x}^{\mathsf{diff}}$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. They receive $[\mathbf{b}]^{\mathsf{B}}$ as output.
  4. Parties now reconstruct $\mathbf{b}$ by sending respective $[\mathbf{b}]_i$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$.
  5. Parties send f^{Select} along with the corresponding wire IDs of $\mathbf{x}^{\mathsf{diff}}$ and $\mathbf{b}$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. They receive $[\mathbf{q}]$ as output.
  6. Parties reconstruct $\mathbf{q}$ by sending respective $[\mathbf{q}]_i$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$.
  7. Parties compute and update $[\mathbf{x}^{\mathsf{tmp}}] = [\mathbf{q}] + [\mathbf{x}^{\mathsf{low}}]$.
  8. Parties set csize = csize/2
- After obtain $[x^{\mathsf{max}}]$, parties compute $[\mathbf{x}'] = [\mathbf{x}] - [x^{\mathsf{max}}]$.
- Parties reconstruct $[\mathbf{x}']$ by sending respective $[\mathbf{x}']_i$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$.
- Parties send f^{dReLU} along with the corresponding wire IDs of $\mathbf{x}'$ to $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. They receive $[\mathbf{z}]^{\mathsf{B}}$ as output.

Figure 5.8.: Two-Party Silent Argmax Protocol

$n(n-1)$ pairwise comparisons in parallel, with a small probability of error.[5] Then, the parties perform n equality checks to determine the position of the maximum value. As a result, the number of communication rounds is significantly improved, where the entire protocol can be executed over two rounds. But the communication and computation overhead will explode if the comparison size increases [145].

We propose a secure argmax protocol variant Π_{SiArg} in Fig 5.8. For simplicity, we assume that the size n is a power of two and each x_i is different. Initially, parties input a secretly shared vector $[\mathbf{x}]$ of size n. In csize $\neq 1$, the parties begin a computation round by reconstructing $\mathbf{x}^{\mathsf{diff}}$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. Then parties call $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ to compute $f^{\mathsf{dReLU}}(\mathbf{x}^{\mathsf{diff}})$, which outputs $[\mathbf{b}]^{\mathsf{B}}$. After reconstructing $\mathbf{b}$, the parties call $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ to compute $f^{\mathsf{Select}}(\mathbf{x}^{\mathsf{diff}}, \mathbf{b})$,

[5] [18] suggests evaluating two separate DCF instances to guarantee perfect correctness of a comparison protocol. In [145], however, only one DCF is evaluated for efficiency, which results in an imperfect comparison outcome.

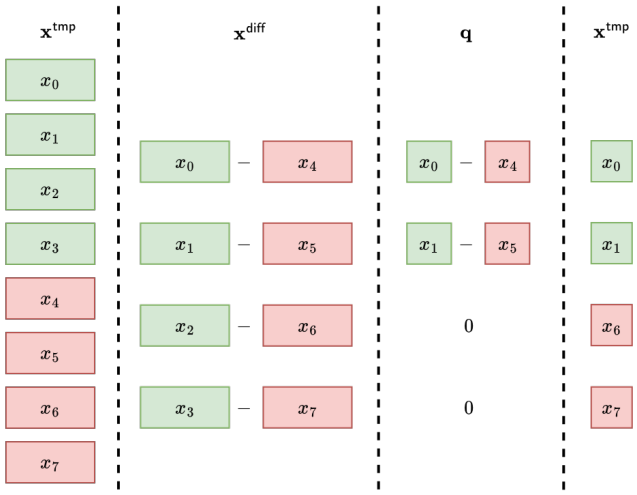


Figure 5.9.: Demonstration of Comparison Candidate Updates in the First Round of Π_{SiBuc}

which outputs $[\mathbf{q}]$. We note that $q_i = 0$, if $x_i - x_{\text{csize}/2+i} < 0$, otherwise $q_i = x_i - x_{\text{csize}/2+i}$. Thus, after reconstructing $\mathbf{q}$ and updating $[\mathbf{x}^{\text{tmp}}]$, the parties now obtain a set of new candidates for the next computation round. They continue the above computation until they find the maximum $[x^{\max}]$. Note that the desired result is a Boolean-shared one hot encoding $[\mathbf{z}]^{\text{B}}$, it is thus equivalent to compute the difference vector $[\mathbf{x}'] = [\mathbf{x}] - [x^{\max}]$, then use the reconstructed x' to compute $f^{\text{dReLU}}(\mathbf{x}')$.

In Fig. 5.9, we demonstrate an example of the comparison candidate updates in Π_{SiArg}. We suppose that $n = 8$ and the comparison result for the first round is $\mathbf{b} = (1, 1, 0, 0)$. We mark $[\mathbf{x}^{\text{high}}]$ in green and $[\mathbf{x}^{\text{low}}]$ in red. Then the expected select result should be $\mathbf{q} = (x_1 - x_5, x_2 - x_6, 0, 0)$. After computing $[\mathbf{x}^{\text{tmp}}] = [\mathbf{q}] + [\mathbf{x}^{\text{low}}]$, we observe that $\mathbf{x}^{\text{tmp}} = (x_1, x_2, x_7, x_8)$, which contains all comparison candidates for the next round.

Efficiency Comparison. For SiGBDT, we always let the parties evaluate two DCFs as proposed in [18] to ensure the perfect correctness of the comparison (dReLU) implementation. Recall that in Π_{PaArg} [145], the parties evaluate $n(n-1)$ DCFs and n equality checks, which results in overall $2n^2k$ bits ($2n(n-1)k$ and $2nk$, respectively) communication overhead over two communication rounds. In contrast, Π_{SiArg} requires $2n$ DCF evaluations over $2\log n + 1$ rounds, where overall $2nk + 2n$ bits are communicated.

5.4.3. Silent Node Split

After the parties find the best split candidate for the current node j, the parties have to update the gradient $[\mathbf{g}_{2j}]$ for the child node $2j$ and $[\mathbf{g}_{2j+1}]$ for the child node $2j+1$, correspondingly.

Recap of SS-based Node Split. State-of-the-art works [56, 108, 171, 173]) apply an SS-based approach to update the gradients. We let $\mathcal{F}_{\text{SS}}^{\text{Select}}$ denote the SS-based functionality, which takes as input secret shares and evaluates f^{Select}. Suppose that the current (shared) sample space is $[\mathbf{u}_j]^{\text{B}}$. The feature holder P_h, which knows the best split candidate, has to share an indicator vector $\mathbf{c}_j$, such that the (shared) sample space of child node $2j$ and $2j+1$ can be computed as $[\mathbf{u}_{2j}]^{\text{B}} = [\mathbf{u}_j]^{\text{B}} * [\mathbf{c}_j]^{\text{B}}$ and $[\mathbf{u}_{2j+1}]^{\text{B}} = [\mathbf{u}_j]^{\text{B}} \oplus [\mathbf{u}_{2j}]^{\text{B}}$. Then the parties can call $\mathcal{F}_{\text{SS}}^{\text{Select}}$ with $[\mathbf{g}_j]$ and $[\mathbf{u}_{2j}]$ as input to compute $[\mathbf{g}_{2j}]$ and update $[\mathbf{g}_{2j+1}] = [\mathbf{g}_j] - [\mathbf{g}_{2j}]$. To avoid the conversion between arithmetic sharing and boolean sharing required in $\mathcal{F}_{\text{SS}}^{\text{Select}}$, both $\mathbf{u}_j$ and $\mathbf{c}_j$ can also be shared arithmetically. The parties then simply compute $[\mathbf{u}_{2j}] = [\mathbf{u}_j] \circ [\mathbf{c}_j]$ and $[\mathbf{g}_{2j}] = [\mathbf{g}_j] \circ [\mathbf{u}_{2j}]$ to update the sample space.

FSS-based Node Split. We realize that the node split computation can be regarded as a byproduct of the Π_{SiBuc} execution. We replace the above element-wise AND computation and $\mathcal{F}_{\text{SS}}^{\text{Select}}$ computation by computing f^{Select} at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$ two times.

At the initial node $j = 1$, the first select computation can be omitted since $[\mathbf{u}_2]^{\text{B}} = [\mathbf{c}_1]^{\text{B}}$ (and $[\mathbf{u}_3]^{\text{B}} = [\mathbf{c}_1]^{\text{B}} \oplus \mathbf{1}$). For the second one, we observe that $\mathbf{g}_0$ is already reconstructed during the execution of Π_{SiBuc} Thus, the parties only have to reconstruct $\mathbf{u}_1$ at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$. Since both $\mathbf{g}_0$ and $\mathbf{u}_1$ are already reconstructed, the parties can compute f^{Select} at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$ to obtain $[\mathbf{g}_1]$. The communication overhead for the initial node $j = 1$ is thus only $2N$ within one single round.

Then at any node $j \neq 1$, $\mathbf{u}_j$ is already reconstructed at its father node, and $\mathbf{g}_j$ is revealed in Π_{SiBuc}. Thus, for the first select computation, the parties just reconstruct $\mathbf{c}_j$ at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$, then compute f^{Select} at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$ with the corresponding wire IDs of $\mathbf{u}_j$ and $\mathbf{c}_j$. They receive $[\mathbf{u}_{2j}]^{\text{B}}$ as output. And for the second select computation, the parties now reconstruct $\mathbf{u}_{2j}$, then compute $f^{\text{Select}}(\mathbf{g}_j, \mathbf{u}_{2j})$ at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$. They receive $[\mathbf{g}_{2j}]$ as output. For the right child node, the parties can locally compute $[\mathbf{g}_{2j+1}] = [\mathbf{g}_j] - [\mathbf{g}_{2j}]$. Since both $\mathbf{u}_{2j}$ and $\mathbf{u}_j$ are already reconstructed, the parties can simply compute $\mathbf{u}_{2j+1} = \mathbf{u}_{2j} \oplus \mathbf{u}_j$ at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$. The communication overhead for any node $j \neq 1$ is thus only $4N$ over two rounds.

5.5. Security of Π_{KeyBuc}, Π_{SiBuc} and Π_{SiArg}

In this section, we provide security proofs to Π_{KeyBuc} and Π_{SiArg}. We note that in Π_{SiBuc}, parties simply compute f^{Select} at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$ and locally summarize output vector entries. We thus directly call Π_{SiBuc} in our training algorithm 1 without explicitly giving a proof to show that Π_{SiBuc} uc-realizes $\mathcal{F}_{\text{2PC}}^{\text{BucAgg}}$.

Theorem 5.5.1. Protocol Π_{SiArg} shown in Fig. 5.8 uc-realizes $\mathcal{F}_{\text{2PC}}^{\text{SiArg}}$ in the $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$-hybrid model, in the presence of a semi-honest adversary who can corrupt $\text{P}_i \in \{\text{P}_h, \text{P}_{1-h}\}$, with static corruption.

Similar to the protocol Π_{SiBuc}, the parties just interact with $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ and locally process the results (secret shares) from $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. We let $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{SiArg}}$ denote the argmax functionality, which allows the adversary to choose its own output share $[\mathbf{z}]^{\mathsf{B}}$. Below, we give a proof sketch to Theorem 5.5.1:

Proof Sketch. The simulator $\mathcal{S}$ can be constructed by simply emulating $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$. $\mathcal{S}$ receives input wire shares from the adversary $\mathcal{A}$, it also emulates $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ by receiving output shares from $\mathcal{A}$. We let $[\cdot]_{\mathcal{A}}$ denote the $\mathcal{A}$'s share of a secret value. $\mathcal{S}$ receives $[\mathbf{x}']_{\mathcal{A}}$ and $[x^{\max}]_{\mathcal{A}}$ from $\mathcal{A}$ (note that both values are determined by $\mathcal{A}$ itself), $\mathcal{A}$'s input can thus be extracted by computing $[\mathbf{x}]_{\mathcal{A}} = [\mathbf{x}']_{\mathcal{A}} - [x^{\max}]_{\mathcal{A}}$. $\mathcal{S}$ also receives $\mathcal{A}$'s share of $[\mathbf{z}]^{\mathsf{B}}$, which will be sent along with $[\mathbf{x}]_{\mathcal{A}}$ to $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{SiArg}}$. It is easy to see that the real execution and the ideal execution is indistinguishable, since the simulator only has to emulate $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ interacting with $\mathcal{A}$ during the entire simulation. □

Theorem 5.5.2. Protocol Π_{KeyBuc} shown in Fig. 5.5 uc-realizes $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{BucAgg}}$ described in Fig. 5.1 in the $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}$-hybrid model, in the presence of a semi-honest adversary who can corrupt $\mathsf{P}_i \in \{\mathsf{P}_h, \mathsf{P}_{1-h}\}$, with static corruption.

We defer the detailed security proof to Appendix A.4.

5.6. Secure GBDT Training via FSS

We have shown that the functionality $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ can be applied to construct the necessary building blocks for training a GBDT model. Now we use $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ to build our silent GBDT training framework SiGBDT as shown in Algorithm 1. In the subsequent sections, we let $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{Grad}}$ denote the gradient computation functionality, $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{Hess}}$ denote the Hessian computation functionality, $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{Gain}}$ denote the gain computation functionality, $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{BestParty}}$ denote the best-party selection functionality and $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{Leaf}}$ denote the leaf-node computation functionality. We omit explicit definitions of these functionalities and assume that they are realized via MPC protocols.[6] We refer the reader to Section 5.1.1 for details on the computation of the corresponding functionalities.

Initialization. At the beginning of training, each party can locally determine the indices $\mathcal{I}^{f,b}$ for each bucket of its own features, which can then be converted to the corresponding vectors $\mathbf{s}^{f,b}$. As the label holder, P_0 computes the initial gradient $\mathbf{g}$ and Hessian $\mathbf{h}$ at the start of the training process and shares $(\mathbf{g}, \mathbf{h}, \mathbf{Y})$ with P_1. This sharing is performed locally by deriving (pseudo-)randomness using a pseudo-random function PRF. Afterwards, the parties initialize all wires with wire IDs at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$.

[6] In realizing $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{Grad}}$, $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{Hess}}$, $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{Gain}}$, and $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{Leaf}}$, we may invoke $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ as a subroutine.

Algorithm 1 Silent GBDT Training Framework

Require: Hyperparameters(T, D, B, $\hat{y}^0$)
P_0: dataset $\mathbf{X}_0 \in \mathbb{R}^{N \times F_0}$, label $\mathbf{Y} \in \mathbb{R}^N$
P_1: dataset $\mathbf{X}_1 \in \mathbb{R}^{N \times F_1}$
Ensure: [GBDT]
Initialization:
P_0: $\mathcal{I}_0^{f,b}(\mathbf{s}_0^{f,b}) \leftarrow \mathsf{LocalBucket}(\mathbf{X_0})$, P_1: $\mathcal{I}_1^{f,b}(\mathbf{s}_1^{f,b}) \leftarrow \mathsf{LocalBucket}(\mathbf{X_1})$
P_0: Set $\bar{\mathbf{Y}} = \hat{y}^0$, $\mathbf{g}, \mathbf{h} \leftarrow \mathsf{LocalGradient}(\mathbf{Y}, \bar{\mathbf{Y}})$
$[\mathbf{g}], [\mathbf{h}], [\mathbf{Y}] \leftarrow \mathsf{Share}(\mathbf{g}, \mathbf{h}, \mathbf{Y})$
P_0 and P_1: call $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ to initialize all wires.
Online training:
for $t = 1, 2, 3, \ldots T$ **do**
  if $t > 1$ **then**
    Gradient: $[\mathbf{g}_0] \leftarrow \mathcal{F}_{\mathsf{2PC}}^{\mathsf{Grad}}([\mathbf{Y}], [\bar{\mathbf{Y}}])$
    Hessian: $[\mathbf{h}_0] \leftarrow \mathcal{F}_{\mathsf{2PC}}^{\mathsf{Hess}}([\mathbf{Y}], [\bar{\mathbf{Y}}])$
  end if
  $\mathbf{u}_0 = \{1, 1, 1, \ldots, 1\} \in \mathbb{R}^N$
  for $j = 1, 2, 3, \ldots n_{\mathsf{non}} + n_{\mathsf{leaf}}$ **do**
    $d = \mathsf{Tree_t.node}(j).\mathsf{depth}$
    if $d < D$ **then**
      for $f \in [F]$ and $b \in [B]$ **do**
        Reconstruct $\mathbf{g}_j$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$, $[\mathbf{g}_j^{f,b}] \leftarrow f^{\mathsf{Select}}(\mathbf{g}_j, \mathrm{s}^{f,b})$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$, $[G_j^{f,b}] \leftarrow \sum_i g_{j,i}^{f,b}$
        Reconstruct $\mathbf{h}_j$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$, $[\mathbf{h}_j^{f,b}] \leftarrow f^{\mathsf{Select}}(\mathbf{h}_j, \mathrm{s}^{f,b})$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$, $[H_j^{f,b}] \leftarrow \sum_i h_{j,i}^{f,b}$
      end for
      For $f \in [F]$, $i \in [B-1]$: $[G_{L,j}^{f,i}] = \sum_{b \le i} [G_j^{f,b}]$, $[G_{R,j}^{f,i}] = \sum_{i < b \le B} [G_j^{f,b}]$
      For $f \in [F]$, $i \in [B-1]$: $[H_{L,j}^{f,i}] = \sum_{b \le i} [H_j^{f,b}]$, $[H_{R,j}^{f,i}] = \sum_{i < b \le B} [H_j^{f,b}]$
      Set $[\mathbf{G}_{L,j}] \leftarrow ([G_{L,j}^{f,i}])_{f \in [F], i \in [B-1]}$, $[\mathbf{G}_{R,j}] \leftarrow ([G_{R,j}^{f,i}])_{f \in [F], i \in [B-1]}$
      Set $[\mathbf{H}_{L,j}] \leftarrow ([H_{L,j}^{f,i}])_{f \in [F], i \in [B-1]}$, $[\mathbf{H}_{R,j}] \leftarrow ([H_{R,j}^{f,i}])_{f \in [F], i \in [B-1]}$
      $[\mathbf{Gain}] \leftarrow \mathcal{F}_{\mathsf{2PC}}^{\mathsf{Gain}}([\mathbf{G}_{L,j}], [\mathbf{G}_{R,j}], [\mathbf{H}_{L,j}], [\mathbf{H}_{R,j}])$
      $[\mathbf{z}]^{\mathsf{B}} \leftarrow \mathcal{F}_{\mathsf{2PC}}^{\mathsf{SiArg}}([\mathbf{Gain}])$
      Reconstruct a bit $\mathsf{P}_h \leftarrow \mathcal{F}_{\mathsf{2PC}}^{\mathsf{BestParty}}([\mathbf{z}]^{\mathsf{B}}, |F_0|)$
      P_h: receives $[\mathbf{z}]_{1-h}^{\mathsf{B}}$ from P_{1-h}
      $[\mathbf{c}_j]^{\mathsf{B}} = \mathsf{P}_h.\mathsf{ApplySplit}(\mathbf{X}_h)$
      if $d = 1$ **then** Set $[\mathbf{u}_{2j}]^{\mathsf{B}} = [\mathbf{c}_j]^{\mathsf{B}}$
      else if $d > 1$ **then**
        Reconstruct $\mathbf{c}_j$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$, $[\mathbf{u}_{2j}]^{\mathsf{B}} \leftarrow f^{\mathsf{Select}}(\mathbf{u}_j, \mathbf{c}_j)$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$
      end if
      Reconstruct $\mathbf{u}_{2j}$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$, $[\mathbf{g}_{2j}] \leftarrow f^{\mathsf{Select}}(\mathbf{g}_j, \mathbf{u}_{2j})$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$
      $[\mathbf{g}_{2j+1}] = [\mathbf{g}_j] - [\mathbf{g}_{2j}]$, receive $\mathbf{u}_{2j+1}$ from $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$ s.t. $\mathbf{u}_{2j+1} = \mathbf{u}_j \oplus \mathbf{u}_{2j}$
    else if $d = D$ **then**
      Leaf: $[\omega_j] \leftarrow \mathcal{F}_{\mathsf{2PC}}^{\mathsf{Leaf}}([\mathbf{g}_j], [\mathbf{h}_j])$
    end if
  end for
  $[\boldsymbol{\omega}] = ([\omega_{n_{\mathsf{non}}+1}], \ldots, [\omega_{n_{\mathsf{non}}+n_{\mathsf{leaf}}}])$, reconstruct $\boldsymbol{\omega}$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$
  $\mathbf{u} = \{\mathbf{u}_{n_{\mathsf{non}}+1}, \ldots, \mathbf{u}_{n_{\mathsf{non}}+n_{\mathsf{leaf}}}\}$
  $[\mathbf{pred}] \leftarrow f^{\mathsf{Select}}(\mathbf{u}, \boldsymbol{\omega})$ at $\mathcal{F}_{\mathsf{FSS}}^{\mathsf{Comp}}$
  $[\bar{\mathbf{Y}}] = [\bar{\mathbf{Y}}] + [\mathbf{pred}]$
end for

Online Training. When training of the second tree begins, the parties must invoke the gradient computation functionality $\mathcal{F}_{\text{2PC}}^{\text{Grad}}$ and the hessian computation functionality $\mathcal{F}_{\text{2PC}}^{\text{Hess}}$ (Equation 5.2) to securely compute $[\mathbf{g}_1]$ and $[\mathbf{h}_1]$ for the root node of the new tree. We use the Sigmoid Cross-Entropy loss in this work. During the execution of protocols realizing $\mathcal{F}_{\text{2PC}}^{\text{Grad}}$ and $\mathcal{F}_{\text{2PC}}^{\text{Hess}}$, the non-linear sigmoid function is approximated using a piecewise linear function, as proposed in [182]. This approximation requires the involved parties to collaboratively and securely determine the interval in which a shared input resides. Thus, both $\mathcal{F}_{\text{2PC}}^{\text{Grad}}$ and $\mathcal{F}_{\text{2PC}}^{\text{Hess}}$ can be realized in the $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$-hybrid model, where f^{dReLU} is computed in parallel. Regarding Equations 5.4 and 5.5, the gain computation functionality $\mathcal{F}_{\text{2PC}}^{\text{Gain}}$ and the leaf computation functionality $\mathcal{F}_{\text{2PC}}^{\text{Leaf}}$ must be realized using a division protocol that computes the division of two secret-shared values. We refer the reader to [31] for further details on executing the dReLU protocol within a secure normalization procedure, which is used to securely compute the normalized input (as the initial value) for the Goldschmidt division algorithm [62].

At any node of a tree, the parties compute f^{Select} at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$ to securely compute the bucket aggregation for each bucket. Then, they invoke $\mathcal{F}_{\text{2PC}}^{\text{Gain}}$ to compute the gain for each split candidate and determine the maximum gain by invoking the argmax functionality $\mathcal{F}_{\text{2PC}}^{\text{SiArg}}$ (implemented by Π_{SiArg}). As explained in Section 5.1.2, an indicator bit is revealed to identify which party holds the best split candidate. Let P_h denote the party holding the best split feature. The argmax result is then reconstructed at P_h, enabling P_h to apply the node split and record the split information. Afterward, P_h shares the split indicator vector $[\mathbf{c}_j]^{\mathsf{B}}$ for the current node, and the parties proceed to apply the node split method described in Section 5.4.3. Finally, if the maximum depth D is reached, the parties compute the leaf weight $[\omega]$ by invoking $\mathcal{F}_{\text{2PC}}^{\text{Leaf}}$ and compute the prediction result $[\mathbf{pred}]$ by computing f^{Select} at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$.

5.7. Evaluation

We implement SiGBDT in `C++`. The evaluation of f^{dReLU} and xor at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$ is implemented using the protocol proposed in [18], and the evaluation of f^{Select} at $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$ is based on the method described in [74].

5.7.1. Evaluation Setup

Testbed Environment. We conduct our evaluations on a server equipped with two `Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz` and `16` $\times$ `128GB` of RAM. To simulate the network conditions, we use the `tc` tool [7] to emulate two representative environments:

- **LAN**: 1 `Gbps` bandwidth + 0.2`ms` round-trip latency;

[7] https://man7.org/linux/man-pages/man8/tc.8.html

Table 5.2.: Accuracy Experiment for Regression (Reg.) and Classification (Cls.)

Type	Dataset	N	F	Metric	Plaintext	SiGBDT
Reg.	Concrete	1,030	8	RMSE	4.10	3.89
	Energy	19,735	27		83.59	83.53
Cls.	Breast	683	30	ACC	1	1
	Credit	30,000	23		0.83	0.83

- **WAN**: 100 `Mbps` bandwidth + `40ms` round-trip latency.

Baseline. We compare SiGBDT with the plaintext XGBoost library to verify the model's accuracy in the non-secure setting. To demonstrate the efficiency benefits of adopting FSS-based protocols, we further compare SiGBDT against several state-of-the-art privacy-preserving solutions (excluding non-open-source implementations such as [108]):

- MP-XGB [173] [8], evaluated natively on CPU;
- SecretFlow [56, 111] [9], evaluated natively on CPU;
- Pivot [171] [10], evaluated in the pre-built docker on CPU.

For all experiments, we focus on the online computation stage. If configurable (in both SiGBDT and secretflow), we set the number of threads to 12. We run SecretFlow using the `SEMI2K` protocol in `standalone mode` to enable fast local simulation. The arithmetic sharing length ℓ is set to 64, and the fractional precision p is set to 20 throughout all experiments.

5.7.2. Accuracy Verification

We employ the plaintext XGBoost library to train models on public datasets, serving as a baseline for comparison. Subsequently, we train models on the same datasets using SiGBDT and compare the resulting training outcomes.

Training Parameters and Datasets. Without loss of generality, we set the training parameters as follows: tree number $T = 20$, max depth $D = 4$, max bucket size $B = 16$. We choose `Concrete Compressive Strength` [11] (1,030 samples and 8 features) and `Energy` [12] (19,735 samples and 27 features) as training datasets for regression task. And we choose `Breast`

[8] https://github.com/HikariX/MP-FedXGB at commit *46807ea*
[9] https://github.com/secretflow/secretflow at commit *d7bb1d1*
[10] https://hub.docker.com/repository/docker/lemonwyc/pivot at commit *942b66c*
[11] https://archive.ics.uci.edu/dataset/165/concrete+compressive+strength
[12] https://www.kaggle.com/loveall/appliances-energy-prediction

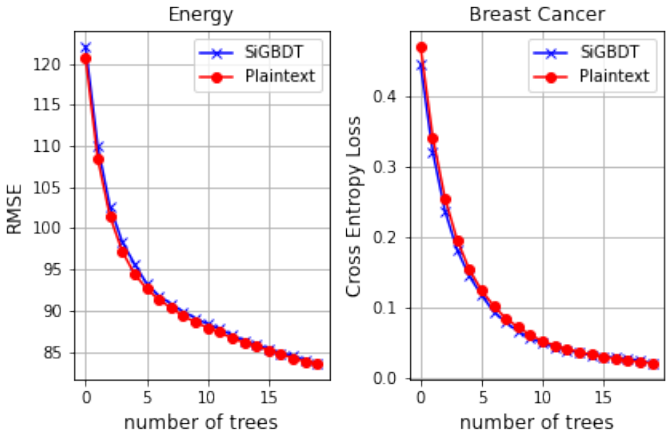


Figure 5.10.: Training Loss Comparison on `Energy` and `Breast Cancer` Datasets

`Cancer` [13] (683 samples and 30 features) and `Credit` [14] (30,000 samples and 23 features) as training datasets for classification task. If a dadaset does not provide a test set, we randomly split the data into 80% training and 20% testing samples; otherwise, we use the original split. We use Root Mean Square Error (RMSE) as the evaluation metric for regression tasks and accuracy for classification tasks. During the training with SiGBDT features are evenly distributed between both parties.

We train a GBDT model on each dataset five times and report the average accuracy. Table 5.2 presents the training accuracy of both SiGBDT and the plaintext model. In addition, Fig. 5.10 shows the training loss curves for both models. We observe that SiGBDT achieves accuracy comparable to the plaintext model, with nearly overlapping training loss curves. The slight difference may be attributed to the use of fixed-point computation with reduced precision. Another possible source of error is the approximation protocols used for operations such as division and the sigmoid function.

5.7.3. Efficiency Experiment

We generate random synthetic datasets to train GBDT models using different frameworks and record the end-to-end training time under both LAN and WAN settings. For the default parameters, we set the number of samples to $N = 10,000$, maximum tree depth to $D = 4$, feature size to $F = 10$, and maximum bucket size to $B = 8$. In addition, we vary the training parameters based on the default settings to evaluate the scalability of each framework. In the following experiment, we set the number of Trees to $T = 10$. We also implement pure SS-based protocols in SiGBDT, which is denoted as SiGBDT-SS [15].

[13] https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic
[14] https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset
[15] SiGBDT-SS applies the indicator-based solution for the bucket aggregation protocol.

Table 5.3.: End-to-End Training Time (Seconds per Tree) in **LAN**

N	F	B	D	SiGBDT	SiGBDT-SS	SecretFlow	MP-XGB	Pivot
10,000	10	8	4	**1.83**	5.73	11.31	19.78	239.82
50,000	10	8	4	**6.08**	13.73	20.51	174.52	499.27
10,000	20	8	4	**2.92**	8.20	11.38	42.94	501.98
10,000	10	16	4	**3.19**	8.70	10.60	38.81	434.02
10,000	10	8	5	**2.74**	9.43	11.26	31.15	398.98

Table 5.4.: End-to-End Training Time (Seconds per Tree) in **WAN**

N	F	B	D	SiGBDT	SiGBDT-SS	SecretFlow	MP-XGB	Pivot
10,000	10	8	4	**10.91**	54.67	87.07	444.10	7,285.89
50,000	10	8	4	**13.79**	79.75	164.20	1,309.65	7,781.40
10,000	20	8	4	**13.76**	75.81	108.63	971.43	17,925.70
10,000	10	16	4	**13.31**	73.49	106.37	745.56	15,473.43
10,000	10	8	5	**18.68**	76.76	119.70	1,052.94	13,075.34

End-to-end Runtime in LAN. We show the average runtime measured in **LAN** in Table 5.3. In short, SiGBDT outperforms all state-of-the-art frameworks including SiGBDT-SS:

- Compared to SiGBDT-SS, SiGBDT is 2.26 to 3.44 × faster;
- Compared to SecretFlow, SiGBDT is 3.32 to 6.18 × faster;
- Compared to MP-XGB, SiGBDT is 10.81 to 28.70 × faster;
- Compared to Pivot, SiGBDT is 82.11 to 171.91 × faster.

End-to-end Runtime in WAN. We then show the average runtime measured in **WAN** in Table 5.4. Again, we observe that the performance advantage of SiGBDT over other frameworks further increases:

- Compared to SiGBDT-SS, SiGBDT is 4.11 to 5.78 × faster;
- Compared to SecretFlow, SiGBDT is 6.40 to 11.90 × faster;
- Compared to MP-XGB, SiGBDT is 40.71 to 94.97 × faster;
- Compared to Pivot, SiGBDT is 564.28 to 1,302.74 × faster.

Compared to the LAN setting, SiGBDT demonstrates a significantly greater performance advantage over state-of-the-art frameworks including SiGBDT-SS in the WAN environment. This is because SiGBDT applies $\mathcal{F}_{\text{FSS}}^{\text{Comp}}$ to significantly reduce communication overhead compared to other frameworks.

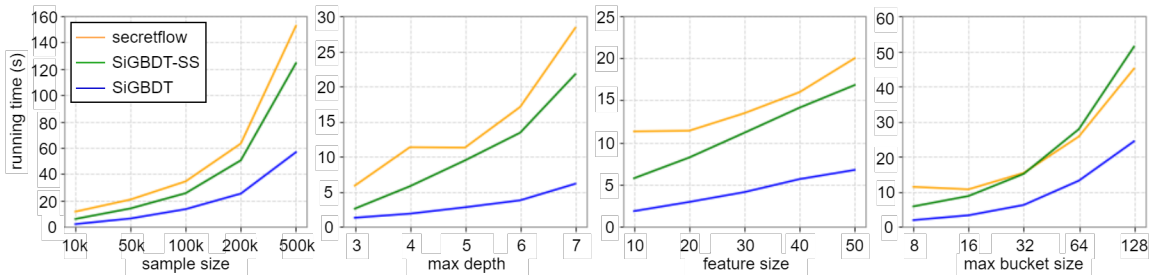


Figure 5.11.: Impact of Increasing Training Set Size on Runtime in Each Framework in **LAN** (Lower Is Better)

Impact of Increased Training Size. To assess the scalability of each framework, we systematically vary the training size by adjusting parameters including sample size, maximum tree depth, feature dimensionality, and maximum bucket size. The resulting average runtimes are presented in Fig. 5.11. Due to rapid runtime increasing trend of both MP-XGB and Pivot, we only plot the runtime of SiGBDT, SiGBDT-SS and SecretFlow in Fig. 5.11. We observe that as sample size and max depth size increase, both SiGBDT-SS and SecretFlow tend to exhibit exponentially increasing runtime, while SiGBDT shows only linear increase. As feature size and max bucket size increase, all three frameworks exhibit similar growth patterns: a linear increase with feature size and an exponential increase with bucket size. However, the runtime growth rate of SiGBDT is more gradual.

5.7.4. Private Inference Efficiency

Previous work [56, 108] assumes that each data entry used for inference is split between two model hosts. However, a common application scenario is that a customer who wants to run inference using a model shared between two cloud service providers may require that the provided input dataset remains undisclosed. In this thesis, we assume that the input dataset used for inference is private. This indicates that the whole inference must be computed with MPC protocols. Given a fully grown GBDT model with depth set to $D = 4$, and a dataset containing $N = 500,000$ samples with $F = 10$ features, SiGBDT takes 91.27 seconds to process all samples in **LAN**. On average, private inference is performed with SiGBDT at a rate of 5,478 samples per second.

6. Secure Aggregation For Federated Learning with Malicious Security against a Dishonest Majority

6.1. Federated Learning

Federated Learning (FL) [115] enables collaborative training of machine learning models across multiple data owners without requiring raw data to be centralized. In an FL system, each client trains the model locally using its private dataset and transmits only the resulting model updates (e.g., gradients or parameters) to a central server. The server aggregates these updates to produce a global model, which is then redistributed to the clients for the next training round. This process is repeated iteratively until convergence. It is crucial to achieve the following goals in order to maintain the robustness of Federated Learning (FL) systems: **(i) Input Privacy.** The private inputs of all honest clients must remain confidential. In particular, no information about an individual client's local update should be revealed beyond what can be inferred from the final aggregation result. **(ii) Input Integrity and Output Correctness.** A malicious server may deviate from the prescribed protocol, potentially leading to incorrect aggregation results, similar in effect to a model poisoning attack launched by a malicious client. Therefore, the protocol must guarantee that the aggregation is performed over authenticated inputs and that the final output is correct. **(iii) Poisoning Resilience.** The system must remain robust against adversarial updates. In particular, it should incorporate robust aggregation mechanisms capable of detecting and mitigating corrupted or malicious gradient updates, thereby preserving the reliability of the global model.

A straightforward approach is to directly apply the protocols proposed in the SPD$\mathbb{Z}_{2^k}$ framework [39] to achieve malicious security against a dishonest majority. However, this approach is suboptimal in the FL setting. In [39], the input protocol is designed under the assumption that every party acts both as an input provider and as a computation node. In particular, all parties participate in the subsequent computation after supplying their inputs. In FL, clients typically serve solely as input parties and do not necessarily participate in the aggregation process. Therefore, they should be treated differently from the servers that execute the secure computation. We address this problem by proposing two *input commitment* protocols that enable clients to efficiently share their inputs with the servers.

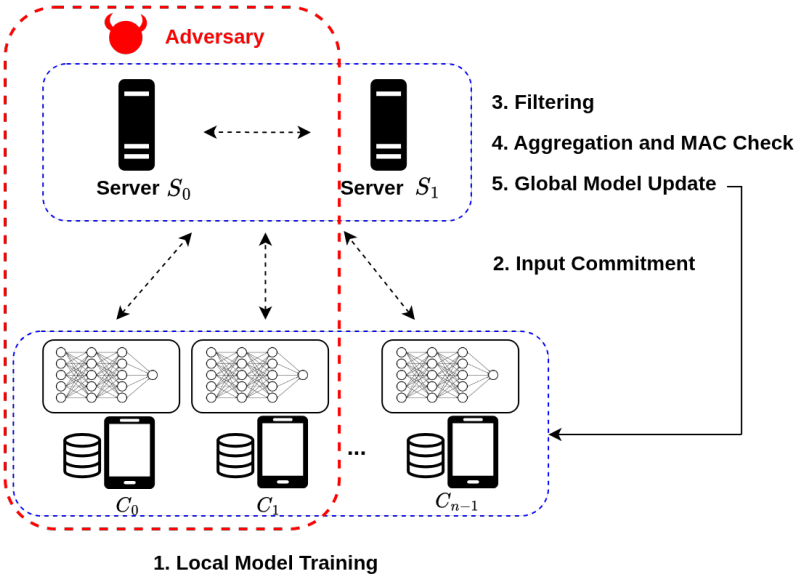


Figure 6.1.: Aggregation protocol with two servers under malicious security against a dishonest majority in `Alpha-DiHo` [77]

We first illustrate the overall workflow of the FL aggregation protocol in Fig. 6.1. After local model training, each client C_i securely shares its local gradient update $\mathbf{u}_i$ with the servers. The servers then perform filtering operations by enforcing both L_∞-norm and L_2-norm bounds. In this thesis, we additionally consider a *dynamic* L_2-norm bound that is computed based on the clients' private inputs. Since this bound depends on confidential information, it must remain hidden from any adversary. After filtering, the servers securely aggregate the **accepted** updates $\mathbf{u}_i$ provided by the clients. Compared to using a public bound, the key challenge of applying a dynamic bound is that aggregation must be carried out without revealing the bound itself. We propose an efficient silent select protocol Π^{SiSelect} to obliviously filter malicious gradient updates. To guarantee the integrity and correctness of the aggregated result $\mathbf{u}_i$, the servers perform MAC verification. Finally, the servers redistribute the aggregated result $\mathcal{U}_q$ to the clients.

Limitation of Norm-Based Defense. In this thesis, we apply L_∞-norm and L_2-norm based defenses to mitigate various sophisticated poisoning attacks [110, 149, 155]. As analyzed in RoFL [110], norm-based defenses provide sufficient mitigation against untargeted model poisoning attacks, in which the adversary aims only to degrade the usability of the final trained model [55]. They are also effective against data poisoning attacks, when prototypical data is maliciously modified and the adversary attempts to maximize attack performance by amplifying gradient updates. However, recent studies have highlighted the inherent limitations of such defenses against diverse backdoor attacks. For instance, the attack in [165] forces the model to misclassify data points lying on the tail of the input distribution. RoFL [110] further demonstrates that tail backdoors remain effective over extended training periods even under norm-based defense mechanisms. In addition, both [110, 165] demonstrate that a strong adversary can consistently manipulate the

global model on tail data points by periodically lowering the attack intensity, thereby evading norm-based defense detection. We refer readers to Flame [127] and RoFL [110] for comprehensive benchmark results.

6.2. Important Building Blocks

In this section, we introduce our input commitment protocol Π^{InCom} in the non-collusion setting and $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ in the server-client collusion setting. We then propose a silent select protocol Π^{SiSelect} to obliviously filter malicious gradient updates.

6.2.1. Input Commitment Protocol

We observe that each client operates independently in the *input commitment* stage. Thus, the overall computation can be viewed as a three-party input commitment protocol, which must be executed multiple times. During the protocol execution, C_i shares its input with the servers and assists them in generating the MAC. We first rule out the naive solution in which C_i simply shares its input with the servers, and the servers compute the authentication MAC on their own, since we cannot guarantee that a malicious server will use the exact share received from C_i when computing the MAC. For simplicity, we consider static corruption in this paper and discuss two cases as follows:

(i) Honest Majority: We note that the non-collusion setting corresponds to the honest-majority assumption in a three-party protocol. This implies that, in the federated learning scenario, either one server **or** multiple clients can be corrupted by a malicious adversary $\mathcal{A}$.

(ii) Dishonest Majority: Meanwhile, the server-client collusion setting corresponds to the dishonest-majority assumption in a three-party protocol. Again, stepping back from the input commitment stage and considering the entire aggregation protocol, this implies that a server **and** multiple clients can be corrupted by the same adversary $\mathcal{A}$.

6.2.1.1. Π^{InCom} under the Honest-Majority Assumption

We present a single execution of the protocol Π^{InCom} as described in Fig. 6.2, where one party from the set $\mathsf{C}_i, \mathsf{S}_0, \mathsf{S}_1$ may be corrupted. The key idea is to let C_i efficiently distribute its gradient shares **and** MAC shares by holding the global MAC key $\alpha = \alpha^0 + \alpha^1$. To reduce communication, we use a correlated randomness functionality $\mathcal{F}^{\mathsf{CR}}$ (described in Fig. A.8), between C_i and S_0. This functionality can be implemented by having the parties hold a pre-shared key and derive pseudo-randomness via a keyed pseudo-random function (PRF). Both servers then perform a *consistency check* to authenticate the distributed MAC shares.

Protocol Π^{InCom}

Private inputs: A client C_i holds $\mathbf{x}$, where $\mathbf{x} = (x_0, ..., x_{t-1}) \in \mathbb{Z}_{2^k}^t$ and $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$.
Public inputs: Public parameters k, s.
Outputs: S_j outputs $[\![\mathbf{x}]\!]^j = (\mathbf{x}^j, \mathbf{m}^j) \in \mathbb{Z}_{2^{k+s}}^t \times \mathbb{Z}_{2^{k+s}}^t$, where $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$.

Initialize: C_i and S_0 call their $\mathcal{F}^{\mathsf{CR,glo}}$ instance, receive $\alpha^0 \xleftarrow{\$} 2^s$. Then C_i and S_1 call their $\mathcal{F}^{\mathsf{CR,glo}}$ instance, receive $\alpha^1 \xleftarrow{\$} 2^s$.
Protocol:

1. C_i and S_0 call their $\mathcal{F}^{\mathsf{CR}}$ instance, receive $\mathbf{x}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}^t$, $x_t^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$ $\mathbf{m}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}^t$ and $m_t^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$.
2. C_i sets $\alpha = \alpha^0 + \alpha^1 \bmod 2^{k+s}$. It chooses $x_t^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$ and $x^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, such that $\mathbf{x} = \mathbf{x}^0 + \mathbf{x}^1 \bmod 2^k$.
3. C_i computes MACs as $\mathbf{m} = \alpha \circ (\mathbf{x}^0 + \mathbf{x}^1) \bmod 2^{k+2s}$ and $m_t = \alpha \cdot (x_t^0 + x_t^1) \bmod 2^{k+2s}$. Then it sets $\mathbf{m}^1 = \mathbf{m} - \mathbf{m}^0 \bmod 2^{k+2s}$ and $m_t^1 = m_t - m_t^0 \bmod 2^{k+2s}$.
4. C_i now sends $(\mathbf{x}^1, x_t^1, \mathbf{m}^1, m_t^1)$ to S_1.

Consistency Check:

5. S_0 and S_1 call $\mathcal{F}^{\mathsf{Rand}}$, receive $\mathbf{r} \in \mathbb{Z}_{2^s}^t$.
6. S_j computes $v^j = \sum_{h=0}^{t-1} x_h^j \cdot r_h + x_t^j \bmod 2^{k+2s}$ and $d^j = \sum_{h=0}^{t-1} m_h^j \cdot r_h + m_t^j \bmod 2^{k+2s}$.
7. S_j sends v^j to S_{j-1} and computes $v = v^0 + v^1 \bmod 2^{k+2s}$. It then commits to and opens $z^j = d^j - v \cdot \alpha^j \bmod 2^{k+2s}$ to S_{j-1}.
8. S_j computes $z = z^0 + z^1 \bmod 2^{k+2s}$ and checks if $z = 0$, aborts if not. Otherwise, P_j outputs $\mathbf{m}^j \bmod 2^{k+s}$.

Figure 6.2.: Input Commitment Protocol Π^{InCom} in the Honest-Majority Setting

Note that all C_i instances use the same global MAC key shares α_0 and α_1 during the execution of Π^{InCom}. These shares are obtained during the initialization phase by calling the 'global' version of the correlated randomness functionality, $\mathcal{F}^{\mathsf{CR,glo}}$ (see Fig. A.9). Unlike individual $\mathcal{F}^{\mathsf{CR}}$ instances, which use client-specific pre-shared keys, $\mathcal{F}^{\mathsf{CR,glo}}$ allows all clients to share the same pre-shared keys with S_0 and S_1, respectively. We also present two ways to bypass the reliance on $\mathcal{F}^{\mathsf{CR,glo}}$:

- Under the assumption of a secure broadcast channel (e.g., as defined in [9]), the servers can broadcast α_j to each C_i, since the broadcast functionality guarantees that all clients receive the same message.
- Alternatively, relying solely on peer-to-peer secure channels, the servers can first exchange commitments of α_j with each other and subsequently decommit to each client. Since at least one server is assumed to be honest, the clients are guaranteed to receive consistent shares of the global MAC key α^j from P_j, or consistent commitments to these shares from the other server P_{1-j}.

Protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$

Private inputs: A client C_i holds $\mathbf{x}$, where $\mathbf{x} = (x_0, ..., x_{t-1}) \in \mathbb{Z}^t_{2^k}$ and $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$.
Public inputs: Public parameters k, s.
Outputs: S_j outputs $[\![\mathbf{x}]\!]^j = (\mathbf{x}^j, \mathbf{m}^j) \in \mathbb{Z}^t_{2^{k+s}} \times \mathbb{Z}^t_{2^{k+s}}$, where $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$.

Initialize: S_0 chooses $\alpha^0 \xleftarrow{\$} \mathbb{Z}_{2^s}$. Then S_1 chooses $\alpha^1 \xleftarrow{\$} \mathbb{Z}_{2^s}$. S_0 initializes an instance of $\mathcal{F}^{\mathsf{vOLE}}$ with C_i, where S_0 inputs α^0. $(\mathsf{S}_0, \mathsf{S}_1)$ initialize another instance of $\mathcal{F}^{\mathsf{vOLE}}$, where S_1 inputs α^1.
Preprocessing:

1. S_0 calls its $\mathcal{F}^{\mathsf{CR}}$ instance (with C_i), receives $\mathbf{x}^0 \xleftarrow{\$} \mathbb{Z}^t_{2^{k+s}}$, $x^0_t \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$.
2. S_0 sets $\tilde{\mathbf{x}}^0 = (\mathbf{x}^0, x^0_t) \in (\mathbb{Z}^t_{2^{k+s}} \times \mathbb{Z}_{2^{k+2s}})$.
3. S_0 and S_1 call their $\mathcal{F}^{\mathsf{vOLE}}$ instance with input $(k+2s, k+s, t+1, \tilde{\mathbf{x}}^0)$ from S_0.
4. S_0 receives $\mathbf{b}^0$ and S_1 receives $\mathbf{a}^0$ such that $\mathbf{a}^0 = \mathbf{b}^0 + \alpha^1 \circ \tilde{\mathbf{x}}^0 \bmod 2^{k+2s}$.

Protocol:

1. C_i calls its $\mathcal{F}^{\mathsf{CR}}$ instance (with S_0), receives $\mathbf{x}^0 \xleftarrow{\$} \mathbb{Z}^t_{2^{k+s}}$, $x^0_t \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$.
2. C_i chooses $x^1_t \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$ and $x^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, such that $\mathbf{x} = \mathbf{x}^0 + \mathbf{x}^1 \bmod 2^k$. It sets $\tilde{\mathbf{x}} = (\mathbf{x}', x_t) \in \mathbb{Z}^t_{2^{k+s}} \times \mathbb{Z}_{2^{k+2s}}$[1], where $\mathbf{x}' = \mathbf{x}^0 + \mathbf{x}^1 \bmod 2^{k+s}$ and $x_t = x^0_t + x^1_t \bmod 2^{k+2s}$.
3. S_0 and C_i call their $\mathcal{F}^{\mathsf{vOLE}}$ instance with input $(k+2s, k+s, t+1, \tilde{\mathbf{x}})$ from C_i.
4. S_0 receives $\mathbf{a}^1$ and C_i receives $\mathbf{b}^1$ such that $\mathbf{a}^1 = \mathbf{b}^1 + \alpha^0 \circ \tilde{\mathbf{x}} \bmod 2^{k+2s}$.
5. C_i sends $(\mathbf{x}^1, x^1_t, \mathbf{b}^1)$ to S_1, which sets $\tilde{\mathbf{x}}^1 = (\mathbf{x}^1, x^1_t) \in (\mathbb{Z}^t_{2^{k+s}}, \mathbb{Z}_{2^{k+2s}})$.
6. The h-th MAC share is defined as follows:
   - S_0: $m^0_h = \mathbf{a}^1[h] - \mathbf{b}^0[h] \bmod 2^{k+2s}$.
   - S_1: $m^1_h = \mathbf{a}^0[h] - \mathbf{b}^1[h] + \alpha^1 \circ \tilde{\mathbf{x}}^1[h] \bmod 2^{k+2s}$

Consistency Check: Same as in Π^{InCom}.

Figure 6.3.: Input Commitment Protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ in the Dishonest-Majority Setting

We observe that if any C_i is maliciously corrupted, the MAC shares received by the servers may be incorrect. In Section 6.3.1, we show that the probability of such a corrupted C_i (or a set of corrupted clients $C_c \subseteq \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$) passing the consistency check is at most 2^{-s}, even if they know both α^0 and α^1.

6.2.1.2. $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ under the Dishonest-Majority Assumption

We present a single execution of $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ as described in Fig. 6.3, where the malicious adversary corrupts both the client C_i and one server from the set $\{\mathsf{S}_0, \mathsf{S}_1\}$. It is easy to see that the above protocol is no longer secure if C_i can collude with any server, since the adversary then holds both global MAC key shares α^0 and α^1, and can easily pass

the *consistency check*. We can adapt the randomness generation protocol proposed in [39] into an input protocol that fits our application scenario. However, this approach requires each client to perform the vector Oblivious Linear Function Evaluation (vOLE) functionality $\mathcal{F}^{\text{vOLE}}$ (described in Fig. A.4) twice, once with each server. To reduce the communication overhead for C_i, we adopt an asymmetric setting in $\Pi^{\text{InCom}}_{\text{Diho}}$ and decompose the computation into

$$\alpha^1 \circ \mathbf{x}^0 + \alpha^1 \circ \mathbf{x}^1 + \alpha^0 \circ (\mathbf{x}^0 + \mathbf{x}^1).$$

The first term $\alpha^1 \circ \mathbf{x}^0$ can be computed between the servers without involving C_i and is therefore moved to the preprocessing stage. The second term $\alpha^1 \circ \mathbf{x}^1$ can be computed locally by S_1. Thus, in the online stage, C_i participates only in a single invocation of $\mathcal{F}^{\text{vOLE}}$ with S_0 to compute the third term $\alpha^0 \circ \mathbf{x}'$. Finally, the servers perform the *consistency check* procedures (as described in Fig 6.2) to verify the correctness of the computed authentication MAC shares.

We note that $\Pi^{\text{InCom}}_{\text{Diho}}$ must be executed multiple times in the federated learning scenario. While adapting $\Pi^{\text{InCom}}_{\text{Diho}}$ from a single-client setting to a multi-client protocol, the $\mathcal{F}^{\text{vOLE}}$ instance between two servers only needs to be executed once, as it covers all individual executions of $\Pi^{\text{InCom}}_{\text{Diho}}$. Following the approach in [39], we discuss different corruption cases in Section 6.3.2 and show that the probability of successfully introducing errors into the MAC shares while still passing the consistency check is negligible.

6.2.2. Silent Select Protocol

Recap of Current L_2-Norm Checks. We begin by analyzing the current L_2-Norm-based defense mechanisms applied in state-of-the-art works. Both Flame [127] and RoFL [110] have shown that using a dynamic L_2-Norm bound β yields better filtering performance compared to a pre-defined bound. Current works such as Flame [127], RoFL [110] and Elsa [141] assume that this bound β can be publicly determined. As a result, either the servers must maintain a separate training dataset to compute the bound [110], or β is effectively reconstructed on the server side [141]. However, in the absence of a separate training dataset, β is computed from the clients' real-time gradient updates. This makes β an intermediate value derived from private inputs, and the reconstruction of β (or β^2) may leak sensitive information to the adversary. Meanwhile, in the server-client collusion setting, reconstructing the comparison result between $(L_2(\mathbf{u}_i))^2$ and β^2 also leaks information about β^2 to the adversary, as it reveals the range information of β^2. In this thesis, β is dynamically set to the mean of all clients' L_2-norms in each iteration, rather than the median as proposed in Flame [127] and RoFL [110], due to efficiency considerations. Parties may execute a secure median protocol to privately determine the median. We leave the choice of a dynamic L_2-norm bound as an future work.

[1] Since $x_{i,t} \in \mathbb{Z}_{2^{k+2s}}$, we do not denote it as $x'_{i,t}$.

Protocol Π^{SiSelect}

Private inputs: Servers hold $[\![\mathbf{x}]\!]$ and $[\![y]\!]_2$, where $\mathbf{x} \in \mathbb{Z}_{2^k}^t$ and $y \in \mathbb{Z}_2$. Additionally, $\mathcal{F}^{\text{TripGen}}$ is already initialized and S_j holds α_j.
Public inputs: Public parameters k, s.
Outputs: Servers output $[\![\mathbf{z}]\!]$, where $z_h = x_h$ if $y = 1$ and $z_h = 0$ otherwise.
Preprocessing:

1. $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$ sends $(\text{BitTripGen}, \mathsf{S}_j, \text{sid})$ to $\mathcal{F}^{\text{TripGen}}$, receives $([\![\mathbf{a}]\!]^j, [\![\mathbf{b}]\!]^j, [\![\mathbf{c}]\!]^j)$, where $b_h = b_h^0 + b_h^1 \bmod 2^k \in \{0,1\}$ and $c_h = a_h \cdot b_h \bmod 2^k$.
2. Let b_h^j and $m_{b_h}^j$ be S_j's share and MAC share of $\mathbf{b}[h]$. S_j defines $[\![\mathbf{b}']\!]_2^j$, where $b_h'^j \equiv b_h^j \pmod{2^{1+s}}$ and $m_{b_h'}^j \equiv m_{b_h}^j \pmod{2^{1+s}}$.

Protocol:

1. Servers run **Open** and **MAC check** (as defined in Fig. A.10) to reconstruct $\mathbf{e} = [\![\mathbf{x}]\!] - [\![\mathbf{a}]\!]$ and $\mathbf{f} = [\![y]\!]_2 + [\![\mathbf{b}']\!]_2$.
2. If $f_h = 0$, S_j sets $[\![z_h]\!]^j = [\![c_h]\!]^j + e_h \cdot [\![b_h]\!]^j \bmod 2^{k+s}$.
   If $f_h = 1$, S_j sets $[\![z_h]\!]^j = j \cdot e_h + [\![a_h]\!]^j - [\![c_h]\!]^j - e_h \cdot [\![b_h]\!]^j \bmod 2^{k+s}$

Figure 6.4.: Silent Select Protocol

Silent Select Protocol Π^{SiSelect}. In AlphaFL, we use the select functionality $\mathcal{F}_{\text{SS}}^{\text{Select}}$ to address the above issue by secretly filtering out outliers. We now introduce our silent select protocol Π^{SiSelect} as shown in Fig. 6.4 to efficiently implement $\mathcal{F}_{\text{SS}}^{\text{Select}}$.

a) Preprocessing: The core idea of Π^{SiSelect} to accelerate the online computation is to generate so called *Select Correlations* in the preprocessing phase. Specifically, servers generate multiplication triples $([\![\mathbf{a}]\!], [\![\mathbf{b}]\!], [\![\mathbf{c}]\!])$ by calling $\mathcal{F}^{\text{TripGen}}$ described in Fig. A.3, where $c_h = a_h \cdot b_h \bmod 2^k$ **and** $b_h = b_h^0 + b_h^1 \bmod 2^k \in \{0,1\}$.[2] Unlike traditional Beaver triples, we restrict b_h to be either 0 or 1 modulo 2^k. Note that the original Beaver triple generation protocol Π^{Triple} from [39] cannot be directly applied here, as parties sample $b_h^j \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$ and then determine a_h^j and c_h^j via the **Combine** procedure. As a result, the parties do not have control over the value of b_h, since the MAC shares of b_h are computed **afterwards**. To restrict the value of b_h, we let the parties first execute the random bit generation $\Pi^{\text{RanBitGen}}$ described in Fig. A.12, instead of sampling b_h randomly. The crucial step to ensure that $b_h \bmod 2^k \in \{0,1\}$ is to generate the MAC of b_h at the very beginning of Π^{Triple}, so that the adversary cannot manipulate the value of b_h. Then the parties use the authenticated value $[\![b_h]\!]$ to generate the triple $([\![a_h]\!], [\![b_h]\!], [\![c_h]\!])$ and execute the Arithmetic-to-Boolean (A2B) protocol from [46] to convert the arithmetic sharing $[\![b_h]\!]$ into a Boolean sharing $[\![b_h']\!]_2$. We define the resulting *select correlations* as follows:

[2] If we reconstruct b_h by computing $b_h = b_h^0 + b_h^1 \bmod 2^{k+s}$, b_h is not necessarily equal to 0 over $\mathbb{Z}_{2^{k+s}}$.

Definition 6.2.1 (*Select Correlations*). *Select correlations* are defined as $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket, \llbracket b' \rrbracket_2)$, such that

$$c = a \cdot b \bmod 2^k \quad \text{and} \quad b = b' \in \{0, 1\}.$$

b) Online Stage: During the online computation, the servers only have to reveal **e** and **f** in a single communication round. Then they can compute the shared output $[\mathbf{z}]$ locally without further interaction. The classic select protocol handles the A2B operation and multiplication in two rounds, resulting in $2t \cdot (k+s)+(s+1)$ bits of communication overhead (excluding the cost of the **MAC check**). In contrast, Π^{SiSelect} incurs a communication overhead of only $t \cdot (k + 2s + 1)$ bits, effectively reducing the communication cost of the classic protocol by approximately half.

6.3. Security Analysis

The high level idea is, if an adversary introduces an error into the MAC shares modulo 2^{k+s}, then in order to avoid detection it must find a compensating error modulo 2^{k+2s}. Doing so requires the adversary to correctly guess all bits of the honest party's MAC key share, which succeeds only with negligible probability.

6.3.1. Consistency Check Details in Π^{InCom}

Single Execution of Π^{InCom}. We now analyze the consistency check of protocol Π^{InCom} in the **honest-majority** setting. We observe that all MAC shares are correctly distributed, if C_i is honest. Thus, we discuss the case where C_i is corrupted by a malicious adversary $\mathcal{A}$ and both S_0 and S_1 are honest. Now different from the analysis provided by [39] where the adversary $\mathcal{A}$ does not know the MAC key shares of other honest parties, both shares α^0 and α^1 are received by $\mathcal{A}$ in Π^{InCom} at the initialization. The error that $\mathcal{A}$ can introduce to the h-th MAC is defined as

$$\rho_h = m_h - \alpha \cdot x'_h \bmod 2^{k+s}. \tag{6.1}$$

After taking random linear combinations with the vector **r** to compute the MAC of v, the reconstructed value $d = d^0 + d^1 \bmod 2^{k+2s}$ satisfies

$$d^0 + d^1 = \sum_{h=0}^{t-1} (\alpha \cdot x'_h + \rho_h) \cdot r_h + \alpha \cdot x_t + \rho_t \bmod 2^{k+2s}. \tag{6.2}$$

Now different from [39], $\mathcal{A}$ cannot introduce any error to v, since both honest S_0 and S_1 will reconstruct $v = v^0 + v^1 \bmod 2^{k+2s}$ honestly. This implies

$$\begin{aligned} 0 &= z^0 + z^1 \bmod 2^{k+2s} \\ &= d^0 + d^1 - \alpha \cdot (\sum_{h}^{t-1} x'_h \cdot r_h + x_t) \bmod 2^{k+2s} \\ &= \sum_{h=0}^{t-1} \rho_h \cdot r_h + \rho_t \bmod 2^{k+2s}. \end{aligned} \tag{6.3}$$

Claim 6.3.1. Suppose that there is at least one non-zero component $\rho_h \bmod 2^{k+s}$, then the probability of passing the check is no more than 2^{-s}.

Proof. Without loss of generality, we suppose $\rho_0 \neq 0 \bmod 2^{k+s}$, then we have

$$\rho_0 \cdot r_0 = \underbrace{\sum_{h=1}^{t-1} \rho_h \cdot r_h + \rho_t}_{S} \bmod 2^{k+2s}. \tag{6.4}$$

Let 2^v be the largest power of two dividing ρ_0, we know that $v < k + s$, since $\rho_0 \neq 0 \bmod 2^{k+s}$. Therefore, we know that $\frac{\rho_0}{2^v}$ is odd and has multiplicative inverse modulo $k + 2s - v$. We have

$$\begin{aligned} r_0 \cdot \frac{\rho_0}{2^v} &= \frac{S}{2^v} \bmod 2^{k+2s-v} \\ r_0 &= \frac{S}{2^v} \cdot (\frac{\rho_0}{2^v})^{-1} \bmod 2^{k+2s-v}. \end{aligned} \tag{6.5}$$

Since $s < k + 2s - v$, r_0 is completely determined. By definition r_0 is randomly chosen at 2^s, we conclude that this particular event happens with probability 2^{-s}. □

Multiple Executions of Π^{InCom} with Clients. We then analyze the case where Π^{InCom} is executed with multiple clients C_i. Similar to the above proof, we only discuss the case where a subset of clients $C_c \subseteq \{C_0, ..., C_{n-1}\}$ are corrupted. Let q denote the number of corrupted clients. Since both servers only perform the consistency check once after receiving all MAC shares, we can extend Equation 6.4 to

$$\rho_0 \cdot r_0 = \underbrace{\sum_{h=1}^{qt-1} \rho_h \cdot r_h + \sum_{h=qt}^{qt+q-1} \rho_h}_{S} \bmod 2^{k+2s}. \tag{6.6}$$

The rest of the proof follows as before, and we therefore conclude that Claim 6.3.1 still holds.

6.3.2. Consistency Check Details in $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$

Single Execution of $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$. We then analyze the consistency check of protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ in the **dishonest-majority** setting. Similar to the proofs provided in [39], we first consider that the adversary does not send any (guess) message to $\mathcal{F}^{\mathsf{vOLE}}$. Let $\hat{\alpha}^j$ and $\hat{\mathbf{x}}^j$ be the actual value (and vector) used by a corrupt P_c in the $\mathcal{F}^{\mathsf{vOLE}}$ instance. We define the correct global MAC key share as α^j, which is the value used by P_c during the first execution of $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$. Additionally, we define the correct value $\mathbf{x}^j$ as the value obtained either from $\mathcal{F}^{\mathsf{CR}}$ or directly from C_i. We then define errors as

$$\gamma^{1-j} = \hat{\alpha}^j - \alpha^j \quad \text{and} \quad \boldsymbol{\delta}^{1-j} = \hat{\mathbf{x}}^j - \mathbf{x}^j.$$

For convenience, we define the errors between two corrupted parties to be zero. Note that a corrupted C_i can introduce an error $\boldsymbol{\delta} = \hat{\mathbf{x}}' - \mathbf{x}^0 - \mathbf{x}^1$ to the protocol. Without loss of generality, we always define the error as $\boldsymbol{\delta}^1 = \hat{\mathbf{x}}^0 - \mathbf{x}^0$, provided that none of the servers are corrupted. We observe following corruption cases:

1. C_i is corrupted: $\mathcal{A}$ can provide an incorrect $\hat{\mathbf{x}}$, thereby introducing an error $\boldsymbol{\delta}^1 = \hat{\mathbf{x}}^0 - \mathbf{x}^0$ while executing $\mathcal{F}^{\mathsf{vOLE}}$ with S_0. It can also send an inconsistent $\mathbf{b}^1$ to S_1.
2. S_0 is corrupted: $\mathcal{A}$ can introduce both types of errors by providing incorrect $\hat{\mathbf{x}}^0$ and $\hat{\alpha}^0$ while executing $\mathcal{F}^{\mathsf{vOLE}}$ with S_1 and C_i.
3. S_1 is corrupted: $\mathcal{A}$ cannot introduce any error during the computation. It only initialize an $\mathcal{F}^{\mathsf{vOLE}}$ instance with S_0 with α^1. And this $\mathcal{F}^{\mathsf{vOLE}}$ instance will be executed only once during the preprocessing stage.
4. S_0 and C_i are corrupted: $\mathcal{A}$ cannot introduce any error during the computation. During the first $\mathcal{F}^{\mathsf{vOLE}}$ computation between S_0 and S_1, there will be no inconsistency attributed to $\mathcal{A}$, since we define the errors between two corrupted parties to be zero. Due to the same reason, $\mathcal{A}$ cannot introduce an error $\boldsymbol{\delta}^1 = \hat{\mathbf{x}}^0 - \mathbf{x}^0$ by providing an "incorrect" $\mathbf{x}^1$ to S_1.
5. S_1 and C_i are corrupted: $\mathcal{A}$ cannot introduce any error during the computation, similar to the case where only S_1 is corrupted.

We now discuss different cases as explained above:

Case 1: Similar to Π^{InCom}, a corrupted C_i cannot affect the *consistency check*. However, the adversary can send an incorrect $\mathbf{b}^1$ to S_1 to compensate for the errors it introduced during the execution of an $\mathcal{F}^{\mathsf{vOLE}}$ instance with S_0. The sum of the MAC shares on x'_h is then given by

$$m_h = \alpha \cdot x'_h + \alpha^0 \cdot \delta_h + \rho_h \bmod 2^{k+2s}.$$

After taking random linear combinations with the vector $\mathbf{r}$ to compute the MAC of v, the reconstructed value $d = d^0 + d^1 \bmod 2^{k+2s}$ satisfies

$$\begin{aligned} d^0 + d^1 &= \sum_{h=0}^{t-1} (\alpha \cdot x'_h + \alpha^0 \cdot \delta_h + \rho_h) \cdot r_h + \alpha \cdot x_t + \alpha^0 \cdot \delta_t + \rho_t \bmod 2^{k+2s} \\ &= \alpha \cdot (\sum_{h}^{t-1} x'_h \cdot r_h + x_t) + \alpha^0 \cdot (\sum_{h=0}^{t-1} \delta_h \cdot r_h + \delta_t) + \sum_{h=0}^{t-1} \rho_h \cdot r_h + \rho_t \bmod 2^{k+2s}. \end{aligned}$$

Claim 6.3.2. Suppose that C_i is the only corrupted party. Assuming that $\boldsymbol{\delta}^j$ is non-zero modulo 2^{k+s} in at least one component for some $j \notin c$. Then, the probability of passing the check is no more than $2^{-s+\log(s+1)}$.

Proof. Then passing the check implies

$$\begin{aligned} 0 &= z^0 + z^1 \bmod 2^{k+2s} \\ &= d^0 + d^1 - \alpha \cdot (\sum_{h}^{t-1} x'_h \cdot r_h + x'_t) \bmod 2^{k+2s} \\ &= \alpha^0 \cdot (\sum_{h=0}^{t-1} \delta_h \cdot r_h + \delta_t) + \underbrace{\sum_{h=0}^{t-1} \rho_h \cdot r_h + \rho_t}_{e} \bmod 2^{k+2s}. \end{aligned} \tag{6.7}$$

Using the Lemma A.5.1 provided in [39], we set $r = k + s$, $m = s$, $\ell = k + 2s$, $\delta_0 = \delta_t$ (and $y = -e$). The above only holds with

$$\Pr[\mathcal{A} \text{ passes check}] \leq 2^{-s+\log(s+1)}.$$

□

Case 2: Let c denote the set of indices corresponding to corrupted parties (servers). The errors in the sum of MAC shares are summarized as

$$\begin{aligned} \mathbf{m}^0 + \mathbf{m}^1 &= \alpha^1 \circ \mathbf{x}^1 + \sum_j \mathbf{a}^j - \mathbf{b}^j + \sum_{j \notin c} \mathbf{x}^j \circ \gamma^j + \sum_{j \notin c} \alpha^j \circ \boldsymbol{\delta}^j \bmod 2^{k+2s} \\ &= \alpha \circ \mathbf{x}' + \sum_{j \notin c} \mathbf{x}^j \circ \gamma^j + \sum_{j \notin c} \alpha^j \circ \boldsymbol{\delta}^j \bmod 2^{k+2s}. \end{aligned} \tag{6.8}$$

To pass the consistency check, the adversary may first open v to a (possibly incorrect) value in $\Pi_{\mathsf{Diho}}^{\mathsf{InCom}}$, say $\hat{v} = v + \epsilon$. Then the adversary must come up with the same error term $e \in \mathbb{Z}_{2^{k+2s}}$ as in [39] such that

$$\begin{aligned} -e &= \alpha \cdot \epsilon + \sum_{j \notin c} (\mathbf{x}^j \cdot \mathbf{r} + x_t^j) \cdot \gamma^j + \sum_{j \notin c} \alpha^j \cdot (\boldsymbol{\delta}^j \cdot \mathbf{r} + \delta_t^j) \bmod 2^{k+2s} \\ \Leftrightarrow -e - \sum_{j \in c} \alpha^j \cdot \epsilon &= \sum_{j \notin c} u^j \cdot \gamma^j + \sum_{j \notin c} \alpha^j \cdot (\boldsymbol{\delta}^j \cdot \mathbf{r} + \delta_t^j + \epsilon) \bmod 2^{k+2s}. \end{aligned} \tag{6.9}$$

We end up with the same error analysis as in [39]. Thus, we directly derive two claims (with a slight modification) from [39]:

Claim 6.3.3. If at least one $\gamma^j \neq 0$ where $j \notin c$, then the probability of passing the check is no more than 2^{-s}.

Proof. Let j be the index of S_j, such that $\gamma^j \neq 0$. In the two-server setting, for a single server S_j, the adversary can introduce at most one nonzero value γ^j in the execution of $\Pi^{\mathsf{lnCom}}_{\mathsf{Diho}}$. Therefore, we have $\gamma^j \leq 2^s$. Note that the distribution of u^j is uniform in $\mathbb{Z}_{2^{k+2s}}$ and independent of all other terms, due to the extra random mask x_t^j, so we can rewrite Equation 6.9 as

$$e' = u^j \cdot \gamma^j \bmod 2^{k+2s}.$$

Let 2^v be the largest power of two dividing γ^j, then we have

$$u^j \cdot \frac{\gamma^j}{2^v} = \frac{e'}{2^v} \bmod 2^{k+2s-v}$$
$$u^j = \frac{e'}{2^v} \cdot (\frac{\gamma^j}{2^v})^{-1} \bmod 2^{k+2s-v}.$$

Since $v < s$ and $k + 2s \geq 2s$, this holds with the probability at most $2^{-k-2s+s} \leq 2^{-s}$. □

Claim 6.3.4. Suppose $\gamma^j = 0$ for all $j \notin c$, and $\boldsymbol{\delta}^j$ is non-zero modulo 2^{k+s} in at least one component for some $j \notin c$. Then, the probability of passing the check is no more than $2^{-s+\log(s+1)}$.

Proof. Passing the check implies that

$$-e - \sum_{j \in c} \alpha^j \cdot \epsilon = \sum_{j \notin c} \alpha^j \cdot (\boldsymbol{\delta}^j \cdot \mathbf{r} + \delta_t^j + \epsilon) \bmod 2^{k+2s}.$$

We consider the case where there is only a single honest server S_j and $\boldsymbol{\delta}^j$ is non-zero modulo 2^{k+s}. The adversary must come up with an error term such that

$$-e' = \alpha^j \cdot (\boldsymbol{\delta}^j \cdot \mathbf{r} + \delta_t^j + \epsilon) \bmod 2^{k+2s}.$$

Using the Lemma A.5.1 provided in [39], we set $r = k + s$, $m = s$, $\ell = k + 2s$, and $\delta_0 = \delta_t^j + \epsilon$. The above only holds with

$$\Pr[\mathcal{A} \text{ passes check}] \leq 2^{-s+\log(s+1)}$$

□

Case 3, 4 and 5: Since the adversary is not capable to introduce any error during the computation, we skip these cases.

Multiple Execution of $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ with Clients. We then analyze the case where $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ is computed with multiple clients. Let i be the index of C_i, we have $i \in [n]$. Let q be the number of clients whose computations have been compromised by the adversary.

Case 1: We adapt Claim 6.3.2 as follows:

Claim 6.3.5. Suppose that a set of clients $C_c \subseteq \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$ is corrupted. Assuming that $\boldsymbol{\delta}_i^j$ is non-zero modulo 2^{k+s} in at least one component for some $j \notin c$. Then, the probability of passing the check is no more than $2^{-s+\log(s+1)}$.

Proof. We can extend Equation 6.7 to

$$0 = \alpha^0 \cdot (\underbrace{\sum_{h=0}^{qt-1} \delta_h \cdot r_h}_{\boldsymbol{\theta}\cdot\boldsymbol{\chi}} + \underbrace{\sum_{h=qt}^{qt+q-1} \delta_h}_{\theta_{qt}}) + \underbrace{\sum_{h=0}^{qt-1} \rho_h \cdot r_h + \sum_{h=qt}^{qt+q-1} \rho_h}_{e} \bmod 2^{k+2s},$$

where $\boldsymbol{\theta} = (\theta_0, ..., \theta_{qt-1})$ and $\boldsymbol{\chi} = (\chi_0, ..., \chi_{qt-1})$.

Using the Lemma A.5.1 provided in [39], we set $r = k + s$, $m = s$, $\ell = k + 2s$, $\delta_0 = \theta_{qt}$ (and $y = -e$). The above only holds with

$$\Pr[\mathcal{A} \text{ passes check}] \leq 2^{-s+\log(s+1)}.$$

□

Case 2: Now we can extend Equation 6.9 to

$$-e - \sum_{j\in c} \alpha^j \cdot \epsilon = \sum_{j\notin c} \left(\sum_{i=0}^{q-1} u_i^j \cdot \gamma_i^j \right) + \sum_{j\notin c} \alpha^j \cdot \left[\sum_{i=0}^{q-1} \left(\boldsymbol{\delta}_i^j \cdot \mathbf{r}_i + \delta_{i,t}^j \right) + \epsilon \right] \bmod 2^{k+2s}. \tag{6.10}$$

We adapt Claim 6.3.3 and Claim 6.3.4 as follows:

Claim 6.3.6. If at least one $\gamma_i^j \neq 0$ where $j \notin c$, then the probability of passing the check is no more than 2^{-s}.

Proof. Let j be the index of S_j, such that $\gamma_i^j \neq 0$. In the two-server setting, for a single server S_j, the adversary can introduce at most one nonzero value γ_i^j in the execution of $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ (with C_i). Therefore, we have $\gamma_i^j \leq 2^s$. Note that the distribution of u_i^j is uniform in $\mathbb{Z}_{2^{k+2s}}$ and independent of all other terms, due to the extra random mask $x_{i,t}^j$, so we can rewrite Equation 6.10 as

$$e' = u_i^j \cdot \gamma_i^j \bmod 2^{k+2s}.$$

Let 2^v be the largest power of two dividing γ_i^j, then we have

$$u_i^j \cdot \frac{\gamma_i^j}{2^v} = \frac{e'}{2^v} \bmod 2^{k+2s-v}$$
$$u_i^j = \frac{e'}{2^v} \cdot (\frac{\gamma_i^j}{2^v})^{-1} \bmod 2^{k+2s-v}.$$

Since $v < s$ and $k + 2s \geq 2s$, this holds with the probability at most $2^{-k-2s+s} \leq 2^{-s}$. □

Claim 6.3.7. Suppose $\gamma_i^j = 0$ for all $j \notin c$ and $i \in [n]$, and $\boldsymbol{\delta}^j$ is non-zero modulo 2^{k+s} in at least one component for some $j \notin c$. Then, the probability of passing the check is no more than $2^{-s+\log(s+1)}$.

Proof. Passing the check implies that

$$-e - \sum_{j \in c} \alpha^j \cdot \epsilon = \sum_{j \notin c} \alpha^j \cdot \left[\sum_{i=0}^{q-1} \left(\boldsymbol{\delta}_i^j \cdot \mathbf{r}_i + \delta_{i,t}^j \right) + \epsilon \right] \bmod 2^{k+2s}.$$

We consider the case in which there is only a single honest server S_j and $\boldsymbol{\delta}^j$ is non-zero modulo 2^{k+s}. The adversary must come up with an error term such that

$$-e' = \alpha^j \cdot \left[\sum_{i=0}^{q-1} \left(\boldsymbol{\delta}_i^j \cdot \mathbf{r}_i + \delta_{i,t}^j \right) + \epsilon \right] \bmod 2^{k+2s}$$
$$= \alpha^j \cdot \Bigg(\underbrace{\sum_{i=0}^{q-1} \boldsymbol{\delta}_i^j \cdot \mathbf{r}_i}_{\boldsymbol{\theta}^j \cdot \boldsymbol{\chi}} + \underbrace{\sum_{i=0}^{q-1} \delta_{i,t}^j + \epsilon}_{\theta_{qt}^j + \epsilon} \Bigg) \bmod 2^{k+2s},$$

where $\boldsymbol{\theta}^j = (\theta_0^j, ..., \theta_{qt-1}^j)$ and $\boldsymbol{\chi} = (\chi_0, ..., \chi_{qt-1})$.

Using the Lemma A.5.1 provided in [39], we set $r = k + s$, $m = s$, $\ell = k + 2s$, and $\delta_0 = \theta_{qt}^j + \epsilon$.

$$\Pr[\mathcal{A} \text{ passes check}] \leq 2^{-s+\log(s+1)}.$$

□

Handling Guess Queries. First, we note that Claim 6.3.6 holds because u_j^i is independent of α^j, and its uniform distribution ensures that the probability of successfully introducing an error term γ_i^j is negligible. By adopting the same proof (to **handle key queries**) provided in [39], we can show that both Claim 6.3.5 and Claim 6.3.7 hold, even if the adversary makes some successful queries to $\mathcal{F}^{\text{vOLE}}$ instances using the (guess) command.

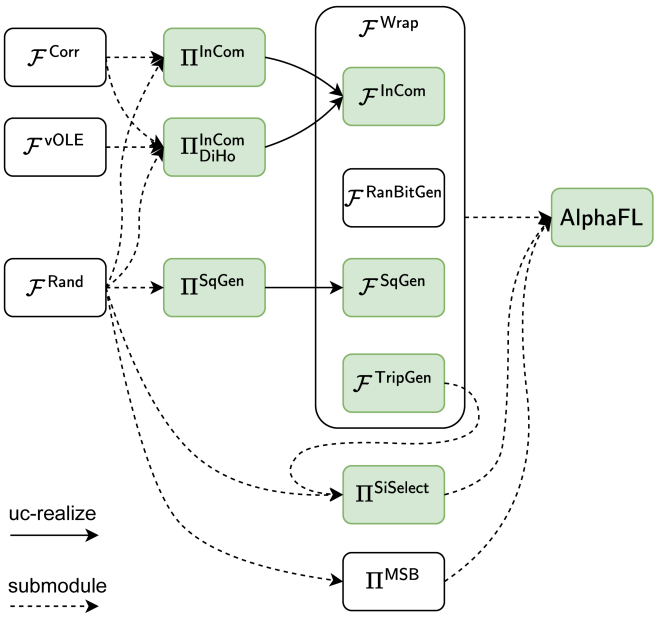


Figure 6.5.: Relationship between protocols and functionalities. Discussed components in this work are highlighted in green. $\mathcal{F}^{\text{Rand}}$ is a submodule of all protocols due to consistency check or Open and MAC check procedure, respectively.

6.3.3. A Subtlety of Modeling Functionalities for SPD$\mathbb{Z}_{2^k}$

While modeling the ideal functionalities for the SPD$\mathbb{Z}_{2^k}$ scheme, we identified a subtle issue where in certain cases, the functionality cannot extract the global MAC key α solely from the received input shares. In $\mathcal{F}^{\text{InCom}}$ (and $\mathcal{F}^{\text{MAC}}$ described in Fig. A.2), α is chosen at the initialization stage by the functionality. We take a two-party B2A functionality $\mathcal{F}^{\text{B2A}}$ (or any functionality $\mathcal{F}$) without the initialization stage as an example, and we show that the functionality may fail to extract α in certain cases. Suppose that $\mathcal{F}^{\text{B2A}}$ receives $[\![x]\!]_2$ as input, it must extract α by computing $\alpha = (m_x^0 + m_x^1) \cdot (x^0 + x^1)^{-1} \bmod 2^{1+s}$, where $(x^j, m_x^j) \in (\mathbb{Z}_{2^{1+s}}, \mathbb{Z}_{2^{1+s}})$. However, if $x' = x^0 + x^1 \bmod 2^{1+s}$ does not have a multiplicative inverse in $\mathbb{Z}_{2^{1+s}}$, then $\mathcal{F}^{\text{B2A}}$ may identify a set of global MAC key shares $\{\alpha_i\}$ such that each element satisfies $\alpha_i \cdot x' \equiv m_x^0 + m_x^1 \pmod{2^{1+s}}$. In [39, 46], this problem is addressed by consolidating all randomness generation functionalities into a single preprocessing functionality $\mathcal{F}^{\text{Pre}}$, which manages the global MAC key generation. In AlphaFL, we build a wrapper functionality $\mathcal{F}^{\text{Wrap}}$ described in Fig. A.7, which accepts commands as defined in $\mathcal{F}^{\text{InCom}}$, $\mathcal{F}^{\text{SqGen}}$ and $\mathcal{F}^{\text{TripGen}}$ (formally defined in Fig. 6.6, Fig. A.6 and Fig. A.3, respectively). We also extend $\mathcal{F}^{\text{Wrap}}$ to include the functionality instructions of $\mathcal{F}^{\text{RanBitGen}}$ (described in Fig. A.5). We show an overview of different protocols and functionalities in Fig. 6.5.

Functionality $\mathcal{F}^{\mathsf{InCom}}$

Initialize: Upon receiving $(\mathsf{Init}, \mathsf{C}_i, \mathsf{sid})$ for all $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$ and $(\mathsf{Init}, \mathsf{S}_j, \mathsf{sid})$ for all $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$:

1. If S_c is corrupted, wait to receive $\alpha^c \in \mathbb{Z}_{2^s}$ from the adversary. Choose $\alpha^{c-1} \in \mathbb{Z}_{2^s}$.

⋆2. If C_i is corrupted, wait to receive $(\alpha^0, \alpha^1) \in (\mathbb{Z}_{2^s}, \mathbb{Z}_{2^s})$ from the adversary. Ignore subsequent messages.

3. Store $\alpha = \alpha^c + \alpha^{c-1} \bmod \mathbb{Z}_{2^{k+s}}$.

4. Send α^j to S_j

⋆5. Send (α^0, α^1) to C_i.

Macro $\mathsf{MacGen}(\mathbf{x}')$ (internal subroutine only):

1. Compute $\mathbf{m} = \alpha \circ \mathbf{x}' \bmod 2^{k+s}$.

2. If $\mathbf{m}^c$ is received from the adversary, then set $\mathbf{m}^{c-1} = \mathbf{m} - \mathbf{m}^c \bmod 2^{k+s}$.

3. Otherwise, set $\mathbf{m}^0 \xleftarrow{\$} \mathbb{Z}^t_{2^{k+s}}$ and $\mathbf{m}^1 = \mathbf{m} - \mathbf{m}^0 \bmod 2^{k+s}$.

4. Output $(\mathbf{m}^0, \mathbf{m}^1)$.

InCom: Upon receiving $(\mathsf{InCom}, \mathsf{C}_i, \mathsf{sid})$ for all $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$ and $(\mathsf{InCom}, \mathsf{S}_j, \mathsf{sid})$ for all $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$

1. If only S_c is corrupted, then for each C_i wait to receive $(\mathbf{x}^c_i, \mathbf{m}^c_i) \in \mathbb{Z}^t_{2^{k+s}} \times \mathbb{Z}^t_{2^{k+s}}$ from the adversary and $(\mathbf{x}_i, \mathsf{C}_i, \mathsf{sid})$ from C_i, where $\mathbf{x}_i \in \mathbb{Z}^t_{2^k}$. Choose $\mathbf{x}^{c-1}_i \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, such that $\mathbf{x}_i = \mathbf{x}^c_i + \mathbf{x}^{c-1}_i \bmod 2^k$.

2. If C_i is corrupted (individually or simultaneously), then for each corrupted C_i wait to receive $(\mathbf{x}^0_i, \mathbf{x}^1_i)$ from the adversary.

3. For all C_i, send $\mathbf{x}^j_i$ to S_j.

∗4. Wait for the adversary to send messages $(\mathsf{guess}, \mathsf{S}_j, B_j)$ for $j \notin c$, where B_j efficiently describes a subset of $\{0,1\}^s$. If C_i is the only corrupted party, ignore queries if $\mathsf{S}_j \neq \mathsf{S}_0$. If $\alpha^j \in B_j$, send success to the adversary. Otherwise abort.

5. For all C_i, compute $\mathbf{x}'_i = \mathbf{x}^0_i + \mathbf{x}^1_i \bmod 2^{k+s}$. Run $\mathsf{MACGen}(\mathbf{x}'_i)$, send $\mathbf{m}^j_i$ to S_j.

Figure 6.6.: Input Commitment Functionality

6.3.4. Security of Π^{InCom}

We formally define the input commitment functionality $\mathcal{F}^{\mathsf{InCom}}$ in Fig. 6.6. We use "⋆" to indicate that a step is only considered in the honest-majority setting. We use "∗" to indicate that a step is only considered in the dishonest-majority setting.

Theorem 6.3.8. Protocol Π^{InCom} shown in Fig. 6.2 uc-realizes $\mathcal{F}^{\mathsf{InCom}}$ described in Fig. 6.6 in the $\mathcal{F}^{\mathsf{CR,glo}}$, $\mathcal{F}^{\mathsf{CR}}$, $\mathcal{F}^{\mathsf{Rand}}$-hybrid model, in the presence of a malicious adversary, who

can corrupt either a subset of clients $C_c \subseteq \{C_0, ..., C_{n-1}\}$ or a server $S_j \in \{S_0, S_1\}$, with static corruption.

We defer the detailed proof to Appendix A.5.2.

6.3.5. Security of $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$

Theorem 6.3.9. Protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ shown in Fig. 6.3 uc-realizes $\mathcal{F}^{\mathsf{InCom}}$ described in Fig. 6.6 in the $\mathcal{F}^{\mathsf{CR}}$, $\mathcal{F}^{\mathsf{Rand}}$, $\mathcal{F}^{\mathsf{vOLE}}$-hybrid model, in the presence of a malicious adversary, who can corrupt either a subset of clients or a server S_j or a subset of clients together with a server S_j, with static corruption.

We defer the detailed proof to Appendix A.5.3.

6.4. Federated Learning with Malicious^2 Security

We now build AlphaFL with proposed components above, and we show protocol details in Fig. 6.7 and Fig. 6.8. We note that in AlphaFL, we set β to the mean of all clients' L_2-Norms, rather than the median as suggested in Flame [127] and RoFL [110]. Different from previous works [2, 37, 110, 141], we additionally let parties execute a silent select protocol Π^{SiSelect} [75] to protect the dynamically chosen L_2-norm bound. We consider the selection of a dynamic L_2-norm bound to be orthogonal to this work.

6.5. Evaluation

We implement AlphaFL in two parts, both based on MP-SPDZ `v0.3.9` [84]. The core secure aggregation building blocks are implemented in `Python` using MP-SPDZ's high-level interface. The novel input commitment protocols are implemented in `C++` via MP-SPDZ's lowest-level interface, as the required functionalities are not exposed at higher levels. The full source code is publicly available at `https://github.com/Barkhausen-Institut/AlphaFL`.

6.5.1. Experiment Setup

Testbed Environment. All evaluations were conducted on an `Ubuntu 24.04.1 LTS` virtual machine provisioned with 48 vCPUs and 128 GB of RAM, hosted on a workstation equipped with two `Intel(R) Xeon(R) Gold 5317` CPUs. All client and server components were executed as separate processes. Similar to previous work, we consider only one single

AlphaFL (Part 1)

Parameters: At current iteration q, let n denote the number of clients, t denotes the size of gradient vectors. Let w be the parameter for L_∞, τ be the minimal valid inputs required to proceed the aggregation protocol. β is the L_2-Norm bound.
Outputs: t-valued global aggregate vector.
Initialize: $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$ sends $(\mathsf{Init}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, receives back α^j.
Preprocessing: For each client $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$:

1. S_j sends $(\mathsf{RanBitGen}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, receives $[\![\mathbf{b}]\!]^j$, where $\mathbf{b} \bmod 2^k \in \{0,1\}^{w*t}$. Let b_i^j and $m_{b_i}^j$ be S_j's share and MAC share of $\mathbf{b}[i]$. S_j defines $[\![\mathbf{b'}]\!]_2^j$, where $b_i'^j \equiv b_i^j \pmod{2^{1+s}}$ and $m_{b_i'}^j \equiv m_{b_i}^j \pmod{2^{1+s}}$.
2. S_j sends $(\mathsf{RanBitGen}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, receives $[\![\mathbf{p}]\!]^j$, where $\mathbf{p} \bmod 2^k \in \{0,1\}^n$. Let p_i^j and $m_{p_i}^j$ be S_j's share and MAC share of $\mathbf{p}[i]$. S_j defines $[\![\mathbf{p'}]\!]_2^j$, where $p_i'^j \equiv p_i^j \pmod{2^{1+s}}$ and $m_{p_i'}^j \equiv m_{p_i}^j \pmod{2^{1+s}}$.
3. S_j sends $(\mathsf{SqCoGen}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, receives $([\![\mathbf{a}]\!]^j, [\![\mathbf{d}]\!]^j)$, where $(\mathbf{a}, \mathbf{d}) \in (\mathbb{Z}_{2^{k+s}}^t, \mathbb{Z}_{2^{k+s}}^t)$ and $d_h = a_h \cdot a_h \bmod 2^k$.
4. S_j sends $(\mathsf{SqCoGen}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, receives $([\![\delta]\!]^j, [\![\gamma]\!]^j)$, where $(\delta, \gamma) \in (\mathbb{Z}_{2^{k+s}}, \mathbb{Z}_{2^{k+s}})$ and $\gamma = \delta \cdot \delta \bmod 2^k$.

Input Commitment: For each client C_i:

1. C_i locally computes the gradient update $\mathbf{u}_i = (u_0, ..., u_{t-1})$, where u_h can be decomposed bit-wise as $(u_{h,0}, ..., u_{h,w-1}) \in \mathbb{Z}_2^w$.
2. S_0, S_1 and C_i send $(\mathsf{InCom}, \cdot, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, then C_i sends $(\mathbf{u}_i, \mathsf{C}_i, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, where $\mathbf{u}_i \in \mathbb{Z}_2^{w*t}$. Servers receive $[\![\mathbf{u}_i]\!]_2$.

L_∞-Norm and B2A: For each C_i:

1. Servers run the **Open** phase of **BatchCheck** (as defined in Fig. A.10) to reconstruct $\mathbf{c} = [\![\mathbf{u}_i]\!]_2 + [\![\mathbf{b'}]\!]_2$, where $\mathbf{c} \in \mathbb{Z}_2^{w*t}$.
2. For $h \in \{0, ..., t-1\}$, let $([\![u_{h,0}]\!], ..., [\![u_{h,w-1}]\!])$ be the arithmetic shares of bits in u_h. Servers locally compute $[\![u_{h,i}]\!] = c_{w\cdot h+i} + [\![b_{w\cdot h+i}]\!] - 2 \cdot c_{w\cdot h+i} \cdot [\![b_{w\cdot h+i}]\!]$.
3. Servers finally compute $[\![u_h]\!] = \sum_{i=0}^{w-2} 2^i \cdot [\![u_{h,i}]\!] + \sum_{i=w}^{k-1} 2^i \cdot [\![u_{h,w}]\!]$.

L_2-Norm Computation: For each client C_i:

1. For $h \in \{0, ..., t-1\}$, servers run the **Open** phase of **BatchCheck** to reconstruct $f_h = [\![u_h]\!] - [\![a_h]\!]$.
2. S_0 computes $[\![v]\!]^0 = \sum_{h=0}^{t-1} [\![d_h]\!]^0 + 2f_h \cdot [\![a_h]\!]^0 - f_h \cdot f_h \bmod 2^{k+s}$. S_1 computes $[\![v]\!]^1 = \sum_{h=0}^{t-1} [\![d_h]\!]^1 + 2f_h \cdot [\![a_h]\!]^1 \bmod 2^{k+s}$.

Figure 6.7.: Maliciously Secure Aggregation Protocol in AlphaFL (Part 1)

network setting, where the network environment is configured using the `tc` tool [3] to

[3] `https://man7.org/linux/man-pages/man8/tc.8.html`

AlphaFL (Part 2)

L_2-Norm Check:

1. Let v_i denote the L_2-Norm of $\mathbf{u_i}$ for C_i. Servers compute $[\![\beta]\!] = \frac{\sum_i [\![v_i]\!]}{n}$ for $i \in \{0, ..., n-1\}$ and $[\![\beta^2]\!]$ using $([\![\delta]\!], [\![\gamma]\!])$ as above.
2. Then for each client C_i, servers compute $[\![y_i]\!] = [\![v_i]\!] - [\![\beta^2]\!]$.
3. Servers run Π^{MSB} with input $[\![y_i]\!]$ to extract the authenticated shared sign bit $[\![s_i]\!]_2$ of y_i ($s_i = 1$ indicates that $v_i < \beta^2$).

Aggregation:

1. For each client C_i, servers run Π^{SiSelect} with $[\![\mathbf{u}_i]\!]$ and $[\![s_i]\!]_2$ as inputs, receive $[\![\mathbf{z}_i]\!]$ as output.
2. Servers run the **Open** phase of **BatchCheck** to reconstruct $e_i = [\![s_i]\!]_2 + [\![p'_i]\!]_2$. Servers compute $[\![s_i]\!] = e_i + [\![p_i]\!] - 2 \cdot e_i \cdot [\![p_i]\!]$.
3. Servers run the **Open** phase of **BatchCheck** to reconstruct $\tau' = \sum_{i=0}^{n-1} [\![s_i]\!]$. If $\tau' < \tau$, servers abort the computation.
4. Otherwise, servers compute $[\![\mathcal{U}_q]\!] = \frac{1}{\tau'} \cdot \sum_{i=0}^{n-1} [\![\mathbf{z}_i]\!]$, where $\mathbf{z}_i = \mathbf{0}$ if $s_i = 0$ and $\mathbf{z}_i = \mathbf{u}_i$ otherwise .

MAC Check:

1. Servers run the **BatchCheck** to check the MACs on values that have been so far opened.
2. If servers do not abort, they open and check the MAC on $[\![\mathcal{U}_q]\!]$ using the **SingleCheck** procedure (as defined in Fig. A.11).

Output: Servers send $\mathcal{U}_q$ to all clients.

Figure 6.8.: Maliciously Secure Aggregation Protocol in AlphaFL (Part 2)

simulate a round-trip latency of `1 ms` and a bandwidth of `10 Gbps`. Gradient updates are represented using 32-bit values. For secure aggregation and L_2-Norm computation, we operate over a 64-bit ring with parameters $w = 32$, $k = 64$ and $s = 63$. The choice of $s = 63$ (instead of $s = 64$) facilitates improved memory alignment in our input commitment implementation.

The end-to-end runtime of AlphaFL consists of two main phases. The first phase is the input commitment between the clients and servers. The second phase is a secure aggregation between the servers, including L_∞-Norm and L_2-Norm checks. We implement and evaluate the protocol Π^{InCom} and $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ separately. The corresponding end-to-end runtimes are denoted as AlphaFL-Ho and AlphaFL-DiHo, respectively.

Table 6.1.: End-to-End Runtime and Total Communication Compared with the Single-Server Framework RoFL

#Params	Alpha-Ho	Alpha-DiHo	RoFL
	Runtime (second)		
62k CIFAR10-S	0.58	7.76	1,848
273k CIFAR10-L	2.88	32.78	14,107
818k SHAKESPEARE	6.31	95.75	28,345[4]
	Total Data Sent (MB)		
62k CIFAR10-S	200	8,201	68
273k CIFAR10-L	887	36,297	301
818k SHAKESPEARE	2,644	108,169	898

6.5.2. Comparison against a Single-Server Framework

Baseline. In the single-server setting, we consider RoFL [110] at commit `c1a0c13`. The total runtime is calculated by summing the recorded gradient encryption time, proof generation time, and communication time at a single client, along with the aggregation and proof verification time at the server. The total data sent is the sum of the data transmitted by all clients.

Parameter Sizes. We consider three models with different parameter sizes, each evaluated with a distinct dataset:

- `LeNet5` [96] trained on `CIFAR10-S`;
- `ResNet18` [67] trained on the `CIFAR10` [91];
- `LSTM` [69] trained on the `Shakespeare` [27].

We only let $n = 4$ clients connect to the server, as RoFL [110] crashes with a higher number of clients during our evaluation.

Communication Comparison. We show the communication overhead for both frameworks in Table 6.1. In the single-server setting, RoFL [110] benefits from a simpler server-client connection, resulting in lower traffic volume. In the two-server setting, frameworks such as AlphaFL require each client to communicate with both servers. Moreover, the two servers must also communicate with each other to execute MPC protocols. As a result, the total communication cost of AlphaFL is more than twice that of RoFL.

[4] This value is approximated due to server-side logging failure. The actual time should be longer.

Table 6.2.: End-to-end runtime (seconds) comparison in a two-server setting. Parenthesized value is the time consumed by vOLE. N/A means that the program aborted.

#Clients	#Params	Alpha-Ho	Alpha-DiHo	Elsa	Prio+
10	100k	1.90	28.67 (24.84)	1.42	11.97
20	100k	3.60	56.14 (49.96)	2.23	14.38
30	100k	4.96	83.62 (75.66)	3.1	19.41
40	100k	7.15	111.11 (101.98)	3.99	23.75
10	300k	5.58	84.84 (75.9)	4.45	51.99
20	300k	10.81	168.1 (152.99)	6.97	61.63
30	300k	16.10	250.68 (230.63)	9.6	N/A
40	300k	21.39	338.21 (311.63)	12.23	N/A

Table 6.3.: End-to-end total data sent (GBs) comparison in a two-server setting. Parenthesized value is the data sent by vOLE. N/A means that the program aborted.

#Clients	#Params	Alpha-Ho	Alpha-DiHo	Elsa	Prio+
10	100k	0.79	32.29 (31.50)	0.82	0.55
20	100k	1.57	64.57 (63.00)	1.65	1.1
30	100k	2.36	96.86 (94.50)	2.47	1.65
40	100k	3.14	129.14 (126.00)	3.29	2.2
10	300k	2.36	96.86 (94.50)	2.47	1.97
20	300k	4.72	193.72 (189.00)	4.94	3.93
30	300k	7.07	290.57 (283.50)	7.41	N/A
40	300k	9.43	387.43 (378.00)	9.88	N/A

End-to-end Runtime Comparison. However, Table 6.1 shows that despite this communication advantage, RoFL is still significantly slower than AlphaFL. AlphaFL-Ho is at least 3 magnitudes faster, while AlphaFL-DiHo is at least 2 magnitudes faster. After studying the segmented time of RoFL, we notice that the most of the time is spent generating ZKPs at the client side.

6.5.3. Comparison in the Two-Server Setting

Baseline. In the two-server setting, we consider two state-of-the-art frameworks:

- Elsa [141] at commit `eabcfd2` with $w = 32$, $l = 64$ as a direct comparison. The total runtime is calculated by summing up all recorded duration at server Alice. The total data sent is calculated by summing all recorded data at both servers, Alice and Bob;
- Prio+ [2] at commit `eabcfd2` (provided by Elsa [141]) with $w = 32$, $l = 64$. The total runtime and the total data sent are evaluated in the same way as in Elsa [141].

Parameter Size. To analyze the scalability of the frameworks, we vary both the gradient vector size and the number of clients for our experiment. We denote the gradient sizes as #Params (corresponding to the vector size t in Π^{InCom} and $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$), where $t \in \{100\text{k}, 300\text{k}\}$. We denote the number of clients as #Clients (corresponding to the client set size n in Π^{InCom} and $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$), where $n \in \{10, 20, 30, 40\}$.

End-to-End Runtime Comparison. We show the end-to-end runtime comparison in Table 6.2. In general, AlphaFL-Ho achieves malicious security under the non-collusion setting and incurs higher runtime than Elsa, but remains more efficient than Prio+:

- Compared to Elsa, AlphaFL-Ho incurs $34\% - 79\%$ additional runtime overhead for $t = 100\text{k}$ and $25\% - 75\%$ additional overhead for $t = 300\text{k}$.
- Compared to Prio+, AlphaFL-Ho is between $2.32\times$ and $5.30\times$ faster for $t = 100\text{k}$, and between $4.70\times$ and $8.32\times$ faster for $t = 300\text{k}$.

On the other hand, AlphaFL-DiHo is completely outperformed by both frameworks. Since the aggregation part in both AlphaFL-Ho and AlphaFL-DiHo is identical, we can conclude that executing the input commitment protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ is the most time-consuming part in AlphaFL-DiHo. We note that in both AlphaFL-Ho and AlphaFL-DiHo, the parties must additionally execute the silent select protocol, which is not required in Elsa or Prio+.

Communication Comparison. We present the required communication overhead in Table 6.3. We observe that the communication volume required by AlphaFL-Ho is very close to that of Elsa and Prio+, whereas AlphaFL-DiHo incurs approximately $40\times$ more communication compared to AlphaFL-Ho. This increase is due to the invocation of the vOLE functionality $\mathcal{F}^{\mathsf{vOLE}}$ within $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$.

6.5.4. Breakdown

In this section, we analyze the runtime and communication efficiency of each module and provide breakdown benchmarks in Fig. 6.9 and Fig. 6.10. We first fix the gradient size to 100k and vary the number of clients, then we fix the number of clients to 20 and vary the gradient size.

Breakdown of AlphaFL-Ho. While running AlphaFL-Ho (in the honest-majority setting), the runtimes of the input commitment protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$, the boolean-to-arithmetic protocol (B2A) and the L_2-Norm check are nearly identical as shown in (a) and (c) of Fig. 6.9. The aggregation and the MAC check phases contribute only a small portion of the overall runtime. On the other hand, the communication overhead of Π^{InCom} dominates all other modular protocols as shown in (b) and (d) of Fig. 6.9. This indicates that the local computation involved in the Boolean-to-Arithmetic (B2A) protocol and the L_2-Norm check significantly impacts the overall execution runtime.

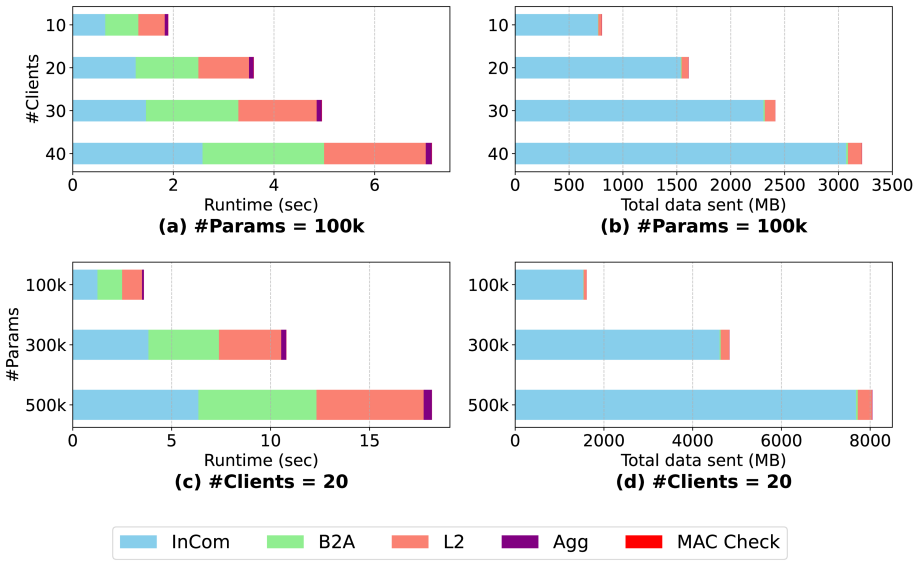


Figure 6.9.: Runtime and Total Data Sent Breakdown of AlphaFL-Ho

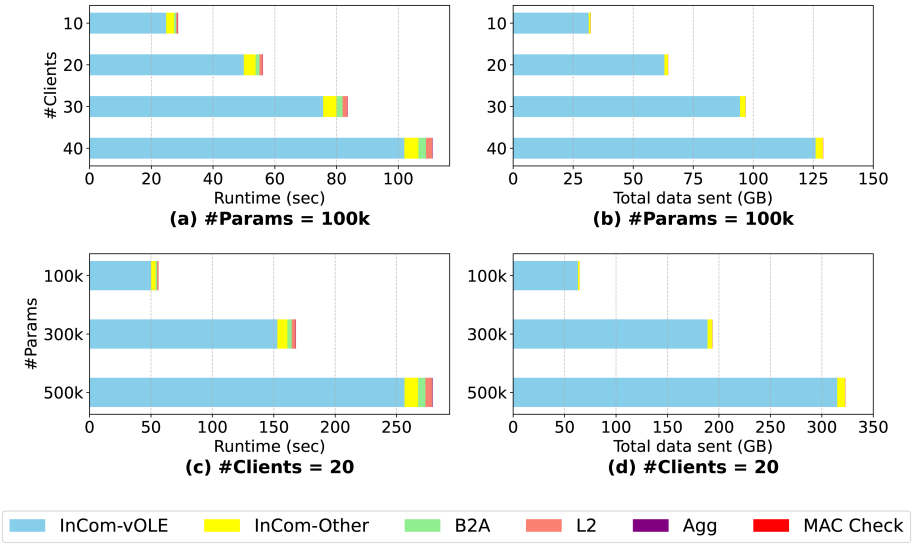


Figure 6.10.: Runtime and Total Data Sent Breakdown of AlphaFL-DiHo

Breakdown of AlphaFL-DiHo**.** While running AlphaFL-DiHo (in the dishonest-majority setting), Fig 6.10 shows that the protocol $\Pi^{\text{InCom}}_{\text{Diho}}$ accounts for the overwhelming majority of the total execution time. The absolute runtime and communication volume of $\Pi^{\text{InCom}}_{\text{Diho}}$ increase proportionally with the parameter sizes and the number of clients, while the total cost, particularly the communication cost, of the other components remains negligible. Within $\Pi^{\text{InCom}}_{\text{Diho}}$, we denote the cost of the vOLE invocation as "InCom-vOLE" and the remainder of the protocol execution as "InCom-Other" as shown in Fig. 6.10. We observe that the vOLE invocation accounts for approximately 96% of both the runtime and the communication traffic during the execution of $\Pi^{\text{InCom}}_{\text{Diho}}$.

Part III.

Conclusion

In this thesis, we studied how multi-party computation (MPC) can be employed to realize privacy-preserving machine learning under different system architectures and threat models. In the first part, we presented Force, a four-party framework for secure neural network training. We introduced a novel four-party $\mathcal{X}$-sharing scheme in the honest-majority setting and designed efficient four-party computation protocols built upon this sharing primitive. In the second part, we considered the privacy-preserving training of gradient boosting decision trees (GBDT). We first developed NodeGuard, a framework based on secret sharing. We then improved its efficiency by replacing the underlying modular protocols with constructions based on function secret sharing (FSS), leading to the enhanced framework SiGBDT. In the third part, we investigated the federated learning setting and proposed AlphaFL. This framework provides malicious security against a dishonest majority from a cryptographic perspective, while also offering protection against poisoning attacks in federated learning.

For future work, we will investigate whether $\mathcal{X}$-sharing scheme can be applied to broader applications, particularly for more complex machine learning algorithms such as large language models. Except using a pure secret sharing scheme as the technique to realize privacy-preserving machine learning, we will also explore alternatives to MPC techniques. For example, we will consider leveraging trusted execution environments (TEEs) to further accelerate online computation. Such optimizations can be tailored to satisfy different security guarantees, as secure enclaves may be modeled under various threat models.

Bibliography

[1] Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. "Secure training of decision trees with continuous attributes". In: *Proceedings on Privacy Enhancing Technologies* 1 (2021), pp. 167–187.

[2] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. "Prio+: Privacy preserving aggregate statistics via boolean shares". In: *International Conference on Security and Cryptography for Networks*. Springer. 2022, pp. 516–539.

[3] Sebastien Andreina, Giorgia Azzurra Marson, Helen Möllering, and Ghassan Karame. "Baffle: Backdoor detection via feedback-based federated learning". In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021, pp. 852–863.

[4] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. "High-throughput semi-honest secure three-party computation with an honest majority". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 805–817.

[5] Sana Awan, Bo Luo, and Fengjun Li. "Contra: Defending against poisoning attacks in federated learning". In: *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer. 2021, pp. 455–475.

[6] Eugene Bagdasaryan, Andreas Veit, Yiqing Hua, Deborah Estrin, and Vitaly Shmatikov. "How to backdoor federated learning". In: *International conference on artificial intelligence and statistics*. PMLR. 2020, pp. 2938–2948.

[7] Soumya Banerjee, Sandip Roy, Sayyed Farid Ahamed, Devin Quinn, Marc Vucovich, Dhruv Nandakumar, Kevin Choi, Abdul Rahman, Edward Bowen, and Sachin Shetty. "Mia-bad: An approach for enhancing membership inference attack and its mitigation with federated learning". In: *2024 International Conference on Computing, Networking and Communications (ICNC)*. IEEE. 2024, pp. 635–640.

[8] Laasya Bangalore, Mohammad Hossein Faghihi Sereshgi, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. "Flag: A framework for lightweight robust secure aggregation". In: *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 2023, pp. 14–28.

[9] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. "Efficient constant-round MPC with identifiable abort and public verifiability". In: *Annual International Cryptology Conference*. Springer. 2020, pp. 562–592.

[10] Donald Beaver. "Efficient multiparty protocols using circuit randomization". In: *Advances in Cryptology—CRYPTO'91: Proceedings 11*. Springer. 1992, pp. 420–432.

[11] Rouzbeh Behnia, Arman Riasi, Reza Ebrahimi, Sherman SM Chow, Balaji Padmanabhan, and Thang Hoang. "Efficient secure aggregation for privacy-preserving federated machine learning". In: *2024 Annual Computer Security Applications Conference (ACSAC)*. IEEE. 2024, pp. 778–793.

[12] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. "{ACORN}: input validation for secure aggregation". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 4805–4822.

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

[14] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. "Analyzing federated learning through an adversarial lens". In: *International conference on machine learning*. PMLR. 2019, pp. 634–643.

[15] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. "Machine learning with adversaries: Byzantine tolerant gradient descent". In: *Advances in neural information processing systems* 30 (2017).

[16] Franziska Boenisch, Adam Dziedzic, Roei Schuster, Ali Shahin Shamsabadi, Ilia Shumailov, and Nicolas Papernot. "When the curious abandon honesty: Federated learning is not private". In: *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2023, pp. 175–199.

[17] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. "Practical secure aggregation for privacy-preserving machine learning". In: *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1175–1191.

[18] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. "Function secret sharing for mixed-mode and fixed-point secure computation". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2021, pp. 871–900.

[19] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function secret sharing". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2015, pp. 337–367.

[20] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function secret sharing: Improvements and extensions". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1292–1303.

[21] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Secure computation with preprocessing via function secret sharing". In: *Theory of Cryptography: 17th International Conference, TCC 2019, Nuremberg, Germany, December 1–5, 2019, Proceedings, Part I 17*. Springer. 2019, pp. 341–371.

[22] L. Breiman, J. H. Freidman, Richard A. Olshen, and C. J. Stone. "CART: Classification and Regression Trees". In: 1984. URL: https://api.semanticscholar.org/CorpusID:59814698.

[23] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. "Bulletproofs: Short proofs for confidential transactions and more". In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 315–334.

[24] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. "FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning". In: *Proceedings on Privacy Enhancing Technologies* 2 (2020), pp. 459–480.

[25] David Byrd and Antigoni Polychroniadou. "Differentially private secure multi-party computation for federated learning in financial applications". In: *Proceedings of the First ACM International Conference on AI in Finance*. 2020, pp. 1–9.

[26] Yuxuan Cai, Wenxiu Ding, Yuxuan Xiao, Zheng Yan, Ximeng Liu, and Zhiguo Wan. "SecFed: A Secure and Efficient Federated Learning Based on Multi-Key Homomorphic Encryption". In: *IEEE Transactions on Dependable and Secure Computing* (2023).

[27] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. "Leaf: A benchmark for federated settings". In: *arXiv preprint arXiv:1812.01097* (2018).

[28] Ran Canetti. "Universally composable security: A new paradigm for cryptographic protocols". In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE. 2001, pp. 136–145.

[29] Ran Canetti and Tal Rabin. "Universal composition with joint state". In: *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*. Springer. 2003, pp. 265–281.

[30] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. "Fltrust: Byzantine-robust federated learning via trust bootstrapping". In: *Network and Distributed Systems Security (NDSS) Symposium* (2021).

[31] Octavian Catrina and Amitabh Saxena. "Secure computation with fixed-point numbers". In: *Financial Cryptography and Data Security: 14th International Conference, FC 2010, Tenerife, Canary Islands, January 25-28, 2010, Revised Selected Papers 14*. Springer. 2010, pp. 35–50.

[32] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. "Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning". In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[33] Bryant Chen, Wilka Carvalho, Heiko H Ludwig, Ian Michael Molloy, Taesung Lee, Jialong Zhang, and Benjamin J Edwards. *Detecting poisoning attacks on neural networks by activation clustering*. US Patent 11,188,789. 2021.

[34] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[35] Weijing Chen, Guoqiang Ma, Tao Fan, Yan Kang, Qian Xu, and Qiang Yang. "Secureboost+: A high performance gradient boosting tree framework for large scale vertical federated learning". In: *arXiv preprint arXiv:2110.10927* (2021).

[36] Kewei Cheng, Tao Fan, Yilun Jin, Yang Liu, Tianjian Chen, Dimitrios Papadopoulos, and Qiang Yang. "Secureboost: A lossless federated learning framework". In: *IEEE Intelligent Systems* 36.6 (2021), pp. 87–98.

[37] Amrita Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. "EIFFeL: Ensuring Integrity for Federated Learning". In: Nov. 2022, pp. 2535–2549. DOI: 10.1145/3548606.3560611.

[38] Henry Corrigan-Gibbs and Dan Boneh. "Prio: Private, robust, and scalable computation of aggregate statistics". In: *14th USENIX symposium on networked systems design and implementation (NSDI 17)*. 2017, pp. 259–282.

[39] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. "SPDZ2k: Efficient MPC mod 2^k for Dishonest Majority". In: *Advances in Cryptology – CRYPTO 2018*. Springer International Publishing, 2018, pp. 769–798.

[40] Tianxiang Dai, Li Duan, Yufan Jiang, Yong Li, Fei Mei, and Yulian Sun. "Force: Highly Efficient Four-Party Privacy-Preserving Machine Learning on GPU". In: *Secure IT Systems: 28th Nordic Conference, NordSec 2023, Oslo, Norway, November 16–17, 2023, Proceedings*. Oslo, Norway: Springer-Verlag, 2023, pp. 330–349. ISBN: 978-3-031-47747-8. DOI: 10.1007/978-3-031-47748-5_18. URL: https://doi.org/10.1007/978-3-031-47748-5_18.

[41] Tianxiang Dai, Li Duan, Yufan Jiang, Yong Li, Fei Mei, and Yulian Sun. *Force: Highly Efficient Four-Party Privacy-Preserving Machine Learning on GPU*. Cryptology ePrint Archive, Paper 2023/493. 2023. URL: https://eprint.iacr.org/2023/493.

[42] Tianxiang Dai, Li Duan, Yufan Jiang, Yong Li, Fei Mei, and Yulian Sun. *Force: Making 4PC > 4 × PC in Privacy Preserving Machine Learning on GPU*. Cryptology ePrint Archive, Paper 2023/493. 2023. URL: https://eprint.iacr.org/2023/493.

[43] Tianxiang Dai, Yufan Jiang, Yong Li, and Fei Mei. " NodeGuard: A Highly Efficient Two-Party Computation Framework for Training Large-Scale Gradient Boosting Decision Tree ". In: *2024 IEEE Security and Privacy Workshops (SPW)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 95–103. DOI: 10.1109/SPW63631.2024.00015. URL: https://doi.ieeecomputersociety.org/10.1109/SPW63631.2024.00015.

[44] Tianxiang Dai, Yufan Jiang, Yong Li, and Fei Mei. *NodeGuard: A Highly Efficient Two-Party Computation Framework for Training Large-Scale Gradient Boosting Decision Tree*. Cryptology ePrint Archive, Paper 2024/535. 2024. URL: https://eprint.iacr.org/2024/535.

[45] Anders Dalskov, Daniel Escudero, and Marcel Keller. "Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2183–2200.

[46] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. "New primitives for actively-secure MPC over rings with applications to private machine learning". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1102–1120.

[47] Saroj Dayal, Dima Alhadidi, Ali Abbasi Tadi, and Noman Mohammed. "Comparative analysis of membership inference attacks in federated learning". In: *Proceedings of the 27th International Database Engineered Applications Symposium*. 2023, pp. 185–192.

[48] Sebastiaan De Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. "Practical secure decision tree learning in a teletreatment application". In: *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18*. Springer. 2014, pp. 179–194.

[49] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY-A framework for efficient mixed-protocol secure two-party computation." In: *NDSS*. 2015.

[50] Ye Dong, Xiaojun Chen, Kaiyun Li, Dakui Wang, and Shuai Zeng. "FLOD: Oblivious defender for private Byzantine-robust federated learning with dishonest-majority". In: *European Symposium on Research in Computer Security*. Springer. 2021, pp. 497–518.

[51] Haohua Duan, Zedong Peng, Liyao Xiang, Yuncong Hu, and Bo Li. "A Verifiable and Privacy-Preserving Federated Learning Training Framework". In: *IEEE Transactions on Dependable and Secure Computing* (2024).

[52] Cynthia Dwork, Aaron Roth, et al. "The algorithmic foundations of differential privacy". In: *Foundations and Trends® in Theoretical Computer Science* 9.3–4 (2014), pp. 211–407.

[53] Cynthia Dwork, Guy N Rothblum, and Salil Vadhan. "Boosting and differential privacy". In: *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE. 2010, pp. 51–60.

[54] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. "Improved primitives for MPC over mixed arithmetic-binary circuits". In: *Annual International Cryptology conference*. Springer. 2020, pp. 823–852.

[55] Minghong Fang, Xiaoyu Cao, Jinyuan Jia, and Neil Gong. "Local model poisoning attacks to {Byzantine-Robust} federated learning". In: *29th USENIX security symposium (USENIX Security 20)*. 2020, pp. 1605–1622.

[56] Wenjing Fang, Derun Zhao, Jin Tan, Chaochao Chen, Chaofan Yu, Li Wang, Lei Wang, Jun Zhou, and Benyu Zhang. "Large-scale secure XGB for vertical federated learning". In: *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2021, pp. 443–452.

[57] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[58] Fangcheng Fu, Yingxia Shao, Lele Yu, Jiawei Jiang, Huanran Xue, Yangyu Tao, and Bin Cui. "Vf2boost: Very fast vertical federated gradient boosting for cross-enterprise learning". In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 563–576.

[59] Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. "Mitigating sybils in federated learning poisoning". In: *arXiv preprint arXiv:1808.04866* (2018).

[60] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. "High-throughput secure three-party computation for malicious adversaries and an honest majority". In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2017, pp. 225–255.

[61] Till Gehlhar, Felix Marx, Thomas Schneider, Ajith Suresh, Tobias Wehrle, and Hossein Yalame. "SafeFL: MPC-friendly framework for private and robust federated learning". In: *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2023, pp. 69–76.

[62] Robert E Goldschmidt. "Applications of division by convergence". PhD thesis. Massachusetts Institute of Technology, 1964.

[63] Xueluan Gong, Yanjiao Chen, Qian Wang, and Weihan Kong. "Backdoor attacks and defenses in federated learning: State-of-the-art, taxonomy, and future directions". In: *IEEE Wireless Communications* 30.2 (2022), pp. 114–121.

[64] Xiaojie Guo, Zheli Liu, Jin Li, Jiqiang Gao, Boyu Hou, Changyu Dong, and Thar Baker. "V eri fl: Communication-efficient and fast verifiable aggregation for federated learning". In: *IEEE Transactions on Information Forensics and Security* 16 (2020), pp. 1736–1751.

[65] Changhee Hahn, Hodong Kim, Minjae Kim, and Junbeom Hur. "Versa: Verifiable secure aggregation for cross-device federated learning". In: *IEEE Transactions on Dependable and Secure Computing* 20.1 (2021), pp. 36–52.

[66] Alon Halevy, Peter Norvig, and Fernando Pereira. "The unreasonable effectiveness of data". In: *IEEE intelligent systems* 24.2 (2009), pp. 8–12.

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[68] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. "Privacy-preserving Machine Learning as a Service". In: *Proceedings on Privacy Enhancing Technologies* 3 (2018), pp. 123–142.

[69] S Hochreiter. "Long Short-term Memory". In: *Neural Computation MIT-Press* (1997).

[70] Hongsheng Hu, Zoran Salcic, Lichao Sun, Gillian Dobbie, and Xuyun Zhang. "Source inference attacks in federated learning". In: *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE. 2021, pp. 1102–1107.

[71] Hongsheng Hu, Xuyun Zhang, Zoran Salcic, Lichao Sun, Kim-Kwang Raymond Choo, and Gillian Dobbie. "Source inference attacks: Beyond membership inference attacks in federated learning". In: *IEEE Transactions on Dependable and Secure Computing* (2023).

[72] Chao Huang, Yanqing Yao, Xiaojun Zhang, Da Teng, Yingdong Wang, and Lei Zhou. "Robust Secure Aggregation with Lightweight Verification for Federated Learning". In: *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE. 2022, pp. 582–589.

[73] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. "Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 809–826.

[74] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. "Orca: FSS-based Secure Training with GPUs". In: *Cryptology ePrint Archive* (2023).

[75] Yufan Jiang, Fei Mei, Tianxiang Dai, and Yong Li. "SiGBDT: Large-Scale Gradient Boosting Decision Tree Training via Function Secret Sharing". In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 274–288. ISBN: 9798400704826. DOI: 10.1145/3634737.3657024. URL: https://doi.org/10.1145/3634737.3657024.

[76] Yufan Jiang, Maryam Zarezadeh, Tianxiang Dai, and Stefan Köpsell. *AlphaFL: Secure Aggregation with Malicious^2 Security for Federated Learning against Dishonest Majority*. Cryptology ePrint Archive, Paper 2025/1289. 2025. URL: https://eprint.iacr.org/2025/1289.

[77] Yufan Jiang, Maryam Zarezadeh, Tianxiang Dai, and Stefan Köpsell. "AlphaFL: Secure Aggregation with Malicious2 Security for Federated Learning against Dishonest Majority". In: *Proceedings on Privacy Enhancing Technologies* 2025 (4), pp. 348–368. DOI: https://doi.org/10.56553/popets-2025-0134.

[78] Zhifeng Jiang, Wei Wang, and Ruichuan Chen. "Dordis: Efficient Federated Learning with Dropout-Resilient Differential Privacy". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. 2024, pp. 472–488.

[79] Weizhao Jin, Yuhang Yao, Shanshan Han, Carlee Joe-Wong, Srivatsan Ravi, Salman Avestimehr, and Chaoyang He. "FedML-HE: An efficient homomorphic-encryption-based privacy-preserving federated learning system". In: *arXiv preprint arXiv:2303.10837* (2023).

[80] Pawel Jurzak, Grzegorz Kaplita, Wojciech Kucharski, and Stefan Koprowski. *Methods and apparatus for detecting malicious re-training of an anomaly detection system.* US Patent 12,013,950. 2024.

[81] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A low latency framework for secure neural network inference". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1651–1669.

[82] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. "Advances and open problems in federated learning". In: *Foundations and trends® in machine learning* 14.1–2 (2021), pp. 1–210.

[83] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. "Lightgbm: A highly efficient gradient boosting decision tree". In: *Advances in neural information processing systems* 30 (2017).

[84] Marcel Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2020, pp. 1575–1590.

[85] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: making SPDZ great again". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2018, pp. 158–189.

[86] Youssef Khazbak, Tianxiang Tan, and Guohong Cao. "MLGuard: Mitigating poisoning attacks in privacy preserving distributed collaborative learning". In: *2020 29th international conference on computer communications and networks (ICCCN)*. IEEE. 2020, pp. 1–9.

[87] Denise-Phi Khuu, Michael Sober, Dominik Kaaser, Mathias Fischer, and Stefan Schulte. "Data Poisoning Detection in Federated Learning". In: *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. 2024, pp. 1549–1558.

[88] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. "Crypten: Secure multi-party computation meets machine learning". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 4961–4973.

[89] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. "SWIFT: Superfast and Robust Privacy-Preserving Machine Learning". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2651–2668. ISBN: 978-1-939133-24-3.

[90] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. "Tetrad: Actively Secure 4PC for Secure Training and Inference". In: *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[91] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[92] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012.

[93] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[94] Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. "Secure collaborative training and inference for xgboost". In: *Proceedings of the 2020 workshop on privacy-preserving machine learning in practice*. 2020, pp. 21–26.

[95] Ya Le and Xuan Yang. "Tiny imagenet visual recognition challenge". In: *CS 231N* 7.7 (2015), p. 3.

[96] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4 (1989), pp. 541–551.

[97] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. "Muse: Secure inference resilient to malicious clients". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2201–2218.

[98] Chester Leung, Andrew Law, and Octavian Sima. "Towards privacy-preserving collaborative gradient boosted decision trees". In: *UC Berkeley* (2019).

[99] Suyi Li, Yong Cheng, Wei Wang, Yang Liu, and Tianjian Chen. "Learning to detect malicious clients for robust federated learning". In: *arXiv preprint arXiv:2002.00211* (2020).

[100] Xueyang Li, Xue Yang, Zhengchun Zhou, and Rongxing Lu. "Efficiently Achieving Privacy Preservation and Poisoning Attack Resistance in Federated Learning". In: *IEEE Transactions on Information Forensics and Security* (2024).

[101] Yi Li and Wei Xu. "PrivPy: General and scalable privacy-preserving data mining". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 1299–1307.

[102] Yehuda Lindell. "How to simulate it–a tutorial on the simulation proof technique". In: *Tutorials on the Foundations of Cryptography* (2017), pp. 277–346.

[103] Yehuda Lindell and Benny Pinkas. "Privacy preserving data mining". In: *Annual International Cryptology Conference*. Springer. 2000, pp. 36–54.

[104] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. "Model ensemble for click prediction in bing search ads". In: *Proceedings of the 26th international conference on world wide web companion*. 2017, pp. 689–698.

[105] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. "Oblivious neural network predictions via minionn transformations". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 619–631.

[106] Kunlong Liu and Trinabh Gupta. "Federated learning with differential privacy and an untrusted aggregator". In: *arXiv preprint arXiv:2312.10789* (2023).

[107] Zizhen Liu, Weiyang He, Chip-Hong Chang, Jing Ye, Huawei Li, and Xiaowei Li. "SPFL: A Self-purified Federated Learning Method Against Poisoning Attacks". In: *IEEE Transactions on Information Forensics and Security* (2024).

[108] Wen-jie Lu, Zhicong Huang, Qizhi Zhang, Yuchen Wang, and Cheng Hong. "Squirrel: A Scalable Secure Two-Party Computation Framework for Training Gradient Boosting Decision Tree". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 6435–6451. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/lu.

[109] Fucai Luo, Saif Al-Kuwari, and Yong Ding. "SVFL: Efficient secure aggregation and verification for cross-silo federated learning". In: *IEEE Transactions on Mobile Computing* 23.1 (2022), pp. 850–864.

[110] Hidde Lycklama, Lukas Burkhalter, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. "Rofl: Robustness of secure federated learning". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 453–476.

[111] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. "SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023, pp. 17–33.

[112] Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. "Flamingo: Multi-round single-server secure aggregation with applications to private federated learning". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 477–496.

[113] Zhuoran Ma, Jianfeng Ma, Yinbin Miao, Yingjiu Li, and Robert H Deng. "ShieldFL: Mitigating model poisoning attacks in privacy-preserving federated learning". In: *IEEE Transactions on Information Forensics and Security* 17 (2022), pp. 1639–1654.

[114] Yunlong Mao, Xinyu Yuan, Xinyang Zhao, and Sheng Zhong. "Romoa: Robust model aggregation for the resistance of federated learning to model poisoning attacks". In: *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer. 2021, pp. 476–496.

[115] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. "Communication-efficient learning of deep networks from decentralized data". In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 1273–1282.

[116] Yuan Meng, Nianhua Yang, Zhilin Qian, and Gaoyu Zhang. "What makes an online review more helpful: an interpretation framework using XGBoost and SHAP values". In: *Journal of Theoretical and Applied Electronic Commerce Research* 16.3 (2020), pp. 466–490.

[117] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. "Delphi: A cryptographic inference service for neural networks". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2505–2522.

[118] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. "PPFL: privacy-preserving federated learning with trusted execution environments". In: *Proceedings of the 19th annual international conference on mobile systems, applications, and services*. 2021, pp. 94–108.

[119] Payman Mohassel and Peter Rindal. "ABY3: A mixed protocol framework for machine learning". In: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018, pp. 35–52.

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

[121] Arup Mondal, Yash More, Ruthu Hulikal Rooparaghunath, and Debayan Gupta. "Poster: Flatee: Federated learning across trusted execution environments". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2021, pp. 707–709.

[122] Xutong Mu, Ke Cheng, Yulong Shen, Xiaoxiao Li, Zhao Chang, Tao Zhang, and Xindi Ma. "FedDMC: Efficient and Robust Federated Learning via Detecting Malicious Clients". In: *IEEE Transactions on Dependable and Secure Computing* (2024).

[123] Luis Muñoz-González, Kenneth T Co, and Emil C Lupu. "Byzantine-robust federated machine learning through adaptive model averaging". In: *arXiv preprint arXiv:1909.05125* (2019).

[124] Moni Naor and Benny Pinkas. "Efficient oblivious transfer protocols". In: *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '01. Washington, D.C., USA: Society for Industrial and Applied Mathematics, 2001, pp. 448–457. ISBN: 0898714907.

[125] Milad Nasr, Reza Shokri, and Amir Houmansadr. "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning". In: *2019 IEEE symposium on security and privacy (SP)*. IEEE. 2019, pp. 739–753.

[126] Lucien K. L. Ng and Sherman S. M. Chow. "SoK: Cryptographic Neural-Network Computation". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 497–514. DOI: 10.1109/SP46215.2023.10179483.

[127] Thien Duc Nguyen, Phillip Rieger, Roberta De Viti, Huili Chen, Björn B Brandenburg, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, et al. "{FLAME}: Taming backdoors in federated learning". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 1415–1432.

[128] Thien Duc Nguyen, Phillip Rieger, Mohammad Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Ahmad-Reza Sadeghi, Thomas Schneider, et al. "Flguard: Secure and private federated learning". In: *Crytography and Security* Preprint (2021).

[129] Thuy Dung Nguyen, Tuan A Nguyen, Anh Tran, Khoa D Doan, and Kok-Seng Wong. "Iba: Towards irreversible backdoor attacks in federated learning". In: *Advances in Neural Information Processing Systems* 36 (2024).

[130] Truc Nguyen and My T Thai. "Preserving privacy and security in federated learning". In: *IEEE/ACM Transactions on Networking* (2023).

[131] OpenAI. *ChatGPT: Large language model*. `https://chat.openai.com/`. 2023.

[132] Ole-Edvard Ørebæk and Marius Geitle. "Exploring the Hyperparameters of XGBoost Through 3D Visualizations." In: *AAAI Spring Symposium: Combining Machine Learning with Knowledge Engineering*. 2021.

[133] Pascal Paillier. "Public-key cryptosystems based on composite degree residuosity classes". In: *Advances in Cryptology—EUROCRYPT'99*. Springer, 1999, pp. 223–238.

[134] Jaehyoung Park and Hyuk Lim. "Privacy-preserving federated learning using homomorphic encryption". In: *Applied Sciences* 12.2 (2022), p. 734.

[135] Dario Pasquini, Danilo Francati, and Giuseppe Ateniese. "Eluding secure aggregation in federated learning via model inconsistency". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 2429–2443.

[136] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[137] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. "ABY2. 0: Improved Mixed-Protocol Secure Two-Party Computation". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2165–2182.

[138] Arpita Patra and Ajith Suresh. "BLAZE: Blazing Fast Privacy-Preserving Machine Learning". In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[139] Simon Queyrut, Valerio Schiavoni, and Pascal Felber. "Mitigating adversarial attacks in federated learning with trusted execution environments". In: *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2023, pp. 626–637.

[140] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. "CrypTFlow2: Practical 2-party secure inference". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 325–342.

[141] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. "Elsa: Secure aggregation for federated learning with malicious actors". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 1961–1979.

[142] Yanli Ren, Yerong Li, Guorui Feng, and Xinpeng Zhang. "Privacy-enhanced and verification-traceable aggregation for federated learning". In: *IEEE Internet of Things Journal* 9.24 (2022), pp. 24933–24948.

[143] Gabriel Rushin, Cody Stancil, Muyang Sun, Stephen Adams, and Peter Beling. "Horse race analysis in credit card fraud—deep learning, logistic regression, and Gradient Boosted Tree". In: *2017 systems and information engineering design symposium (SIEDS)*. IEEE. 2017, pp. 117–121.

[144] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[145] Théo Ryffel, Pierre Tholoniat, David Pointcheval, and Francis Bach. "AriaNN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing". In: *Proceedings on Privacy Enhancing Technologies* 1 (2022), pp. 291–316.

[146] Zeinab Shahbazi and Yung-Cheol Byun. "Product recommendation based on content-based filtering using XGBoost classifier". In: *Int. J. Adv. Sci. Technol* 29 (2019), pp. 6979–6988.

[147] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[148] Mengxue Shang, Dandan Zhang, and Fengyin Li. "Decentralized distributed federated learning based on multi-key homomorphic encryption". In: *2023 International Conference on Data Security and Privacy Protection (DSPP)*. IEEE. 2023, pp. 260–265.

[149] Virat Shejwalkar, Amir Houmansadr, Peter Kairouz, and Daniel Ramage. "Back to the drawing board: A critical evaluation of poisoning attacks on production federated learning". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 1354–1371.

[150] Shiqi Shen, Shruti Tople, and Prateek Saxena. "Auror: Defending against poisoning attacks in collaborative deep learning systems". In: *Proceedings of the 32nd annual conference on computer security applications*. 2016, pp. 508–519.

[151] Lei Shi, Zhen Chen, Yucheng Shi, Guangtao Zhao, Lin Wei, Yongcai Tao, and Yufei Gao. "Data poisoning attacks on federated learning by using adversarial samples". In: *2022 International Conference on Computer Engineering and Artificial Intelligence (ICCEAI)*. IEEE. 2022, pp. 158–162.

[152] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015.

[153] Timothy Stevens, Christian Skalka, Christelle Vincent, John Ring, Samuel Clark, and Joseph Near. "Efficient differentially private secure aggregation for federated learning via hardness of learning with errors". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 1379–1395.

[154] Gan Sun, Yang Cong, Jiahua Dong, Qiang Wang, Lingjuan Lyu, and Ji Liu. "Data poisoning attacks on federated machine learning". In: *IEEE Internet of Things Journal* 9.13 (2021), pp. 11365–11375.

[155] Ziteng Sun, Peter Kairouz, Ananda Theertha Suresh, and H Brendan McMahan. "Can you really backdoor federated learning?" In: *arXiv preprint arXiv:1911.07963* (2019).

[156] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. "CryptGPU: Fast privacy-preserving machine learning on the GPU". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1021–1038.

[157] Zhenya Tian, Jialiang Xiao, Haonan Feng, and Yutian Wei. "Credit risk assessment based on gradient boosting decision tree". In: *Procedia Computer Science* 174 (2020), pp. 150–160.

[158] Zhihua Tian, Rui Zhang, Xiaoyang Hou, Lingjuan Lyu, Tianyi Zhang, Jian Liu, and Kui Ren. "FederBoost: Private Federated Learning for GBDT". In: *IEEE Transactions on Dependable and Secure Computing* (2023).

[159] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. "Data poisoning attacks against federated learning systems". In: *Computer security–ESORICs 2020: 25th European symposium on research in computer security, ESORICs 2020, guildford, UK, September 14–18, 2020, proceedings, part i 25*. Springer. 2020, pp. 480–501.

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

[161] Stacey Truex, Ling Liu, Ka-Ho Chow, Mehmet Emre Gursoy, and Wenqi Wei. "LDP-Fed: Federated learning with local differential privacy". In: *Proceedings of the third ACM international workshop on edge systems, analytics and networking*. 2020, pp. 61–66.

[162] United Nations Conference on Trade and Development. *Data Protection and Privacy Legislation Worldwide*. 2022. URL: https://unctad.org/page/data-protection-and-privacy-legislation-worldwide.

[163] Sameer Wagh, Divya Gupta, and Nishanth Chandran. "SecureNN: 3-Party Secure Computation for Neural Network Training." In: *Proc. Priv. Enhancing Technol.* 2019.3 (2019), pp. 26–49.

[164] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. "Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning". In: *Proceedings on Privacy Enhancing Technologies* 1 (2021), pp. 188–208.

[165] Hongyi Wang, Kartik Sreenivasan, Shashank Rajput, Harit Vishwakarma, Saurabh Agarwal, Jy-yong Sohn, Kangwook Lee, and Dimitris Papailiopoulos. "Attack of the tails: Yes, you really can backdoor federated learning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 16070–16084.

[166] Ning Wang, Yang Xiao, Yimin Chen, Yang Hu, Wenjing Lou, and Y Thomas Hou. "Flare: defending federated learning against model poisoning attacks via latent space representations". In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. 2022, pp. 946–958.

[167] Yong Wang, Aiqing Zhang, Shu Wu, and Shui Yu. "VOSA: Verifiable and oblivious secure aggregation for privacy-preserving federated learning". In: *IEEE Transactions on Dependable and Secure Computing* 20.5 (2022), pp. 3601–3616.

[168] Zhipeng Wang, Nanqing Dong, Jiahao Sun, William Knottenbelt, and Yike Guo. "zkfl: Zero-knowledge proof-based gradient aggregation for federated learning". In: *IEEE Transactions on Big Data* (2024).

[169] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. "Piranha: A GPU Platform for Secure Computation". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 827–844.

[170] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H Yang, Farhad Farokhi, Shi Jin, Tony QS Quek, and H Vincent Poor. "Federated learning with differential privacy: Algorithms and performance analysis". In: *IEEE transactions on information forensics and security* 15 (2020), pp. 3454–3469.

[171] Yuncheng Wu, Shaofeng Cai, Xiaokui Xiao, Gang Chen, and Beng Chin Ooi. "Privacy preserving vertical federated learning for tree-based models". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 2090–2103.

[172] Chulin Xie, Minghao Chen, Pin-Yu Chen, and Bo Li. "Crfl: Certifiably robust federated learning against backdoor attacks". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 11372–11382.

[173] Lunchen Xie, Jiaqi Liu, Songtao Lu, Tsung-Hui Chang, and Qingjiang Shi. "An efficient learning framework for federated xgboost using secret sharing and distributed optimization". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 13.5 (2022), pp. 1–28.

[174] Guowen Xu, Hongwei Li, Sen Liu, Kan Yang, and Xiaodong Lin. "VerifyNet: Secure and verifiable federated learning". In: *IEEE Transactions on Information Forensics and Security* 15 (2019), pp. 911–926.

[175] Guowen Xu, Hongwei Li, Yun Zhang, Shengmin Xu, Jianting Ning, and Robert H Deng. "Privacy-preserving federated deep learning with irregular users". In: *IEEE Transactions on Dependable and Secure Computing* 19.2 (2020), pp. 1364–1381.

[176] Jian Xu, Shao-Lun Huang, Linqi Song, and Tian Lan. "Byzantine-robust federated learning through collaborative malicious gradient filtering". In: *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2022, pp. 1223–1235.

[177] Gang Yan, Hao Wang, Xu Yuan, and Jian Li. "Defl: Defending against model poisoning attacks in federated learning via critical learning periods awareness". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 9. 2023, pp. 10711–10719.

[178] Andrew Chi-Chih Yao. "How to generate and exchange secrets". In: *27th annual symposium on foundations of computer science (Sfcs 1986)*. IEEE. 1986, pp. 162–167.

[179] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. "Byzantine-robust distributed learning: Towards optimal statistical rates". In: *International conference on machine learning*. Pmlr. 2018, pp. 5650–5659.

[180] Haiyang Yu, Runtong Xu, Hui Zhang, Zhen Yang, and Huan Liu. "EV-FL: Efficient Verifiable Federated Learning With Weighted Aggregation for Industrial IoT Networks". In: *IEEE/ACM Transactions on Networking* (2023).

[181] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. "{BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning". In: *2020 USENIX annual technical conference (USENIX ATC 20)*. 2020, pp. 493–506.

[182] Ming Zhang, Stamatis Vassiliadis, and Jose G. Delgado-Frias. "Sigmoid generators for neural computing using piecewise approximations". In: *IEEE transactions on Computers* 45.9 (1996), pp. 1045–1049.

[183] Qizhi Zhang, Yuan Zhao, Lichun Li, JiaoFu Zhang, Qichao Zhang, Yashun Zhou, Dong Yin, Sijun Tan, and Shan Yin. "MORSE-STF: A Privacy Preserving Computation System". In: *CoRR* abs/2109.11726 (2021).

[184] Zaixi Zhang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. "Fldetector: Defending federated learning against model poisoning attacks via detecting malicious clients". In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2022, pp. 2545–2555.

[185] Huadi Zheng, Haibo Hu, and Ziyang Han. "Preserving user privacy for machine learning: Local differential privacy or federated machine learning?" In: *IEEE Intelligent Systems* 35.4 (2020), pp. 5–14.

[186] Yin Zhu, Junqing Gong, Kai Zhang, and Haifeng Qian. "Malicious-Resistant Non-Interactive Verifiable Aggregation for Federated Learning". In: *IEEE Transactions on Dependable and Secure Computing* (2024).

A. Appendix

A.1. Functionalities

A.1.1. Keyed bucket triple generation functionality $\mathcal{F}_{\text{2PC}}^{\text{KBucGen}}$

See Fig. A.1.

A.1.2. MAC Functionality $\mathcal{F}^{\text{MAC}}$

See Fig A.2.

A.1.3. Triple Generation Functionality $\mathcal{F}^{\text{TripGen}}$

See Fig. A.3.

$\mathcal{F}_{\text{2PC}}^{\text{KBucGen}}$

Internal State: **ready** $\in \{\text{true}, \text{false}\}$.
Initialization: Set **ready** = false.
Compute: Let $\mathcal{P} = \{\text{P}_h, \text{P}_{1-h}\}$. Upon receiving (BucSGen, j,P$_i$, sid) from P$_i \in \mathcal{P}$:

- If $j = 1$:
  - If P$_h$ is corrupted, for each bucket b of each feature f, wait to receive $\mathbf{r}^{f,b}$ from the adversary, where $\mathbf{r}^{f,b} = (r_0^{f,b}, ..., r_{N-1}^{f,b})$ and $r_i^{f,b} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$. Otherwise, sample $\mathbf{r}^{f,b}$.
  - Record and send all $\mathbf{r}^{f,b}$ to P$_h$ and set **ready** = true.
- If **ready** = true:
  - If P$_{1-h}$ is corrupted, wait to receive $\mathbf{k}_j$ from the adversary, where $\mathbf{k}_j = (k_{j,0}, ..., k_{j,N-1})$ and $k_{j,i} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$. Otherwise, sample $\mathbf{k}_j$.
  - Send $\mathbf{k}_j$ to P$_{1-h}$.
  - For each bucket b of each feature f: Compute $m_j^{f,b} = \mathbf{k}_j \cdot \mathbf{r}^{f,b}$. Wait to receive $[m_j^{f,b}]_c$ from $\mathcal{S}$ corrupting P$_c$, compute $[m_j^{f,b}]_{c-1} = m_j^{f,b} - [m_j^{f,b}]_c$. Send all $[m_j^{f,b}]_i$ to P$_i \in \mathcal{P}$.
- If **ready** = false, send (failed, P$_i$, sid) to P$_i \in \mathcal{P}$.

Figure A.1.: Keyed Bucket Triple Generation Functionality

Functionality $\mathcal{F}^{\text{MAC}}$

$\mathcal{F}^{\text{MAC}}$ generates shares of a global MAC key and, on input shares of a value, distributes MAC shares of this value. Let P_2 denote the corrupted party, and c is the index of the corrupted party.
Initialize: Upon receiving $(\text{Init}, S_j, \text{sid})$ from $S_j \in \{S_0, S_1\}$:

1. Wait to receive $\alpha^c \in \mathbb{Z}_{2^s}$ from the adversary. Choose $\alpha^{c-1} \in \mathbb{Z}_{2^s}$.
2. Store $\alpha = \alpha^c + \alpha^{c-1}$ (over $\mathbb{Z}$).
3. Send α^j to S_j.

Macro $\text{MacGen}(\ell, \{x^0, x^1\})$ (internal subroutine only):

1. Let $x = x^0 + x^1 \bmod 2^\ell$. Compute $m = x \cdot \alpha \bmod 2^\ell$.
2. Wait to receive m^c from the adversary, then set $m^{c-1} = m - m^c \bmod 2^\ell$. Output (m^0, m^1).

Authentication: Upon receiving $(\text{MAC}, \ell, r, \{x_i^j\}, S_j, \text{sid})$ from $S_j \in \{S_0, S_1\}$, where $x^j \in \mathbb{Z}_{2^r}$, $i \in \{0, ..., t-1\}$ and $\ell \geq r$:

1. Wait for the adversary to send a message (guess, S_j, B_j) for $j \notin c$, where B_j efficiently describes a subset of $\{0,1\}^s$. If $\alpha^j \in B_j$ then send success to the adversary. Otherwise abort.
2. Execute $\text{Auth}(\ell, \{x_i^0, x_i^1\})$ for $i \in \{0, ..., t-1\}$ and then wait for the adversary to send either OK or Abort. If the adversary sends OK then send the MAC shares $\mathbf{m}^j \in \mathbb{Z}_{2^\ell}^t$ to party S_j, otherwise abort.

Figure A.2.: MAC Functionality [39]

A.1.4. Vector Oblivious Linear Function Evaluation Functionality $\mathcal{F}^{\text{vOLE}}$

See Fig. A.4.

A.1.5. Random Bit Generation Functionality $\mathcal{F}^{\text{RanBitGen}}$

See Fig. A.5.

A.1.6. Square Correlation Generation Functionality $\mathcal{F}^{\text{SqGen}}$

See Fig. A.6.

A.1.7. Wrapper Functionality $\mathcal{F}^{\text{Wrap}}$

See Fig. A.7.

Functionality $\mathcal{F}^{\mathsf{TripGen}}$

The functionality $\mathcal{F}^{\mathsf{TripGen}}$ has all the same features as $\mathcal{F}^{\mathsf{MAC}}$, with the additional command:
Triple Generation: Upon receiving $(\mathsf{TripGen}, \mathsf{S}_j, \mathsf{sid})$ from $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$:

1. Wait to receive $(a^c, b^c, c^c) \in (\mathbb{Z}_{2^{k+s}}, \mathbb{Z}_{2^{k+s}}, \mathbb{Z}_{2^{k+s}})$ from the adversary, sample random $a^{c-1} \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$ and $b^{c-1} \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$.
2. Compute $a = a^c + a^{c-1} \bmod 2^k$ and $b = b^c + b^{c-1} \bmod 2^k$. Compute $c = a \cdot b \bmod 2^k$
3. Sample $r \xleftarrow{\$} \mathbb{Z}_{2^s}$, compute $c = c + 2^k \cdot r \bmod 2^{k+s}$, set $c^{c-1} = c - c^c \bmod 2^{k+s}$.
4. Send (a^j, b^j, c^j) to S_j.
5. Run $\mathsf{MACGen}(\{a, b, c\})$. Send (m_a^j, m_b^j, m_c^j) to S_j.

Bit Triple Generation: Upon receiving $(\mathsf{BitTripGen}, \mathsf{P}_i, \mathsf{sid})$ from $\mathsf{P}_i \in \{\mathsf{S}_0, \mathsf{S}_1\}$:

1. Wait to receive $(a^c, b^c, c^c) \in (\mathbb{Z}_{2^{k+s}}, \mathbb{Z}_{2^{k+s}}, \mathbb{Z}_{2^{k+s}})$ from the adversary, sample random $a^{c-1} \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$ and $b^{c-1} \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, such that $b = b^c + b^{c-1} \bmod 2^k$ and $b \in \{0, 1\}$.
2. Compute $a = a^c + a^{c-1} \bmod 2^k$ and $b = b^c + b^{c-1} \bmod 2^k$. Compute $c = a \cdot b \bmod 2^k$
3. Sample $r \xleftarrow{\$} \mathbb{Z}_{2^s}$, compute $c = c + 2^k \cdot r \bmod 2^{k+s}$, set $c^{c-1} = c - c^c \bmod 2^{k+s}$.
4. Send (a^j, b^j, c^j) to S_j.
5. Run $\mathsf{MACGen}(\{a, b, c\})$. Send (m_a^j, m_b^j, m_c^j) to S_j.

Figure A.3.: Triple Generation Functionality

Functionality $\mathcal{F}^{\mathsf{vOLE}}$

Initialize: Upon receiving $(\mathsf{Init}, \alpha, \mathsf{sid})$ from P_i, store α and ignore any subsequent $(\mathsf{Init}, \mathsf{sid})$ messages.
Compute: Upon receiving $(\mathsf{sid}, \ell, r, t, \mathbf{x})$ from P_j, where $\mathbf{x} \in \mathbb{Z}_{2^t}^t$:

1. Sample $\mathbf{b} \xleftarrow{\$} \mathbb{Z}_{2^\ell}^t$. If P_j is corrupted, receive $\mathbf{b} \in \mathbb{Z}_{2^\ell}^t$ from the adversary.
2. Compute $\mathbf{a} = \mathbf{b} + \alpha \cdot \mathbf{x} \bmod 2^\ell$.
3. If P_i is corrupted, receive $\mathbf{a} \in \mathbb{Z}_{2^\ell}^t$ from the adversary and compute $\mathbf{a} = \mathbf{b} + \alpha \cdot \mathbf{x} \bmod 2^\ell$.
4. If P_j is corrupted, wait for the adversary to input a meesage (Guess, S), where S efficiently describes a subset of $\{0, 1\}^s$. If $\alpha \in S$, then send $(\mathsf{Success})$ to $\mathcal{S}$. Otherwise, abort and terminate.
5. Output $\mathbf{a}$ to P_i and $\mathbf{b}$ to P_j.

Figure A.4.: Vector Oblivious Linear Function Evaluation Functionality [39]

Functionality $\mathcal{F}^{\text{RanBitGen}}$

The functionality $\mathcal{F}^{\text{RanBitGen}}$ has all the same features as $\mathcal{F}^{\text{MAC}}$, with the additional command:
Random Bit Generation: Upon receiving $(\text{RanBitGen}, \text{S}_j, \text{sid})$ from $\text{S}_j \in \{\text{S}_0, \text{S}_1\}$:

1. Wait to receive $b^c \in \mathbb{Z}_{2^{k+s}}$ from the adversary, sample random $b^{c-1} \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, such that $b = b^c + b^{c-1} \bmod 2^k$ and $b \in \{0, 1\}$.
2. Send b^j to S_j.
3. Run $\text{MACGen}(b)$. Send m_b^j to S_j.

Figure A.5.: Random Bit Generation Functionality

Functionality $\mathcal{F}^{\text{SqGen}}$

The functionality $\mathcal{F}^{\text{SqGen}}$ has all the same features as $\mathcal{F}^{\text{InCom}}$, with the additional command:
Square Correlation Generation: Upon receiving $(\text{SqCoGen}, \text{P}_i, \text{sid})$ from $\text{P}_i \in \{\text{C}_i, \text{S}_0, \text{S}_1\}$ (or $\text{P}_i \in \{\text{S}_0, \text{S}_1\}$ in 2PC):

1. If $\text{P}_c \in \{\text{S}_0, \text{S}_1\}$, wait to receive $(a^c, d^c) \in (\mathbb{Z}_{2^{k+s}}, \mathbb{Z}_{2^{k+s}})$ from the adversary, sample random $a^{c-1} \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$.

*2. If C_i is corrupted, wait to receive (a^0, a^1, d^0) from the adversary.

3. Compute $a = a^c + a^{c-1} \bmod 2^k$. Compute $d = a \cdot a \bmod 2^k$.
4. Sample $r \xleftarrow{\$} \mathbb{Z}_{2^s}$, compute $d = d + 2^k \cdot r \mod 2^{k+s}$, set $d^{c-1} = d - d^c \bmod 2^{k+s}$.
5. Send (a^j, d^j) to S_j.
6. Run $\text{MACGen}(\{a, d\})$. Send (m_a^j, m_d^j) to S_j.

Figure A.6.: Square Correlation Generation Functionality

A.1.8. Correlated Randomness Functionality $\mathcal{F}^{\text{CR}}$

See Fig. A.8.

Functionality $\mathcal{F}^{\text{Wrap}}$

Initialize: Same as $\mathcal{F}^{\text{InCom}}$.
Macro $\text{MACGen}(\mathbf{x})$: Same as $\mathcal{F}^{\text{InCom}}$.
InCom: Same as $\mathcal{F}^{\text{InCom}}$.
Square Correlation Generation: Same as $\mathcal{F}^{\text{SqGen}}$.
Random Bit Generation: Same as $\mathcal{F}^{\text{RanBitGen}}$.
Bit Triple Generation: Same as $\mathcal{F}^{\text{TripGen}}$.

Figure A.7.: Wrapper Functionality

Functionality $\mathcal{F}^{\text{CR}}$

1. If S_j is corrupted, wait to receive k from the adversary. Otherwise, randomly choose k.
2. Send k to C_i.
3. Upon receiving $(\text{CRGen}, \text{ctr}, \mathsf{P}_i, \text{sid})$ from $\mathsf{P}_i \in \{\mathsf{C}_i, \mathsf{S}_j\}$, compute $r \leftarrow \text{PRF}_k(\text{ctr})$, send r to all P_i.

Figure A.8.: Correlated Randomness Functionality

Functionality $\mathcal{F}^{\text{CR,glo}}$

1. If $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$ is corrupted, wait to receive k from the adversary. Otherwise, randomly choose k.
2. Send k to $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$.
3. Upon receiving $(\text{CRGen}, \text{ctr}, \mathsf{P}_i, \text{sid})$ from $\mathsf{P}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}, \mathsf{S}_j\}$, compute $r \leftarrow \text{PRF}_k(\text{ctr})$, send r to all P_i.

Figure A.9.: "Global" Correlated Randomness Functionality

A.1.9. Correlated Randomness Functionality $\mathcal{F}^{\text{CR,glo}}$

See Fig. A.9.

A.2. Protocols

A.2.1. Batch Check

See Fig. A.10.

A.2.2. Single Check

See Fig. A.11.

BatchCheck

Open: To open a value x_h:

1. S_j samples $r_h^j \xleftarrow{\$} \mathbb{Z}_{2^s}^t$, then call $\mathcal{F}^{\text{MAC}}$ to obtain $[\![r_h]\!]^j$.
2. Servers then compute $[\![\tilde{x}_h]\!] = [\![x_h]\!] + 2^k \cdot [\![r_h]\!]$. We denote S_j's share and MAC share on $\tilde{x}_h$ as $\tilde{x}_h^j$ and m_h^j.
3. S_j sends $\tilde{x}_h^j$ to S_{j-1} and reconstruct $\tilde{x}_h$.

MAC Check (in Batch):

1. Servers call $\mathcal{F}^{\text{Rand}}$, receive $\mathbf{r} \in \mathbb{Z}_{2^s}^t$.
2. Servers then compute $v = \sum_{h=0}^{t-1} r_h \cdot \tilde{x}_h \bmod 2^{k+s}$.
3. S_j computes $\tilde{m}^j = \sum_{h=0}^{t-1} r_h \cdot m_h^j \bmod 2^{k+s}$ and $z^j = \tilde{m}^j - \alpha^j \cdot v \bmod 2^{k+s}$.
4. S_j commits and opens z^j, then verifies if $z = z^0 + z^1 \bmod 2^{k+s}$ is zero. If the check passes, parties accept $x_h = \tilde{x} \bmod 2^k$, otherwise they abort.

Figure A.10.: BatchCheck Procedure [39]

SingleCheck

1. To open $[\![y]\!]$, servers run **Open** phase in **BatchCheck**, receive $\tilde{y}$. We denote S_j's MAC share on $\tilde{y}$ as m^j.
2. S_j computes $z^j = m^j - \alpha^j \cdot \tilde{y} \bmod 2^{k+s}$.
3. S_j commits and opens z^j, then verifies if $z = z^0 + z^1 \bmod 2^{k+s}$ is zero. If the check passes, parties accept $y = \tilde{y} \bmod 2^k$, otherwise they abort.

Figure A.11.: SingleCheck Procedure [39]

Protocol $\Pi^{\text{RanBitGen}}$

Output: Servers output $[b]$, where $b \in \mathbb{Z}_2$.
Protocol:
In the following, parties use an instance of SPD$\mathbb{Z}_{2^k}$ over $\mathbb{Z}_{2^{k+2}}$ with MAC shares over $\mathbb{Z}_{2^{k+s+1}}$.

1. S_j sample $u^j \xleftarrow{\$} \mathbb{Z}_{2^{k+2}}$.
2. S_j call $\mathcal{F}^{\text{MAC}}$ with u^j as input, receives $[\![u]\!]^j$.
3. Servers compute $[\![a]\!] = 2 \cdot [\![u]\!] + 1$.
4. Servers compute $[\![e]\!] = [\![a]\!] \cdot [\![a]\!]$.
5. Servers run **Open** and **MAC check** to obtain e, abort if a is not odd.
6. Let c be the smallest square root modulo 2^{k+2} of e and let c^{-1} be its inverse modulo 2^{k+2}. Servers compute $[\![d]\!] \leftarrow c^{-1}[\![a]\!] + 1$.
7. Let $(d^j, m_d^j) \in (\mathbb{Z}_{2^{k+s+1}}, \mathbb{Z}_{2^{k+s+1}})$ be S_j's share of d and of its MAC. S_j sets $b^j \leftarrow \frac{d^j}{2} \bmod 2^{k+s}$ and $m_b^j \leftarrow \frac{m_d^j}{2} \bmod 2^{k+s}$.
8. S_j outputs $[b]^j := (b^j, m_b^j)$.

Figure A.12.: Authenticated Random Bit Generation $\Pi^{\text{RanBitGen}}$ [46]

A.2.3. Protocol $\Pi^{\text{RanBitGen}}$

See Fig. A.12.

A.2.4. Protocol Π^{MSB}

See Fig. A.13.

Protocol Π^{MSB}

Private input: Servers hold $[x]$.
Output: Servers output $[s]_2$, where $s = 1$ if $x < 0$ and $s = 0$ otherwise.
Preprocessing:

1. Servers send $(\mathsf{RanBitGen}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{RanBitGen}}$, receive $(\llbracket a \rrbracket, \llbracket b_0 \rrbracket, ..., \llbracket b_{k-1} \rrbracket)$, where $a, b_i \bmod 2^k \in \{0, 1\}$.
2. Servers compute $\llbracket r \rrbracket = \sum_{i=0}^{k-1} 2^i \cdot \llbracket b_i \rrbracket$.

Protocol:

1. Servers run **Open** and **Batch check** to reconstruct $c \leftarrow \llbracket a \rrbracket + \llbracket r \rrbracket$.
2. Servers compute $c' = c \bmod 2^{k-1}$ and $\llbracket r' \rrbracket = \sum_{i=0}^{k-2} 2^i \cdot \llbracket b_i \rrbracket$.
3. Servers run Π^{A2B} with $(\llbracket b_0 \rrbracket, ..., \llbracket b_{k-2} \rrbracket)$ as input, receive $([b_0]_2, ..., [b_{k-2}]_2)$ as output.
4. Servers run Π^{BitLT} with $(c', \llbracket b_0 \rrbracket_2, ..., \llbracket b_{k-2} \rrbracket_2)$ as input, receive $\llbracket p \rrbracket_2$ as output.
5. Servers run Π^{B2A} with $\llbracket p \rrbracket_2$ as input, receive $\llbracket p \rrbracket$ as output.
6. Servers compute $\llbracket x' \rrbracket = c' - \llbracket r' \rrbracket + 2^{k-1} \cdot \llbracket p \rrbracket$ and $\llbracket d \rrbracket \leftarrow \llbracket x \rrbracket - \llbracket x' \rrbracket$.
7. Servers run **Open** and **Batch check** to reconstruct $e \leftarrow \llbracket d \rrbracket + 2^{k-1} \cdot \llbracket a \rrbracket$.
8. Let e_{k-1} be the most significant bit of e. Servers output $\llbracket s \rrbracket_2 \leftarrow e_{k-1} + \llbracket a \rrbracket - 2e_{k-1} \cdot \llbracket a \rrbracket$.

Figure A.13.: Extract MSB protocol Π^{MSB} [46]. Within Π^{MSB}, the A2B protocol Π^{A2B}, the bitwise comparison protocol Π^{BitLT} and B2A protocol Π^{B2A} can be found in [46].

A.3. Security Proof for Force

A.3.1. Security of Π_{CMSGen}

For the following proof, we suppose that $\psi\text{to}\phi$ = AConvertToAB. We first assume that P_0 is corrupted.

Proof. We construct an adversary $\mathcal{S}$ for the ideal model such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol Π_{CMSGen} or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{CMSGen}}$. $\mathcal{S}$ simulates a real execution in which the corrupted party P_0 controlled by $\mathcal{A}$ delivers message to uncorrupted parties in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ emulates an honest P_1 and P_3, receives $k^0_{\mathsf{AConvertToAB}}$ from $\mathcal{A}$.
2. $\mathcal{S}$ emulates an honest P_2, receives $k^1_{\mathsf{AConvertToAB}}$ from $\mathcal{A}$.

Functionality $\mathcal{F}_{\mathsf{4PC}}^{\mathsf{CMSGen}}$

If $\psi\text{to}\phi$ = AConvertToAB:

- If P_0 is corrupted:
  - Wait to receive $k^0_{\mathsf{AConvertToAB}}, k^1_{\mathsf{AConvertToAB}}$ from the adversary.
  - Upon receiving $(\mathsf{CMSGen}, \mathsf{ctr}, \mathsf{P}_i, \mathsf{sid})$ from all P_i, computes $r_0 = \mathsf{PRF}_{k^0_{\mathsf{AConvertToAB}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^1_{\mathsf{AConvertToAB}}}(\mathsf{ctr})$. Then choose $r_1 \xleftarrow{\$} \mathbb{Z}_{2^k}$ and set $r = r_0 + r_1 \bmod 2^k$.
- If P_1 is corrupted:
  - Wait to receive $k^2_{\mathsf{AConvertToAB}}$ from the adversary, choose and send $k^0_{\mathsf{AConvertToAB}} \xleftarrow{\$} \{0,1\}^\kappa$ to the adversary.
  - Upon receiving $(\mathsf{CMSGen}, \mathsf{ctr}, \mathsf{P}_i, \mathsf{sid})$ from all P_i, computes $r = \mathsf{PRF}_{k^2_{\mathsf{AConvertToAB}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^0_{\mathsf{AConvertToAB}}}(\mathsf{ctr})$. Then choose $r_0 \xleftarrow{\$} \mathbb{Z}_{2^k}$ and set $r_1 = r - r_0 \bmod 2^k$.
- If P_2 is corrupted:
  - Choose $k^1_{\mathsf{AConvertToAB}} \xleftarrow{\$} \{0,1\}^\kappa$ and $k^2_{\mathsf{AConvertToAB}} \xleftarrow{\$} \{0,1\}^\kappa$, send $k^1_{\mathsf{AConvertToAB}}$ and $k^2_{\mathsf{AConvertToAB}}$ to the adversary.
  - Upon receiving $(\mathsf{CMSGen}, \mathsf{ctr}, \mathsf{P}_i, \mathsf{sid})$ from all P_i, computes $r_1 = \mathsf{PRF}_{k^1_{\mathsf{AConvertToAB}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^2_{\mathsf{AConvertToAB}}}(\mathsf{ctr})$. Then choose $r_0 \xleftarrow{\$} \mathbb{Z}_{2^k}$ and set $r = r_0 + r_1 \bmod 2^k$.
- If P_3 is corrupted:
  - Choose $k^0_{\mathsf{AConvertToAB}} \xleftarrow{\$} \{0,1\}^\kappa$ and $k^2_{\mathsf{AConvertToAB}} \xleftarrow{\$} \{0,1\}^\kappa$, send $k^0_{\mathsf{AConvertToAB}}$ and $k^2_{\mathsf{AConvertToAB}}$ to the adversary.
  - Upon receiving $(\mathsf{CMSGen}, \mathsf{ctr}, \mathsf{P}_i, \mathsf{sid})$ from all P_i, computes $r = \mathsf{PRF}_{k^2_{\mathsf{AConvertToAB}}}(\mathsf{ctr}) - \mathsf{PRF}_{k^0_{\mathsf{AConvertToAB}}}(\mathsf{ctr})$. Then choose $r_0 \xleftarrow{\$} \mathbb{Z}_{2^k}$ and set $r_1 = r - r_0 \bmod 2^k$.
- Send r_0 to P_0, r_1 to P_2, r to P_1 and P_3.

Figure A.14.: Four-Party Changing-Mode Sharing Generation Functionality (Part 1)

Functionality $\mathcal{F}_{\text{4PC}}^{\text{CMSGen}}$

If ψtoϕ = ABtoAC:

- If P_0 is corrupted:
  - Wait to receive $k^0_{\text{ABtoAC}}, k^1_{\text{ABtoAC}}$ from the adversary.
  - Upon receiving $(\text{CMSGen}, \text{ctr}, \mathsf{P}_i, \text{sid})$ from all P_i, computes $r_0 = \text{PRF}_{k^0_{\text{ABtoAC}}}(\text{ctr}) - \text{PRF}_{k^1_{\text{ABtoAC}}}(\text{ctr})$. Then choose $r_1 \xleftarrow{\$} \mathbb{Z}_{2^k}$ and set $r = r_0 + r_1 \bmod 2^k$.
- If P_1 is corrupted:
  - Wait to receive k^2_{ABtoAC} from the adversary, choose and send $k^1_{\text{ABtoAC}} \xleftarrow{\$} \{0,1\}^\kappa$ to the adversary.
  - Upon receiving $(\text{CMSGen}, \text{ctr}, \mathsf{P}_i, \text{sid})$ from all P_i, computes $r_1 = \text{PRF}_{k^1_{\text{ABtoAC}}}(\text{ctr}) - \text{PRF}_{k^2_{\text{ABtoAC}}}(\text{ctr})$. Then choose $r_0 \xleftarrow{\$} \mathbb{Z}_{2^k}$ and set $r = r_0 + r_1 \bmod 2^k$.
- If P_2 or P_3 is corrupted:
  - Choose $k^0_{\text{ABtoAC}} \xleftarrow{\$} \{0,1\}^\kappa$ and $k^2_{\text{ABtoAC}} \xleftarrow{\$} \{0,1\}^\kappa$, send k^0_{ABtoAC} and k^2_{ABtoAC} to the adversary.
  - Upon receiving $(\text{CMSGen}, \text{ctr}, \mathsf{P}_i, \text{sid})$ from all P_i, computes $r = \text{PRF}_{k^2_{\text{ABtoAC}}}(\text{ctr}) - \text{PRF}_{k^0_{\text{ABtoAC}}}(\text{ctr})$. Then choose $r_0 \xleftarrow{\$} \mathbb{Z}_{2^k}$ and set $r_1 = r - r_0 \bmod 2^k$.
- Send r_0 to P_0, r_1 to P_1, r to P_2 and P_3.

Figure A.15.: Four-Party Changing-Mode Sharing Generation Functionality (Part 2)

3. $\mathcal{S}$ sends $(k^0_{\text{ACtoAB}}, k^1_{\text{ACtoAB}})$ to $\mathcal{F}_{\text{4PC}}^{\text{CMSGen}}$.

Since $\mathcal{A}$ only delivers message to other honest parties and the functionality $\mathcal{F}_{\text{4PC}}^{\text{CMSGen}}$ generates $\mathcal{A}$'s output exactly as $\mathcal{A}$ computes in the ideal execution, we conclude that the adversary's view is indistinguishable in the real and ideal execution. It remains to show that the outputs computed by parties P_1, P_2, P_3 are indistinguishable in the both worlds. In the real execution, r_1 is computed as $r_1 = \text{PRF}_{k^1_{\text{ACtoAB}}}(\text{sid}) - \text{PRF}_{k^2_{\text{ACtoAB}}}(\text{sid})$. Since k^2_{ACtoAB} is chosen independently from both k^0_{ACtoAB} and k^1_{ACtoAB}, r_1 generated by a pseudorandom function $\text{PRF}_{k^2_{\text{ACtoAB}}}$ is indistinguishable with the one generated by a real random function in the ideal execution. The reduction is straight forward due to the security of the psudorandom function. Thus we conclude that $\mathbf{REAL}_{\Pi_{\text{CMSGen}}, \mathcal{A}, \mathcal{Z}}$ is indistinguishable from $\mathbf{IDEAL}_{\mathcal{F}_{\text{4PC}}^{\text{CMSGen}}, \mathcal{S}, \mathcal{Z}}$ for the environment $\mathcal{Z}$. □

If ψtoϕ = ABtoAC, the simulator is constructed in a similar way, where it still emulates parties receiving k^0_{ABtoAC} and k^1_{ABtoAC} from $\mathcal{A}$ and sends $\mathcal{A}$'s computed result to $\mathcal{F}_{\text{4PC}}^{\text{CMSGen}}$ as its output. For other corruption cases, the proofs are similar.

Functionality $\mathcal{F}_{\text{4PC}}^{\text{chMo}}$

- Upon receiving $(\text{chMode}, [x]_{\psi}^{\mathsf{P}_c}, \mathsf{P}_i, \text{sid})$ from $\mathsf{P}_i \in \mathcal{P}$:
  - Wait to receive $[x]_{\phi}^{\mathsf{P}_c}$ from the adversary.
  - Let P_{p} denote the reconstruction partner of P_c regarding share-mode ψ. Suppose $[x]_{\phi}^{\mathsf{P}_c} = r_0$, compute $r_1 = [x]_{\psi}^{\mathsf{P}_c} + [x]_{\psi}^{\mathsf{P}_{\mathsf{p}}} - r_0 \bmod 2^k$.
  - Assign (r_0, r_1) appropriately to each party's output regarding ϕ. Send $[x]_{\phi}^{\mathsf{P}_i}$ to each $\mathsf{P}_i \in \mathcal{P}$.

Figure A.16.: Four-Party Change Share-Mode Functionality

A.3.2. Security of Π_{chMo}

For the following proof, we suppose that $\psi = \mathsf{AC}$ and $\phi = \mathsf{AB}$. We first assume that P_0 is corrupted.

Proof. We construct an adversary $\mathcal{S}$ for the ideal model such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol Π_{chMo} or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\text{4PC}}^{\text{chMo}}$. $\mathcal{S}$ simulates a real execution in which the corrupted party P_0 controlled by $\mathcal{A}$ delivers message to uncorrupted parties in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{4PC}}^{\text{Pre}}$, samples and sends $[\![r]\!]_{\text{ACtoAB}}^{\mathsf{P}_0}$ to $\mathcal{A}$.
2. $\mathcal{S}$ emulates P_2, receives d_0 from $\mathcal{A}$.
3. $\mathcal{S}$ samples $d_1 \xleftarrow{\$} \mathbb{Z}_{2^k}$, sends d_1 to $\mathcal{A}$ as an honest P_2.
4. $\mathcal{S}$ sends $[x]_{\text{AC}}^{\mathsf{P}_0}$ to $\mathcal{F}_{\text{4PC}}^{\text{chMo}}$, it also sends $[x]_{\text{AB}}^{\mathsf{P}_0} = d_0 + d_1 \bmod 2^k$ to $\mathcal{F}_{\text{4PC}}^{\text{chMo}}$.

In the real execution, P_2 computes d_2 as $d_2 = [x]_{\psi}^{\mathsf{P}_2} + [\![r]\!]_{\text{ACtoAB}}^{\mathsf{P}_2} \bmod 2^k$, while d_2 is sampled uniformly at random in the ideal execution. Since $[\![r]\!]_{\text{ACtoAB}}^{\mathsf{P}_2}$ is distributed uniformly at random to $\mathcal{Z}$, so is the computed (masked) value d_2. Besides, $[x]_{\text{AB}}^{\mathsf{P}_0}$ computed by $\mathcal{A}$ is set to the exact $\mathcal{A}$'s output, which serves as a random mask to x satisfying $x = [x]_{\text{AB}}^{\mathsf{P}_0} + [x]_{\text{AB}}^{\mathsf{P}_1} \bmod 2^k$ just as in the real execution. Thus we conclude that $\mathbf{REAL}_{\Pi_{\text{CMSGen}}, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\text{4PC}}^{\text{Pre}}}$ is indistinguishable from $\mathbf{IDEAL}_{\mathcal{F}_{\text{4PC}}^{\text{CMSGen}}, \mathcal{S}, \mathcal{Z}}^{\mathcal{F}_{\text{4PC}}^{\text{Pre}}}$ for the environment $\mathcal{Z}$. □

For other corruption cases and share-modes, the proofs are similar.

Functionality $\mathcal{F}_{\text{4PC}}^{\text{Mult}}$

- Upon receiving $(\text{Mult}, [x]_{\psi}^{\text{P}_c}, [y]_{\phi}^{\text{P}_c}, \theta, \text{P}_c, \text{sid})$ from P_c and other P_j, where $\text{P}_j \in \mathcal{P} \backslash \text{P}_c$:
  - Wait to receive $[z]_{\theta}^{\text{P}_c}$ from $\mathcal{S}$.
  - Let P_{par} denote the reconstruction party of P_c regarding $[x]_{\psi}$ and P'_{par} denote the reconstruction party regarding $[y]_{\phi}$. Reconstruct $x = [x]_{\psi}^{\text{P}_c} + [x]_{\psi}^{\text{P}_{\text{par}}} \bmod 2^k$ and $y = [y]_{\psi}^{\text{P}_c} + [y]_{\psi}^{\text{P}'_{\text{par}}} \bmod 2^k$.
  - Compute $z = xy \bmod 2^k$. Suppose $[z]_{\theta}^{\text{P}_c} = r_0$, compute $r_1 = z - r_0 \bmod 2^k$.
  - Assign (r_0, r_1) appropriately to each party's output regarding θ. Send $[z]_{\theta}^{\text{P}_i}$ to each $\text{P}_i \in \mathcal{P}$.

Figure A.17.: Four-Party Multiplication Functionality

A.3.3. Security of Π_{Mult}

We first discuss the case if the share-modes of both $\mathcal{X}$-shared inputs are different. For the following proof, we suppose that $\psi = \text{AC}$, $\phi = \text{AB}$ and $\theta = \text{AC}$. Again, we begin by assuming that P_0 is corrupted.

Proof. We construct an adversary $\mathcal{S}$ for the ideal model such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol Π_{Mult} or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\text{4PC}}^{\text{Mult}}$. $\mathcal{S}$ simulates a real execution in which the corrupted party P_0 controlled by $\mathcal{A}$ delivers message to uncorrupted parties in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{4PC}}^{\text{Pre}}$, samples and sends $[\![0]\!]_{\text{4o4}}^{\text{P}_0}$ to $\mathcal{A}$.
2. $\mathcal{S}$ emulates P_1, receives e^{P_0} from $\mathcal{A}$.
3. $\mathcal{S}$ samples $e^{\text{P}_1} \xleftarrow{\$} \mathbb{Z}_{2^k}$, sends e^{P_1} to $\mathcal{A}$ as an honest P_2.
4. $\mathcal{S}$ sends $[x]_{\text{AC}}^{\text{P}_0}$, $[x]_{\text{AB}}^{\text{P}_0}$ to $\mathcal{F}_{\text{4PC}}^{\text{Mult}}$, it also sends $[z]_{\text{AC}}^{\text{P}_0} = e^{\text{P}_0} + e^{\text{P}_1} \bmod 2^k$ to $\mathcal{F}_{\text{4PC}}^{\text{Mult}}$.

In the real execution, P_1 computes e^{P_1} as $e^{\text{P}_1} = [x]_{\text{AC}}^{\text{P}_1}[y]_{\text{AB}}^{\text{P}_1} + [0]_{\text{4o4}}^{\text{P}_1} \bmod 2^k$, while e^{P_1} is sampled uniformly at random in the ideal execution. Since $[0]_{\text{4o4}}^{\text{P}_1}$ is distributed uniformly at random to $\mathcal{Z}$, so is the computed (masked) value d_2. Besides, $[z]_{\text{AC}}^{\text{P}_0}$ computed by $\mathcal{A}$ is set to the exact $\mathcal{A}$'s output, which serves as a random mask to x satisfying $z = [z]_{\text{AC}}^{\text{P}_0} + [z]_{\text{AC}}^{\text{P}_2} \bmod 2^k$ just as in the real execution. Thus we conclude that $\mathbf{REAL}_{\Pi_{\text{CMSGen}}, \mathcal{A}, \mathcal{Z}}^{\mathcal{F}_{\text{4PC}}^{\text{Pre}}}$ is indistinguishable from $\mathbf{IDEAL}_{\mathcal{F}_{\text{4PC}}^{\text{CMSGen}}, \mathcal{S}, \mathcal{Z}}^{\mathcal{F}_{\text{4PC}}^{\text{Pre}}}$ for the environment $\mathcal{Z}$. □

We now discuss the second case, where the share-modes of two $\mathcal{X}$-shared inputs are the same. We suppose that $\psi = \phi = \text{AC}$ and $\theta = \text{AC}$. We still assume that P_0 is corrupted.

Proof. We construct an adversary $\mathcal{S}$ for the ideal model such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol $\Pi^{\mathsf{chMo}}_{\mathsf{Mult}}$ or with $\mathcal{S}$ in the ideal process for $\mathcal{F}^{\mathsf{Mult}}_{\mathsf{4PC}}$. $\mathcal{S}$ simulates a real execution in which the corrupted party P_0 controlled by $\mathcal{A}$ delivers message to uncorrupted parties in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ emulates the functionality $\mathcal{F}^{\mathsf{Pre}}_{\mathsf{4PC}}$, samples and sends $[\![0]\!]^{\mathsf{P}_0}_{\mathsf{4o4}}$ to $\mathcal{A}$.
2. $\mathcal{S}$ emulates the functionality $\mathcal{F}^{\mathsf{chMo}}_{\mathsf{4PC}}$, receives $(\mathsf{chMode}, [y]^{\mathsf{P}_0}_{\mathsf{AC}}, \mathsf{P}_0, \mathsf{sid})$ from $\mathcal{A}$, sends $[y]^{\mathsf{P}_0}_{\mathsf{AB}}$ to $\mathcal{A}$.
3. $\mathcal{S}$ emulates P_1, receives e^{P_0} from $\mathcal{A}$.
4. $\mathcal{S}$ samples $e^{\mathsf{P}_1} \xleftarrow{\$} \mathbb{Z}_{2^k}$, sends e^{P_1} to $\mathcal{A}$ as an honest P_2.
5. $\mathcal{S}$ sends $[x]^{\mathsf{P}_0}_{\mathsf{AC}}$, $[x]^{\mathsf{P}_0}_{\mathsf{AC}}$ to $\mathcal{F}^{\mathsf{Mult}}_{\mathsf{4PC}}$, it also sends $[z]^{\mathsf{P}_0}_{\mathsf{AC}} = e^{\mathsf{P}_0} + e^{\mathsf{P}_1} \bmod 2^k$ to $\mathcal{F}^{\mathsf{Mult}}_{\mathsf{4PC}}$.

In the real execution, P_1 uses $[0]^{\mathsf{P}_1}_{\mathsf{4o4}}$ to mask the computed result $[x]^{\mathsf{P}_1}_{\mathsf{AC}}[y]^{\mathsf{P}_1}_{\mathsf{AB}} \bmod 2^k$, where $[y]^{\mathsf{P}_1}_{\mathsf{AB}}$ is the received result from $\mathcal{F}^{\mathsf{chMo}}_{\mathsf{4PC}}$. The rest of the proof is essentially the same as the previous one. □

We emphasize that the protocol steps executed by each party are symmetric. In the case of other corruption scenarios, we can construct a simulator $\mathcal{S}$ in a similar manner to the approach described above, simulating the behavior of the honest parties receiving and sending message to $\mathcal{A}$. The Indistinguishability of two executions follows from the fact that all exchanged messages are masked using a four-party zero sharing.

A.4. Security Proof for NodeGuard

We first consider the case where P_{1-h} is corrupted.

Proof. We construct an adversary $\mathcal{S}$ for the ideal model such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol Π_{KeyBuc} or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{BucAgg}}$. $\mathcal{S}$ simulates a real execution in which the corrupted party P_{1-h} controlled by $\mathcal{A}$ delivers message to uncorrupted parties in the internal (simulated) interaction. The $\mathcal{S}$ works as follows (at any node j):

- Preprocessing: $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}$, waits to receive $\mathbf{k}_j$, $[m_j^{f,b}]_{1-h}$ from $\mathcal{A}$.
- If $j = 1$: $\mathcal{S}$ samples $\mathbf{v}^{f,b} \xleftarrow{\$} \mathbb{Z}_{2^k}^N$ and sends $\mathbf{v}^{f,b}$ to $\mathcal{A}$ as an honest P_h.
- At any node j:
  1. $\mathcal{S}$ emulates an honest P_h, receives $\mathbf{q}_j$ from $\mathcal{A}$.
  2. $\mathcal{S}$ computes $\mathcal{A}$'s input as $[\mathbf{g}_j]_{1-h} = \mathbf{q}_j - \mathbf{k}_j \bmod 2^k$.
  3. $\mathcal{S}$ computes $[z_j^{f,b}]_{1-h}$ as $\mathcal{A}$ will do and sends $[\mathbf{g}_j]_{1-h}$, $[z_j^{f,b}]_{1-h}$ to $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{BucAgg}}$.

In the real execution, P_h computes $\mathbf{v}^{f,b}$ as $\mathbf{v}^{f,b} = \mathbf{s}^{f,b} - \mathbf{r}^{f,b} \bmod 2^k$, while $\mathbf{v}^{f,b}$ is sampled uniformly at random in the ideal execution. Since $\mathbf{r}^{f,b}$ is distributed uniformly at random to $\mathcal{Z}$, so is the computed (masked) value $\mathbf{v}^{f,b}$. Besides, $[z_j^{f,b}]_{1-h}$ computed by $\mathcal{A}$ is set to the exact $\mathcal{A}$'s output, which serves as a random mask for $z_j^{f,b}$ satisfying $z_j^{f,b} = [z_j^{f,b}]_h + [z_j^{f,b}]_{1-h} \bmod 2^k$ just as in the real execution. Thus we conclude that $\mathbf{REAL}_{\Pi_{\mathsf{KeyBuc}},\mathcal{A},\mathcal{Z}}^{\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}}$ is indistinguishable from $\mathbf{IDEAL}_{\mathcal{F}_{\mathsf{2PC}}^{\mathsf{BucAgg}},\mathcal{S},\mathcal{Z}}^{\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}}$ for the environment $\mathcal{Z}$. □

We now consider the case in which P_h is corrupted.

Proof. We construct an adversary $\mathcal{S}$ for the ideal model such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol Π_{KeyBuc} or with $\mathcal{S}$ in the ideal process for $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{BucAgg}}$. $\mathcal{S}$ simulates a real execution in which the corrupted party P_h controlled by $\mathcal{A}$ delivers message to uncorrupted parties in the internal (simulated) interaction. The $\mathcal{S}$ works as follows (at any node j):

- Preprocessing:
  1. If $j = 1$, $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{2PC}}^{\mathsf{KBucGen}}$, waits to receive $\mathbf{r}^{f,b}$ from $\mathcal{A}$.
  2. $\mathcal{S}$ waits to receive $[m_j^{f,b}]_h$ from $\mathcal{A}$.
- If $j = 1$:
  1. $\mathcal{S}$ emulates an honest P_{1-h}, receives $\mathbf{v}^{f,b}$ from $\mathcal{A}$.

2. $\mathcal{S}$ computes $\mathcal{A}$'s input as $\mathbf{s}^{f,b} = \mathbf{v}^{f,b} - \mathbf{r}^{f,b} \bmod 2^k$. It translates $\mathbf{s}^{f,b}$ to $\mathcal{I}^{f,b}$ and sends $\mathcal{I}^{f,b}$ to $\mathcal{F}_{\text{2PC}}^{\text{BucAgg}}$.

- At any node j:

1. $\mathcal{S}$ samples $\mathbf{q}_j \xleftarrow{\$} \mathbb{Z}_{2^k}^N$ and sends $\mathbf{q}_j$ to $\mathcal{A}$ as an honest P_{1-h}.

2. $\mathcal{S}$ computes $[z_j^{f,b}]_h$ as $\mathcal{A}$ will do and sends $[\mathbf{g}_j]_h$, $[z_j^{f,b}]_h$ to $\mathcal{F}_{\text{2PC}}^{\text{BucAgg}}$.

Note that $\mathcal{A}$'s input $[\mathbf{g}_j]_h$ is extracted by $\mathcal{S}$, when $\mathcal{A}$ plays the role of P_{1-h}. In the real execution, P_{1-h} computes $\mathbf{q}_j$ as $\mathbf{q}_j = [\mathbf{g}_j]_{1-h} - \mathbf{k}_j \bmod 2^k$, while $\mathbf{q}_j$ is sampled uniformly at random in the ideal execution. Since $\mathbf{k}_j$ is distributed uniformly at random to $\mathcal{Z}$, so is the computed (masked) value $\mathbf{q}_j$. Besides, $[z_j^{f,b}]_h$ computed by $\mathcal{A}$ is set to the exact $\mathcal{A}$'s output, which serves as a random mask for $z_j^{f,b}$ satisfying $z_j^{f,b} = [z_j^{f,b}]_h + [z_j^{f,b}]_{1-h} \bmod 2^k$ just as in the real execution. Thus we conclude that $\mathbf{REAL}_{\Pi_{\text{KeyBuc}},\mathcal{A},\mathcal{Z}}^{\mathcal{F}_{\text{2PC}}^{\text{KBucGen}}}$ is indistinguishable from $\mathbf{IDEAL}_{\mathcal{F}_{\text{2PC}}^{\text{BucAgg}},\mathcal{S},\mathcal{Z}}^{\mathcal{F}_{\text{2PC}}^{\text{KBucGen}}}$ for the environment $\mathcal{Z}$. □

A.5. Security Proof for AlphaFL

Regardless of the servers, we assume that there exists at least one honest client.

A.5.1. Lemma 1 in [39]

Lemma A.5.1. Let ℓ, r and m be positive integers such at $\ell - r \leq m$. Let $\delta_0, \delta_1, ..., \delta_t \in \mathbf{Z}$ and suppose that not all the δ_i are zero modulo 2^r. for $i > 0$. Let Y be a probability distribution of $\mathbb{Z}$. Then, if the distribution Y is independent from the uniform distribution sampling α below, we have

$$\Pr_{\alpha, r_1, ..., r_t \xleftarrow{\$} \mathbb{Z}_{2^m}, y \xleftarrow{\$} Y} \left[y = \alpha \cdot \left(\delta_0 + \sum_{i=1}^{t} r_i \cdot \delta_i \right) \bmod 2^\ell \right] \leq 2^{-\ell + r + \log(\ell - r + 1)}.$$

A.5.2. Security of Π^{lnCom}

Let $\mathcal{A}$ be a malicious, static adversary that interacts with parties performing the protocol Π^{lnCom}. We construct an adversary $\mathcal{S}$ for the ideal model such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol $\mathcal{F}^{\mathsf{lnCom}}$ or with $\mathcal{S}$ in the ideal process for $\mathcal{F}^{\mathsf{lnCom}}$.

Simulating the case where S_0 is corrupted: $\mathcal{S}$ simulates a real execution in which the corrupted S_0 controlled by $\mathcal{A}$ delivers messages to uncorrupted S_1 and C_i in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

Initialize: Emulate $\mathcal{F}^{\mathsf{CR,glo}}$, generate (α^0, α^1), send α^0 to $\mathcal{A}$ and $\mathcal{F}^{\mathsf{lnCom}}$.
Protocol: Then for each C_i, emulate $\mathcal{F}^{\mathsf{CR}}$, generate and send $\mathbf{x}_i^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}^t$, $x_{i,t}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$ $\mathbf{m}_i^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}^t$ and $m_{i,t}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$ to $\mathcal{A}$.
Consistency Check:

1. Emulate $\mathcal{F}^{\mathsf{Rand}}$, generate and send $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_{2^s}^{nt}$ to $\mathcal{A}$.
2. Act as an honest S_1, receive $\hat{v}^0$ from $\mathcal{A}$. Randomly sample and send v^1 to $\mathcal{A}$.
3. Compute v^0 and z^0 just as $\mathcal{A}$ will do. Set $z^1 = (\hat{v}^0 - v^0) \cdot \alpha^1 - z^0 \bmod 2^{k+2s}$.
4. Commit and send z^1 to $\mathcal{A}$, receive commitment z^0 from $\mathcal{A}$.
5. Check if $z^0 + z^1 = 0 \bmod 2^{k+2s}$, abort as an honest S_1 if not.
6. Otherwise, for each C_i, compute $\mathbf{m}_i^0 \bmod 2^{k+s}$ and send $(\mathbf{x}_i^0, \mathbf{m}_i^0)$ to $\mathcal{F}^{\mathsf{lnCom}}$. Then halt.

Proof. We now prove that $\mathsf{REAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}}}_{\Pi^{\mathsf{lnCom}},\mathcal{A},\mathcal{Z}}$ is indistinguishable from $\mathsf{IDEAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}}}_{\mathcal{F}^{\mathsf{lnCom}},\mathcal{S},\mathcal{Z}}$.

We first prove that the messages received by adversary during the protocol execution are distributed identically in the real and ideal execution. In the real execution $v^1 = \sum_{h=0}^{nt-1} x_h^1 \cdot r_h + \sum_{h=nt}^{nt+t-1} x_h^1 \bmod 2^{k+2s}$ is computed by S_1, while in the ideal execution v^1 is chosen uniformly at random by $\mathcal{S}$. Since $\sum_{h=nt}^{nt+t-1} x_h^1$ is distributed uniformly at random to $\mathcal{A}$, so is the masked value v^1. Note that the consistency check should always be passed since C_i is honest (in the honest-majority setting). Thus, any error $e = \hat{v}^0 - v^0$ introduced by $\mathcal{A}$ will cause S_1 to open a commitment with difference $(\hat{v}^0 - v^0) \cdot \alpha^1$ in the real world, which is perfectly simulated by the simulator. The above concludes the identical distribution of messages in the real and ideal execution.

It is easy to see that the probability of passing the consistency check in both executions is identical. In fact, the honest S_1 already receives the correctly computed MAC shares. It remains to argue that the MAC shares output by all parties are identically distributed in both executions. In both executions, $\mathcal{A}$ receives MAC shares from from $\mathcal{F}^{\mathsf{CR}}$, which are chosen uniformly at random. Then in the real execution, since C_i is honest, it first computes the correct MACs then subtract the MACs by the shares received from $\mathcal{F}^{\mathsf{CR}}$ and set the result to S_1's MAC shares. In the ideal execution, $\mathcal{S}$ sends $\mathcal{A}$'s MAC shares to $\mathcal{F}^{\mathsf{lnCom}}$, which sets them to be exactly $\mathcal{A}$'s output. Then $\mathcal{F}^{\mathsf{lnCom}}$ computes the MAC shares of S_1 in the same way as C_i in the real execution, so they are distributed identically in both worlds.

We conclude that the simulation is indistinguishable for $\mathcal{Z}$. □

Simulating the case where $C_c \subseteq \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$ is corrupted: Suppose that there are q corrupted clients. $\mathcal{S}$ simulates a real execution in which the corrupted C_c controlled by $\mathcal{A}$ delivers messages to uncorrupted S_0 and S_1 in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

Initialize: Emulate $\mathcal{F}^{\mathsf{CR,glo}}$, generate (α^0, α^1), send them to $\mathcal{A}$ and $\mathcal{F}^{\mathsf{lnCom}}$.
Protocol: For each corrupted C_i, where $i \in [q]$:

1. Emulate $\mathcal{F}^{\mathsf{CR}}$, generate $(\mathbf{x}_i^0, x_{i,t}^0, \mathbf{m}_i^0, m_{i,t}^0)$ and send them to $\mathcal{A}$.
2. Act as an honest S_1, receive $(\mathbf{x}_i^1, x_{i,t}^1, \mathbf{m}_i^1, m_{i,t}^1)$ from $\mathcal{A}$.

Consistency Check:

3. Perform *Consistency Check* just as honest S_0 and S_1 will do, abort if the consistency check fails.
4. Otherwise, for each corrupted C_i, send $(\mathbf{x}_i^0, \mathbf{x}_i^1)$ and $\mathbf{m}_i^0 \bmod 2^{k+s}$ to $\mathcal{F}^{\mathsf{lnCom}}$. Then halt.

Proof. Since both functionalities $\mathcal{F}^{\mathsf{CR,glo}}$ and $\mathcal{F}^{\mathsf{CR}}$ are emulated by $\mathcal{S}$, $\mathcal{A}$ only sends messages to $\mathcal{S}$ and do not receive any messages from $\mathcal{S}$. Thus, it is clear that the message transcript accessible to an adversary during the protocol is distributed the same way in both the real and ideal executions. Again, since $\mathcal{S}$ uses the shares received from $\mathcal{A}$ to perform the *consistency check* just as S_0 and S_1 do in the real execution, we argue that the probability of passing the consistency check in both the ideal and real execution is identical.

It remains to show that the MAC shares computed in both worlds are identically distributed. From Claim 6.3.1, we know that if the consistency check passes then the parties output correctly generated MAC shares received from $\mathcal{A}$, except with negligible probability. First, we notice that the shares output by S_0 in the real execution are exactly the values it receives from $\mathcal{F}^{\mathsf{CR}}$, which is emulated by $\mathcal{S}$ in the ideal execution and sent to $\mathcal{F}^{\mathsf{lnCom}}$ as S_0's output. Then, S_1 outputs the (correct) MAC shares received from $\mathcal{A}$ in the real execution. These are computed in the same way by $\mathcal{F}^{\mathsf{lnCom}}$ in the ideal execution by subtracting S_0's MAC shares from the correct MACs.

We conclude that the simulation is indistinguishable for $\mathcal{Z}$. □

A.5.3. Security of $\Pi^{\mathsf{lnCom}}_{\mathsf{Diho}}$

Let $\mathcal{A}$ be a malicious, static adversary that interacts with parties performing the protocol $\Pi^{\mathsf{lnCom}}_{\mathsf{Diho}}$ as shown in Fig. 6.6. We construct an adversary $\mathcal{S}$ for the ideal model such that no environment $\mathcal{Z}$ can tell with non-negligible probability whether it is interacting with $\mathcal{A}$ and the protocol $\mathcal{F}^{\mathsf{lnCom}}$ or with $\mathcal{S}$ in the ideal process for $\mathcal{F}^{\mathsf{lnCom}}$.

Simulating the case where $C_c \subseteq \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$ is corrupted: Suppose that there are q corrupted clients. $\mathcal{S}$ simulates a real execution in which the corrupted C_c controlled by $\mathcal{A}$ delivers messages to uncorrupted S_0 and S_1 in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

For each corrupted C_i:

1. Emulate $\mathcal{F}^{\mathsf{CR}}$ instance, sample and send $(\mathbf{x}_i^0, x_{i,t}^0)$ to $\mathcal{A}$.
2. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive $\tilde{\mathbf{x}}_i = (\mathbf{x}'_i, x_{i,t})$[1] and $\mathbf{b}_i^1$ from $\mathcal{A}$ as input to $\mathcal{F}^{\mathsf{vOLE}}$.
3. If $\mathcal{A}$ sends any (Guess, S) message to $\mathcal{F}^{\mathsf{vOLE}}$, forward the guess to $\mathcal{F}^{\mathsf{lnCom}}$. If $\mathcal{F}^{\mathsf{lnCom}}$ aborts, then abort; otherwise, store the set $S_0 = S_0 \cap S$ (where initially $S_0 = \mathbb{Z}_{2^s}$).
4. After delivering output shares as $\mathcal{F}^{\mathsf{vOLE}}$ for all corrupted clients, sample $\alpha^0 \xleftarrow{\$} S_0$ (only once in the entire simulation).
5. Honestly compute $\mathbf{a}_i^1 = \alpha^0 \circ \tilde{\mathbf{x}}_i - \mathbf{b}_i^1 \bmod 2^{k+2s}$.

[1] Since $x_{i,t} \in \mathbb{Z}_{2^{k+2s}}$, we do not denote it as $x'_{i,t}$.

6. Act as an honest S_1, receive $(\mathbf{x}_i^1, x_{i,t}^1, \hat{\mathbf{b}}_i^1)$ from $\mathcal{A}$.
7. Define $\delta_{i,h} = x'_{i,h} - x_{i,h}^0 - x_{i,h}^1$ and $\rho_{i,h} = \hat{b}_{i,h} - b_{i,h}$ for $h \in [t+1]$.

Consistency Check:

7. Check if the following holds:

$$0 = \alpha^0 \cdot (\underbrace{\sum_{h=0}^{qt-1} \delta_h \cdot r_h}_{\theta \cdot \chi} + \underbrace{\sum_{h=qt}^{qt+q-1} \delta_h}_{\theta_{qt}}) + \underbrace{\sum_{h=0}^{qt-1} \rho_h \cdot r_h + \sum_{h=qt}^{qt+q-1} \rho_h}_{e} \bmod 2^{k+2s},$$

   abort if the check fails.
8. Otherwise, for each corrupted C_i, send $(\mathbf{x}_i^0, \mathbf{x}_i^1)$ to $\mathcal{F}^{\mathsf{InCom}}$. Then halt.

Proof. We now prove that $\mathsf{REAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}},\mathcal{F}^{\mathsf{vOLE}}}_{\Pi^{\mathsf{InCom}}_{\mathsf{Diho}},\mathcal{A},\mathcal{Z}}$ is indistinguishable from $\mathsf{IDEAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}},\mathcal{F}^{\mathsf{vOLE}}}_{\mathcal{F}^{\mathsf{InCom}},\mathcal{S},\mathcal{Z}}$.

We first prove that the message distribution are identical in the real and idea execution. For each C_i, the ideal functionalities $\mathcal{F}^{\mathsf{CR}}$, $\mathcal{F}^{\mathsf{Rand}}$, $\mathcal{F}^{\mathsf{Rand}}$ are emulated by $\mathcal{S}$ and thus indistinguishable in both worlds. Thus, we conclude that the messages are identically distributed in both worlds. Moreover, the errors introduced by $\mathcal{A}$ in the ideal execution are the same as in the real execution, and the condition for successfully passing the consistency check is identical in both executions. We conclude that the probability of passing the consistency check in both executions are identical.

Due to Claims 6.3.5 we know that if the consistency check passes, then the MAC shares computed in the real protocol execution are correctly computed as the MAC shares output by $\mathcal{F}^{\mathsf{InCom}}$ in the ideal world, except with negligible probability. In the real execution, both parties' MAC shares are obtained by summing up the random outputs received from $\mathcal{F}^{\mathsf{vOLE}}$ instances. One of them is distributed uniformly at random and serves as a random mask for the correct MACs. In the ideal execution, $\mathcal{F}^{\mathsf{InCom}}$ draws S_0's MAC shares randomly, then sets S_1's MAC shares by subtracting S_0's MAC shares from the correct MAC shares. In both executions, the MACs are correctly computed, and S_0's MAC shares serve as a random mask. Thus, honest parties' outputs are identically distributed in both worlds.

We thus conclude that the simulation is indistinguishable for $\mathcal{Z}$. □

Simulating the case where S_0 is corrupted: $\mathcal{S}$ simulates a real execution in which the corrupted S_0 controlled by $\mathcal{A}$ delivers message to uncorrupted S_1 and C_i in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

Preprocessing:

1. Upon first time receiving α^0 from $\mathcal{A}$ as input to an instance of $\mathcal{F}^{\mathsf{vOLE}}$, send α^0 to $\mathcal{F}^{\mathsf{InCom}}$. Otherwise stores α^0.

2. Emulate $\mathcal{F}^{\mathsf{CR}}$ instance, for each C_i, sample $\mathbf{x}_i^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}^t$, $x_{i,t}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$. Let $\tilde{\mathbf{x}}_i^0 = (\mathbf{x}_i^0, x_{i,t}^0)$. Send $\{\tilde{\mathbf{x}}_i^0\}_{i\in[n]}$ to $\mathcal{A}$.
3. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, for each C_i, receive $\hat{\tilde{\mathbf{x}}}_i^0 = (\hat{\mathbf{x}}_i^0, \hat{x}_{i,t}^0)$ and $\mathbf{b}_i^0$ from $\mathcal{A}$ as input to $\mathcal{F}^{\mathsf{vOLE}}$. Note that $\hat{\tilde{\mathbf{x}}}_i^0$ can be different from $\tilde{\mathbf{x}}_i^0$.
4. If $\mathcal{A}$ sends any (guess, S) message to $\mathcal{F}^{\mathsf{vOLE}}$, forward the guess to $\mathcal{F}^{\mathsf{InCom}}$. If $\mathcal{F}^{\mathsf{InCom}}$ aborts then abort, otherwise store the set $S_1 = S_1 \cap S$ (where initially $S_1 = \mathbb{Z}_{2^s}$).
5. Sample $\alpha^1 \xleftarrow{\$} S_1$. For each C_i, honestly compute $\mathbf{a}_i^0 = \alpha^1 \circ \hat{\tilde{\mathbf{x}}}_i^0 - \mathbf{b}_i^0 \bmod 2^{k+2s}$.

Then for each C_i**:**

6. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive $\mathbf{a}_i^1$ from $\mathcal{A}$ as input to $\mathcal{F}^{\mathsf{vOLE}}$.
7. Sample $x_{i,t}^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$, use zero-valued share inputs to set $\tilde{\mathbf{x}}_i^1 = (\mathbf{0}, x_{i,t}^1)$.
8. Set $\mathbf{x}_i' = \mathbf{x}_i^0 + \mathbf{0}$ and $x_{i,t} = x_{i,t}^0 + x_{i,t}^1 \bmod 2^{k+2s}$. Set $\tilde{\mathbf{x}}_i = (\mathbf{x}_i', x_{i,t})$. Honestly compute $\mathbf{b}_i^1 = \alpha^0 \circ \tilde{\mathbf{x}}_i - \mathbf{a}_i^1 \bmod 2^{k+2s}$.
9. Honestly compute the MAC shares $m_{i,h}^1$ for $h \in [t+1]$.

Consistency Check:

10. Emulate $\mathcal{F}^{\mathsf{Rand}}$, send $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_{2^s}^{nt}$ to $\mathcal{A}$.
11. Act as an honest S_1, send v^1 to $\mathcal{A}$, receive back v^0 and reconstruct v.
12. Receive and open the commitment z^0 from $\mathcal{A}$. Honestly compute d^1 and open S_1's commitment $z^1 = d^1 - v \cdot \alpha^1 \bmod 2^{k+2s}$.
13. Perform the consistency check. If the check fails, abort and terminate.
14. If the check passes then define $\mathcal{A}$'s MAC shares $\mathbf{m}_i^0$ using the received values for $\mathcal{F}^{\mathsf{vOLE}}$. For each client C_i, send $\mathbf{x}_i^0$ and $\mathbf{m}_i^0 \bmod 2^{k+s}$ to $\mathcal{F}^{\mathsf{InCom}}$. Then halt.

Proof. We now prove that $\mathsf{REAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}},\mathcal{F}^{\mathsf{vOLE}}}_{\Pi^{\mathsf{InCom}}_{\mathsf{Diho}},\mathcal{A},\mathcal{Z}}$ is indistinguishable from $\mathsf{IDEAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}},\mathcal{F}^{\mathsf{vOLE}}}_{\mathcal{F}^{\mathsf{InCom}},\mathcal{S},\mathcal{Z}}$.

We first prove that the message distribution are identical in the real and idea execution. For each C_i, the ideal functionalities $\mathcal{F}^{\mathsf{CR}}$, $\mathcal{F}^{\mathsf{Rand}}$, $\mathcal{F}^{\mathsf{Rand}}$ are emulated by $\mathcal{S}$ and thus indistinguishable in both worlds. In the real execution $v^1 = \sum_{h=0}^{nt-1} x_h^1 \cdot r_h + \sum_{h=nt}^{nt+n-1} x_h^1 \bmod 2^{k+2s}$ is computed by S_1, while in the ideal execution v^1 is chosen uniformly at random by $\mathcal{S}$. Since $\sum_{h=nt}^{nt+n-1} x_h^1$ is distributed uniformly at random to $\mathcal{A}$, so is the masked value v^1. We note that z^1 is computed in the same way in both executions, which only reflects the errors introduced by $\mathcal{A}$ and thus perfectly simulated by $\mathcal{S}$. Thus, we conclude that the messages are identically distributed in both worlds. Moreover, since the errors introduced by $\mathcal{A}$ to the ideal execution is the same as in the real execution, we conclude that the probability of passing the consistency check in both executions are identical.

Due to Claims 6.3.6 and 6.3.7, we know that if the consistency check passes, then the MAC shares computed in the real protocol execution are correctly computed as the MAC shares output by $\mathcal{F}^{\mathsf{lnCom}}$ in the ideal world, except with negligible probability. $\mathcal{A}$'s MAC shares are set by $\mathcal{S}$ to the exact computed result obtained by $\mathcal{A}$. In the real execution, S_1's MAC shares are obtained by summing up the random output received from $\mathcal{F}^{\mathsf{vOLE}}$ instances, which serves as a random mask for the correct MACs. In the ideal execution, after receiving $\mathcal{A}$'s MAC shares, $\mathcal{F}^{\mathsf{lnCom}}$ subtracts $\mathcal{A}$'s MAC shares from the correct MACs and sets the result as S_1's MAC shares. These are identical to the MAC shares computed by S_1 in the real execution.

We thus conclude that the simulation is indistinguishable for $\mathcal{Z}$. □

Simulating the case where S_1 is corrupted: Similar to the case when S_0 is corrupted.

Simulating the case where S_1 and a subset of clients C_c are corrupted: $\mathcal{S}$ simulates a real execution in which the corrupted S_1 and C_c controlled by $\mathcal{A}$ deliver messages to the uncorrupted S_0 and C_i in the internal (simulated) interaction. $\mathcal{S}$ works as follows:
Preprocessing:

1. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive α^1 and $\{\mathbf{a}_i^0\}_{i\in[n]}$ from $\mathcal{A}$. Send α^1 to $\mathcal{F}^{\mathsf{lnCom}}$.
2. For each C_i, sample $\mathbf{x}_i^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}^t$, $x_{i,t}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$, set $\tilde{\mathbf{x}}_i^0 = (\mathbf{x}_i^0, x_{i,t}^0)$. Honestly compute $\mathbf{b}_i^0 = \alpha^1 \cdot \tilde{\mathbf{x}}_i^0 - \mathbf{a}_i^0 \bmod 2^{k+2s}$.

Then for each corrupted C_i:

3. Emulate $\mathcal{F}^{\mathsf{CR}}$ instance, send previously sampled $(\mathbf{x}_i^0, x_{i,t}^0)$ to $\mathcal{A}$.
4. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive $\tilde{\mathbf{x}}_i = (\mathbf{x}_i', x_{i,t})$ and $\mathbf{b}_i^1$ from $\mathcal{A}$ as input to $\mathcal{F}^{\mathsf{vOLE}}$.
5. If $\mathcal{A}$ sends any (Guess, S) message to $\mathcal{F}^{\mathsf{vOLE}}$, forward the guess to $\mathcal{F}^{\mathsf{lnCom}}$. If $\mathcal{F}^{\mathsf{lnCom}}$ aborts, then abort; otherwise, store the set $S_0 = S_0 \cap S$ (where initially $S_0 = \mathbb{Z}_{2^s}$).
6. After delivering output shares as $\mathcal{F}^{\mathsf{vOLE}}$ for all corrupted clients, sample $\alpha^0 \xleftarrow{\$} S_0$ (only once during the entire simulation).
7. Honestly compute $\mathbf{a}_i^1 = \alpha^0 \circ \tilde{\mathbf{x}}_i - \mathbf{b}_i^1 \bmod 2^{k+2s}$. Then compute $m_{i,h}^0$ for $h \in [t+1]$.

For each honest C_i:

8. Randomly sample $\mathbf{x}_i^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}^t$, $x_{i,t}^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$ and $\mathbf{b}_i^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}^t$, send $(\mathbf{x}_i^1, x_{i,t}^1, \mathbf{b}_i^1)$ to $\mathcal{A}$.
9. Sample $x_{i,t}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$, use zero-valued share inputs to set $\tilde{\mathbf{x}}_i^0 = (\mathbf{0}, x_{i,t}^0)$.
10. Honestly compute $m_{i,h}^0$ for $h \in [t+1]$.

Consistency Check:

11. Emulate $\mathcal{F}^{\mathsf{Rand}}$, send $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_{2^s}^{nt}$ to $\mathcal{A}$.

12. Act as an honest S_0, send v^0 to $\mathcal{A}$, receive back v^1 and reconstruct v.
13. Receive and open the commitment z^1 from $\mathcal{A}$. Honestly compute d^0 and open S_0's commitment $z^0 = d^0 - v \cdot \alpha^0 \bmod 2^{k+2s}$.
14. Perform the consistency check. If the check fails, abort and terminate.
15. If the check passes then define $\mathcal{A}$'s MAC shares $\mathbf{m}_i^1$ using the received values for $\mathcal{F}^{\mathsf{vOLE}}$. For each corrupted C_i, send $(\mathbf{x}_i^0, \mathbf{x}_i^1)$ and $\mathbf{m}_i^1 \bmod 2^{k+s}$ to $\mathcal{F}^{\mathsf{InCom}}$. For each honest C_i, send $\mathbf{x}_i^1$ and $\mathbf{m}_i^1 \bmod 2^{k+s}$ to $\mathcal{F}^{\mathsf{InCom}}$. Then halt.

Proof. We now prove that $\mathsf{REAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}},\mathcal{F}^{\mathsf{vOLE}}}_{\Pi^{\mathsf{InCom}}_{\mathsf{Diho}},\mathcal{A},\mathcal{Z}}$ is indistinguishable from $\mathsf{IDEAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}},\mathcal{F}^{\mathsf{vOLE}}}_{\mathcal{F}^{\mathsf{InCom}},\mathcal{S},\mathcal{Z}}$.

Similar to the case when S_0 is corrupted, v^0 is uniformly at random due to $\sum_{h=nt}^{nt+n-1} x_h^0$. We conclude that the messages simulated by $\mathcal{S}$ are indistinguishable from those in the real execution.

We still need to prove that the distribution of the MAC shares are identical in both worlds. We consider two different cases. In the first case, both C_i and S_1 are corrupted. And in the second case, only S_1 is corrupted. As discussed in Section 6.3.2, in both cases the adversary cannot introduce any error to S_0's MAC shares. As a result, an honest S_0 already receives the correctly computed MAC shares.

As any error introduced by $\mathcal{A}$ to the consistency check is identical in both executions, the probability that the consistency check results in abort is thus the same in both executions. In the real execution, S_0's MAC shares are the sum of outputs received from $\mathcal{F}^{\mathsf{vOLE}}$ instances, which serves as a random mask for the correct MACs. In the ideal execution, $\mathcal{F}^{\mathsf{InCom}}$ computes the correct MACs then subtract $\mathcal{A}$'s MAC shares (set by $\mathcal{S}$) from those MACs. The result is set to S_1's output, which is identical to the MAC shares computed by S_1 in the real execution.

We thus conclude that the simulation is indistinguishable for $\mathcal{Z}$. □

Simulating the case where S_0 and a subset of clients C_c are corrupted: $\mathcal{S}$ simulates a real execution in which the corrupted S_0 and C_c controlled by $\mathcal{A}$ delivers message to uncorrupted S_1 and C_i in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

Preprocessing:

1. Upon first time receiving α^0 from $\mathcal{A}$ as input to an instance of $\mathcal{F}^{\mathsf{vOLE}}$, send α^0 to $\mathcal{F}^{\mathsf{InCom}}$. Otherwise stores α^0.
2. Emulate $\mathcal{F}^{\mathsf{CR}}$ instance, for each C_i, sample $\mathbf{x}_i^0 \xleftarrow{\$} \mathbb{Z}^t_{2^{k+s}}$, $x^0_{i,t} \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$. Let $\tilde{\mathbf{x}}_i^0 = (\mathbf{x}_i^0, x^0_{i,t})$. Send $\{\tilde{\mathbf{x}}_i^0\}_{i\in[n]}$ to $\mathcal{A}$.
3. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, for each C_i, receive $\hat{\tilde{\mathbf{x}}}_i^0 = (\hat{\mathbf{x}}_i^0, \hat{x}^0_{i,t})$ and $\mathbf{b}_i^0$ from $\mathcal{A}$ as input to $\mathcal{F}^{\mathsf{vOLE}}$. Note that $\hat{\tilde{\mathbf{x}}}_i^0$ can be different from $\tilde{\mathbf{x}}_i^0$.

4. If $\mathcal{A}$ sends any (guess, S) message to $\mathcal{F}^{\mathsf{vOLE}}$, forward the guess to $\mathcal{F}^{\mathsf{InCom}}$. If $\mathcal{F}^{\mathsf{InCom}}$ aborts then abort, otherwise store the set $S_1 = S_1 \cap S$ (where initially $S_1 = \mathbb{Z}_{2^s}$).
5. Sample $\alpha^1 \xleftarrow{\$} S_1$, for each C_i, honestly compute $\mathbf{a}_i^0 = \alpha^1 \circ \hat{\tilde{\mathbf{x}}}_i^0 - \mathbf{b}_i^0 \bmod 2^{k+2s}$.

Then for each corrupted C_i:

6. Receive $(\mathbf{x}_i^1, x_{i,t}^1, \mathbf{b}_i^1)$ from $\mathcal{A}$.
7. Honestly compute the MAC shares $m_{i,h}^1$ for $h \in [t+1]$.

Then for each honest C_i:

8. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive $\mathbf{a}_i^1$ from $\mathcal{A}$ as input to $\mathcal{F}^{\mathsf{vOLE}}$.
9. Sample $x_{i,t}^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+2s}}$, use zero-valued share inputs to set $\tilde{\mathbf{x}}_i^1 = (\mathbf{0}, x_{i,t}^1)$.
10. Set $\mathbf{x}_i' = \mathbf{x}_i^0 + \mathbf{0}$ and $x_{i,t} = x_{i,t}^0 + x_{i,t}^1 \bmod 2^{k+2s}$. Set $\tilde{\mathbf{x}}_i = (\mathbf{x}_i', x_{i,t})$. Honestly compute $\mathbf{b}_i^1 = \alpha^0 \circ \tilde{\mathbf{x}}_i - \mathbf{a}_i^1 \bmod 2^{k+2s}$.
11. Honestly compute the MAC shares $m_{i,h}^1$ for $h \in [t+1]$.

Consistency Check:

12. Emulate $\mathcal{F}^{\mathsf{Rand}}$, send $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_{2^s}^{nt}$ to $\mathcal{A}$.
13. Act as an honest S_1, send v^1 to $\mathcal{A}$, receive back v^0 and reconstruct v.
14. Receive and open the commitment z^0 from $\mathcal{A}$. Honestly compute d^1 and open S_1's commitment $z^1 = d^1 - v \cdot \alpha^1 \bmod 2^{k+2s}$.
15. Perform the consistency check. If the check fails, abort and terminate.
16. If the check passes then define $\mathcal{A}$'s MAC shares $\mathbf{m}_i^0$ using the received values for $\mathcal{F}^{\mathsf{vOLE}}$. For each corrupted C_i, send $(\mathbf{x}_i^0, \mathbf{x}_i^1)$ and $\mathbf{m}_i^0 \bmod 2^{k+s}$ to $\mathcal{F}^{\mathsf{InCom}}$. For each honest C_i, send $\mathbf{x}_i^0$ and $\mathbf{m}_i^0 \bmod 2^{k+s}$ to $\mathcal{F}^{\mathsf{InCom}}$. Then halt.

Proof Sketch. The proof proceeds similarly to the case where only S_0 is corrupted, except that the adversary may now additionally corrupt a subset of clients C_c. We first observe that if both S_0 and C_i are corrupted, the adversary cannot introduce any error into S_1's MAC shares $\mathbf{m}_i^1$. For an honest C_i, the adversary may introduce the same type of errors into $\mathbf{m}_i^1$ as in the case where only S_0 is corrupted. Thus, the adversary must again find a compensating error of the same form as in that case. By Claims 6.3.6 and 6.3.7, the probability that the adversary succeeds in doing so is negligible. □