AlphaFL: Secure Aggregation with Malicious^2 Security for Federated Learning against Dishonest Majority

Yufan Jiang
Karlsruhe Institute of Technology
KASTEL Security Research Labs
Germany
yufan.jiang@kit.edu

Maryam Zarezadeh
Barkhausen Institut
Germany
maryam.zarezadeh@barkhauseninstitut.org

Tianxiang Dai
Lancaster University Leipzig
Germany
t.dai@lancaster.ac.uk

Stefan Köpsell
Barkhausen Institut
Germany
stefan.koepsell@barkhauseninstitut.org

Abstract

Federated learning (FL) proposes to train a global machine learning model across distributed datasets. However, the aggregation protocol as the core component in FL is vulnerable to well-studied attacks, such as inference attacks, poisoning attacks [71] and malicious participants who try to deviate from the protocol [24]. Therefore, it is crucial to achieve both malicious security and poisoning resilience from cryptographic and FL perspectives, respectively. Prior works either achieve incomplete malicious security [76], address issues by using expensive cryptographic tools [22, 59] or assume the availability of a clean dataset on the server side [32].

In this work, we propose AlphaFL, a two-server secure aggregation protocol achieving both **malicious security** in the *universal composability (UC)* framework [19] and **poisoning resilience** in FL (thus malicious^2) against a dishonest majority. We design maliciously secure multi-party computation (MPC) protocols [24, 26, 48] and introduce an efficient input commitment protocol tolerating server-client collusion (dishonest majority). We also propose an efficient input commitment protocol for the non-collusion case (honest majority), which triples the efficiency in time and quadruples that in communication, compared to the state-of-the-art solution in MP-SPDZ [46]. To achieve poisoning resilience, we carry out L_∞ and L_2-Norm checks with a dynamic L_2-Norm bound by introducing a novel silent select protocol, which improves the runtime by at least two times compared to the classic select protocol. Combining these, AlphaFL achieves malicious^2 security at a cost of $25\% - 79\%$ more runtime overhead than the state-of-the-art semi-malicious counterpart Elsa [76], with even less communication cost.

Keywords

Federated Learning, Secure Aggregation, Multi-Party Computation, Poisoning Resilience

Proceedings on Privacy Enhancing Technologies 2025(4), 348–368

https://doi.org/10.56553/popets-2025-0134

1 Introduction

Federated Learning (FL) [63] enables the training of machine learning models across multiple data sources without centralizing data. In FL, clients train models locally on their own datasets and then send the local gradient updates to a central server, which aggregates and redistributes the consolidated model. This iterative process continues until the model converges. FL is designed to safeguard user privacy, but it does not offer a definitive assurance of privacy protection. For example, if an adversary gains access to the gradient updates sent from individual clients, it may be able to deduce information from the clients' local datasets [13, 14, 22, 59, 76]. A wide range of research has examined and explored various inference attacks that could compromise the privacy of FL systems [5, 27, 39, 40, 68]. To counteract such attacks, secure aggregation is applied to safeguard the privacy of the clients' input data [22, 59, 76]. Another challenge in FL is its susceptibility to poisoning attacks. In such attacks, malicious actors can inject corrupted updates into the learning system with the intent of degrading the accuracy of the global model [11, 80, 84] or embedding backdoors [33, 70, 85, 89] that could be exploited in the future. Therefore, it is crucial to achieve the following goals to maintain the robustness of FL systems: **(i) Input Privacy.** The private input of all honest clients must be protected. There should be no single bit leakage about each client's update, except the desired final aggregation result. **(ii) Input Integrity/Output Correctness.** A malicious server may deviate from the protocol and thus perform a *Model Poisoning Attack* just as a malicious client will do. Thus, we require input integrity and output correctness. **(iii) Poisoning Resilience.** Ensuring poisoning resilience in FL is the key to maintaining a reliable global model in spite of attacks. This involves using robust aggregation techniques to detect and mitigate corrupted gradient updates.

In single-aggregator setting, RoFL [59] provides input privacy and an enforcement of norm-based defenses by applying expensive non-interactive zero-knowledge proofs (NIZK), specifically Bulletproofs [15]. Eiffel [22] uses the verifiable Shamir's secret sharing scheme [78] in combination with secret-shared non-interactive proofs (SNIP) to achieve secure aggregation with verified inputs, which requires a public (honest) server to implement the bulletin board. Crucial drawbacks of current single-server systems include the secure channel establishment (or key setup) among clients and the involvement of clients in computation with servers in multiple rounds. In distributed-server setting, Elsa [76] focuses on safeguarding data privacy and designing efficient protocols for semi-honest servers to perform norm-based checks rather than ensuring the output correctness. Thus, the "malicious security" achieved in Elsa [76] is incomplete from cryptographic point of view. SafeFL [32] on the other hand, assumes that servers have access to a clean dataset as an additional resource to carry out the filtering, similar to the approach proposed in FLTrust [20].

Although applying the SPD$\mathbb{Z}_{2^k}$ framework [24] already achieves malicious security against a dishonest majority, simply executing the existing protocols without any adaption is suboptimal for the FL scenario, especially regarding the input protocol. In [24], the input protocol is proposed based on the fact that each party plays the role of both an input party and a computation node. However in FL, clients only serve as input parties and may not necessarily be part of the coming up aggregation process. Thus, they could (and should) be treated differently from the servers. The discussion of how to efficiently integrate such external parties into the existing SPD$\mathbb{Z}_{2^k}$ scheme (for different corruption models) is still missing.

In this work, we propose efficient protocols to address the above issue and improve the efficiency of further MPC protocols. We present AlphaFL: a secure aggregation protocol in a two-server setting, which achieves both **malicious security** and **poisoning resilience** (thus malicious[2]). We tolerate server-client collusion and thus consider the dishonest majority setting. Following recent works [12, 59, 76, 80, 94, 102, 104], we include norm-based filtering mechanisms (e.g. L_∞ and L_2-Norms) against malicious gradient updates. Our contributions can be summarized as follows:

- We propose maliciously secure *input commitment* protocols as backbone to apply the *Information-Theoretic Message Authentication Code (MAC)* scheme [24] in secure aggregation. By executing the (three-party) *input commitment* protocol, each client efficiently shares its gradient and helps to generate the MAC. We provide detailed mathematical proofs to show that the probability of successfully introducing an error and still passing the consistency check in our protocols is negligible. We consider two cases where one of the three parties is corrupted or a server colludes with the client (dishonest majority).
- We propose an efficient silent select protocol to filter malicious gradient updates after the L_2-Norm check. In AlphaFL, servers secretly aggregate accepted gradient updates without reconstructing the L_2-Norm bound and check results, which prevents corrupted parties from performing attacks based on inferring the bound. Compared to the classic select protocol, our protocol cuts the online communication in half. To support computation of the L_2-Norm check, we also present [1] a simple way to generate *square correlation* on ring.
- We prove the security of proposed protocols in the *universal composability (UC)* framework [19]. And then we identify a subtlety in constructing ideal functionalities for the SPD$\mathbb{Z}_{2^k}$ scheme [24], which is elaborated in Section 5.3.
- By introducing different building blocks, we build AlphaFL as an efficient aggregation protocol with malicious[2] security in FL systems. We perform a fair evaluation of our scheme and compare it with other state-of-the-art solutions. The results indicate that in the non-collusion setting, AlphaFL achieves malicious[2] security at a cost of 25% – 79% more runtime overhead than the state-of-the-art semi-malicious counterpart Elsa [76], with even less communication overhead.

[1]We notice that MP-SPDZ [46] has a similar idea as us for square correlation generation and implemented it earlier, but without any documentation.

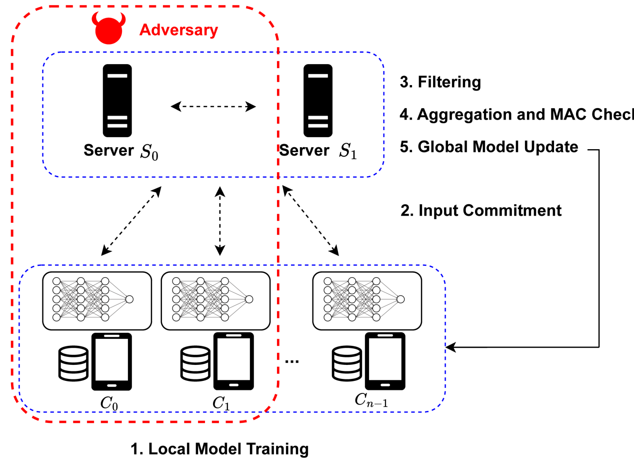

Figure 1: Aggregation protocol with malicious security in AlphaFL **with distributed servers**

1.1 AlphaFL Overview

Motivation. The fundamental strategy employed by AlphaFL to counteract threats posed by a malicious server or collusion between a server and client centers on utilizing a MAC scheme [24], which empowers servers with the capability to authenticate and verify the integrity of the results generated during the computation process. In scenarios where a malicious adversary successfully compromises a subset of participating parties and diverges from the prescribed protocol execution, the system is designed so that at least one honest server will detect the anomaly.

Overview. In AlphaFL, the maliciously secure aggregation protocol is meticulously divided into five stages as illustrated in Fig. 1. **(1/2)** After the local model training, each client C_i securely commits its local gradient update $\mathbf{u}_i$ to the servers. This commitment involves the client in submitting its data in a way that prevents any subsequent alterations, which is crucial for maintaining data integrity throughout the process. **(3)** Servers perform filtering operations by executing checks using both L_∞-Norm and L_2-Norm. In AlphaFL, we also consider applying a dynamic L_2-Norm bound, which is computed based on the clients' private input and thus must be kept secret from the adversary. **(4)** Following the filtering process, the servers proceed to aggregate the accepted updates $\mathbf{u}_i$ provided by each client. The crucial part of applying such a dynamic bound (compared to a public bound) is that servers must perform the secure aggregation secretly without revealing the bound. We elaborate our solution in Section 4.3. **(5)** To ensure the integrity and correctness of the aggregated result $\mathcal{U}$, servers perform a MAC verification. This step ensures the aggregated output is unaltered and thus accurately reflects the combined updates of clients. Finally, servers update the global model $\mathcal{M}_{q-1}$ to $\mathcal{M}_q$ and redistribute $\mathcal{M}_q$ to clients.

2 Related Work

Malicious FL clients can compromise the global model in two ways: a) data poisoning, where harmful data is added to training sets [50, 81, 82, 86] and b) model poisoning, where attackers submit a maliciously altered update [4, 30, 61, 90]. Previous studies have explored mitigating poisoning attacks [53, 57, 66, 67, 98, 102]. One such approach is utilizing Cosine distance to detect poisoned updates that deviate from benign ones [3, 20, 28, 31, 61, 62]. Clustering [21, 71, 97] and anomaly detection [2, 44] are also applied to identify and filter malicious updates. Another approach is clipping and noising, which smooths updates and reduces discrepancies [54, 94]. Combining these two methods with secure aggregation is challenging as it hides necessary inputs. Several methods employ Byzantine-robust [30] defenses against these threats like Krum [12] and trimmed mean [99].

To protect user privacy and guarantee output correctness, secure aggregation in one-server setting is well studied [35, 77, 91, 105]. SecAgg [14] combines masking, Shamir's secret sharing [78] and symmetric encryption to protect local models from unauthorized access. VerifyNet [95] and VeriFL [34] build on top of SecAgg [14] with additional verifiability features to ensure aggregation accuracy. SecAgg+ [10], SVFL [58], and Flamingo [60] use masking techniques to attempt at a better efficiency. E-SeaFL [8] applies authenticated homomorphic vector commitments to generate a proof of the honestly aggregated result. Both [41] and [100] require a trusted third party to support result verification. Meanwhile, [74] introduces several attacks in the presence of one malicious server.

While the above works do not consider poisoning attacks from the clients, Prio [23], Prio+ [1] and Eiffel [22] use SNIP to validate clients' input. RoFL [59] also uses NIZK to perform norm-based defenses. Conversely, Acorn [9] proposes to use range proofs while Flag [6] improves security for adaptive adversaries. On the other hand, MLGuard [49], FLGuard [72] and SafeFL [32] apply MPC protocols to filter invalid inputs. Elsa [76] accelerates online computation by offloading oblivious transfer correlations and Beaver triples to clients. Prior works also apply techniques such as differential privacy [42, 56, 88, 93, 103], trusted execution environment [45, 64, 65, 75], homormorphic encryption [17, 43, 73, 79, 101], Zero Knowledge Proofs (ZKPs) [29, 69, 92] and hybrid approaches [16, 83, 87, 96] to counter corrupted actors in FL.

3 Preliminaries

A bold value $\mathbf{x}$ denotes a vector $\mathbf{x} = \{x_0, ..., x_{n-1}\}$, where x_h (and sometimes $\mathbf{x}[i]$) is the h-th (i-th) element of $\mathbf{x}$. If a party sets $\tilde{\mathbf{x}} = (\mathbf{x}, x_t)$, it extends the vector $\mathbf{x}$ with an additional element x_t. We also use $\equiv_k$ to denote the modulo computation. We let $\otimes$ denote element-wise multiplication.

3.1 SPD$\mathbb{Z}_{2^k}$ Secret-sharing

In SPD$\mathbb{Z}_{2^k}$ [24], an Information-Theoretic MAC Scheme is introduced. Each party holds an additive MAC key share $\alpha^j \in \mathbb{Z}_{2^s}$, such that $\alpha = \alpha^0 + \alpha^1 \bmod 2^{k+s}$ is a secret **global** MAC key. An authenticated, secret value $x \in \mathbb{Z}_{2^k}$ is shared between parties (in 2PC), if each party holds $x^j \in \mathbb{Z}_{2^{k+s}}$ over a larger ring $\mathbb{Z}_{2^{k+s}}$, such that $x' = x^0 + x^1 \bmod 2^{k+s}$ and $x = x' \bmod 2^k$. Additionally, each party holds a shared MAC $m^j \in \mathbb{Z}_{2^{k+s}}$, such that $m = m^0 + m^1$ and $m = \alpha \cdot x' \bmod 2^{k+s}$. Since α is a global MAC key, we abbreviate each local share as (x^j, m^j) and denote such a sharing scheme as $[\cdot]$. A boolean shared value is denoted as $[x]_2$, where $[x]_2^j \leftarrow (x^j, m_x^j)$ and $x^j, m_x^j \in \mathbb{Z}_{2^{1+s}}$. Addition and multiplication of boolean shared values over $\mathbb{Z}_{2^{1+s}}$ correspond to XOR and AND computations over $\mathbb{Z}_2$. As remarked by [26], we only require $z \leftarrow x \cdot y \bmod 2$, but not necessarily $z \leftarrow x \cdot y \bmod 2^{1+s}$.

3.2 L_2-Norm and L_∞-Norm

The Euclidean Norm of a vector $\mathbf{x} \leftarrow (x_0, ..., x_{n-1})$, or L_2-Norm, is defined as $L_2(\mathbf{x}) \leftarrow \sqrt{x_0^2 + ... + x_{n-1}^2}$. Performing L_2-Norm check is central to our approach in countering boosted gradients. Due to the computation complexity, we bound the square of L_2-Norm by β^2 instead of directly bounding L_2-Norm. 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. With L_∞-Norm, $x_{\max} \leftarrow \max_i |x_i|$ is now bounded by 2^{w-1} for some $w \in \mathbb{N}$. In this work, we simply bound every element in $\mathbf{x}$ by 2^{w-1}.

3.2.1 Limitation of a Norm-based Defense. Some studies [59, 80, 85] have shown that norm-based defenses can effectively defend against many sophisticated poisoning attacks. However, recent research has highlighted its inherent limitations in terms of effectiveness against various backdoor attacks. For example, [89] forces a model to misclassify data points living on the tail of the input distribution. RoFL [59] also proves that a tail backdoor remains effective for a long period even in the presence of norm-based defenses. In addition, both works [59, 89] have shown that an strong attacker can continuously influence the global model on tail data points by periodically lower the attack intensity, without being detected by the norm-based defenses. We refer to Flame [71] and RoFL [59] for more comprehensive benchmarks.

3.3 Threat Model

3.3.1 Malicious Adversary in UC. In this paper, we consider security against malicious adversaries, where a corrupted party can arbitrarily deviate from the protocol. 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 1 (Universal Composability (UC) security). *Let $\mathcal{F}$ be a functionality and let Π be a protocol that computes $\mathcal{F}$. Protocol Π is said to **uc-realizes $\mathcal{F}$ in the presence of static 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 environment $\mathcal{Z}$:*

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

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

We follow the security definition described in the UC framework [19]. Using a hybrid model, an *uc-secure* protocol can be abstracted as an ideal functionality and invoked within other protocols.

3.3.2 Malicious Client in Federated Learning. We let C_c denote the compromised clients and q denote the index of iteration. 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 2 (Compromised model [71]). *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 c' chosen by the adversary. The model is said to be successfully compromised by the adversary, if:*

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

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

3.3.3 Malicious² Security. In the traditional *Security with Abort* paradigm [55], input validity is out of scope by definition. In FL, a malicious adversary may not just deviate from the protocol, but also corrupts a subset of clients and use malicious inputs to manipulate the final result as explained in Section 3.3.2. We thus use the term *Malicious² Security* to denote the malicious security in the UC framework and poisoning resilience in FL. In this work, we consider two different settings: one where a malicious adversary corrupts either a subset of clients or a server (non-colluding case), and another where it corrupts a subset of clients together with one of the servers (server-client collusion). We will elaborate these two settings in Section 4.1.

3.3.4 Trivial Attacks. An adversary can corrupt a server, causing it to disconnect from the network, which cannot be prevented by other honest protocol participants. Meanwhile, any corrupted party may falsely abort under the *Security with abort* paradigm [55], even if all other parties are behaving honestly. Furthermore, we do not discuss the case where the adversary corrupts two servers.

4 Important Building Blocks

In this section, we propose our input commitment protocols Π^{InCom} against an honest majority and $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ against a dishonest majority. Then we present a simple protocol Π^{SqGen} to generate square correlations with the help of Beaver triples, which is implemented in MP-SPDZ [46]. Regarding the privacy of dynamic L_2-Norm bounds, we propose a silent select protocol Π^{SiSelect} to obliviously filter malicious gradient updates.

4.1 Input Commitment Protocol

As mentioned previously, the input protocol in [24] is constructed where each participant serves as a computation node. In fact, each party exactly holds a global MAC key share, including the input party itself. This leaves us with an opportunity to change the global MAC key share holding and construct efficient protocols in different corruption settings.

Protocol Π^{InCom}

Private inputs: A client C_i holds $\mathbf{x}$, where $\mathbf{x} = \{x_0, \ldots, x_{t-1}\} \in \mathbb{Z}^t_{2^{k+s}}$ and $\mathsf{C}_i \in \{\mathsf{C}_0, \ldots, \mathsf{C}_{n-1}\}$.
Public inputs: Public parameters k, s and sid.
Outputs: S_j outputs $[\mathbf{x}]^j \leftarrow (\mathbf{x}^j, \mathbf{m}^j) \in (\mathbb{Z}^t_{2^{k+s}}, \mathbb{Z}^t_{2^{k+s}})$, where $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$. C_i outputs $(\mathbf{x}^0, \mathbf{x}^1)$.
Initialize: C_i and S_0 call their $\mathcal{F}^{\mathsf{CR,glo}}$ instance, receive $\alpha^0 \stackrel{\$}{\leftarrow} 2^s$. Then C_i and S_1 call their $\mathcal{F}^{\mathsf{CR,glo}}$ instance, receive $\alpha^1 \stackrel{\$}{\leftarrow} 2^s$.
Protocol:

1. C_i and S_0 call their $\mathcal{F}^{\mathsf{CR}}$ instance, receive $\mathbf{x}^0 \stackrel{\$}{\leftarrow} \mathbb{Z}^t_{2^{k+s}}$, $x^0_t \stackrel{\$}{\leftarrow} \mathbb{Z}_{2^{k+s}}$ $\mathbf{m}^0 \stackrel{\$}{\leftarrow} \mathbb{Z}^t_{2^{k+2s}}$ and $m^0_t \stackrel{\$}{\leftarrow} \mathbb{Z}_{2^{k+2s}}$.
2. C_i sets $\alpha \leftarrow \alpha^0 + \alpha^1 \mod 2^{k+s}$ and chooses $x^1_t \stackrel{\$}{\leftarrow} \mathbb{Z}_{2^{k+s}}$, then computes $\mathbf{x}^1 \leftarrow \mathbf{x} - \mathbf{x}^0$ and $x_t \leftarrow x^0_t + x^1_t \mod 2^{k+s}$.
3. C_i computes MACs as $\mathbf{m} \leftarrow \mathbf{x} \cdot \alpha \mod 2^{k+2s}$ and $m_t \leftarrow x_t \cdot \alpha \mod 2^{k+2s}$. Then it sets $\mathbf{m}^1 \leftarrow \mathbf{m} - \mathbf{m}^0$ and $m^1_t \leftarrow m_t - m^0_t \mod 2^{k+2s}$.
4. C_i now sends $(\mathbf{x}^1, x^1_t, \mathbf{m}^1, m^1_t)$ to S_1.

Consistency Check

5. S_0 and S_1 call $\mathcal{F}^{\mathsf{Rand}}$, receive $\mathbf{r} \in \mathbb{Z}^t_{2^s}$.
6. S_j computes $v^j \leftarrow \sum_{h=0}^{t-1} x^j_h \cdot r_h + x^j_t \mod 2^{k+2s}$ and $d^j \leftarrow \sum_{h=0}^{t-1} m^j_h \cdot r_h + m^j_t \mod 2^{k+2s}$.
7. S_j sends v^j to S_{j-1} and computes $v \leftarrow v^0 + v^1 \mod 2^{k+2s}$. It then commits and sends $z^j \leftarrow d^j - v \cdot \alpha^j \mod 2^{k+2s}$ to S_{j-1}.
8. S_j computes $z \leftarrow z^0 + z^1 \mod 2^{k+2s}$ and checks if $z = 0$, aborts if not.

Figure 2: Input commitment protocol Π^{InCom}

Note that each client operates independently in the **Input Commitment** stage, so the whole computation can be viewed as a three-party input commitment protocol, where C_i shares its authenticated input to servers. We first exclude the naive solution, where we simply let C_i share its input to servers and let the servers compute the authentication MAC by themselves, since we cannot guarantee that a malicious server will use the exact share received from C_i to compute the MAC. For simplicity, we consider static corruption in this paper and discuss two cases as follows: **(i) Honest Majority:** we propose the protocol Π^{InCom} as described in Fig. 2, where we allow one party from $\{\mathsf{C}_i, \mathsf{S}_0, \mathsf{S}_1\}$ to be corrupted. This indicates that either one server **or** multiple clients can be corrupted by a malicious adversary $\mathcal{A}$ in the federated learning scenario. Yet, MP-SPDZ [46] has implemented another variant of the input commitment protocol, which is proven to be secure if the client is honest (honest majority) [25]. We show in Section 7.6 that our honest majority variant achieves a better efficiency, and then we provide a security analysis in Section 5.1 to show that the implemented protocol in [46] is vulnerable in the collusion case. **(ii) Dishonest Majority:** We consider a more complicated setting, where the malicious adversary corrupts both the client C_i and a server from $\{\mathsf{S}_0, \mathsf{S}_1\}$. Then we propose $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ as described in Fig. 3. Again, if we jump out from the first stage and review the entire aggregation protocol, this means that a server and multiple clients can be corrupted by the same adversary $\mathcal{A}$.

4.1.1 Π^{InCom} *in Honest Majority Setting.* The key idea is to let C_i efficiently distribute MAC shares by holding the global MAC key $\alpha \leftarrow \alpha^0 + \alpha^1$. We use a correlated randomness functionality $\mathcal{F}^{\mathsf{CR}}$ (described in Fig. 17) between C_i and S_0 to reduce the communication, which can be implemented by letting parties hold a pre-shared key and derive pseudo-randomness by evaluating keyed pseudo-random function (PRF). Then both servers can perform *consistency check* to authenticate the distributed MAC shares. We observe that if C_i is honest, the distributed MAC shares must be correct. If C_i is however maliciously corrupted and distributes incorrect MAC shares to servers, we show in Section 5.2 that the probability of passing the consistency check is only 2^{-s}, even if C_i holds both α^0 and α^1. The protocol Π^{InCom} is formally described in Fig. 2.

Note that all C_i use the same global MAC key shares α_0, α_1 while executing Π^{InCom}. They obtain the shares by calling the "global" version of the functionality instance $\mathcal{F}^{\mathsf{CR,glo}}$ (described in Fig. 18) during initialization. It differs from an $\mathcal{F}^{\mathsf{CR}}$ instance, since all C_i share the same key (with S_0 and S_1, respectively) as setup, where the pre-shared key for an $\mathcal{F}^{\mathsf{CR}}$ instance varies from client to client. We also present two ways to bypass the reliance on $\mathcal{F}^{\mathsf{CR,glo}}$:

- With the assumption of a secure broadcast channel (e.g. as defined in [7]), servers can broadcast α_j to each C_i, since the broadcast functionality guarantees that all clients receive the same message.
- By relying solely on a peer-to-peer secure channel, servers can first exchange the commitments of α_j with each other and later decommit to each client. Since there will be at least one honest server participating in the protocol, it is guaranteed that clients will receive either the same global MAC key share or the same commitment of α^j.

4.1.2 $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ *in Dishonest Majority Setting.* It is easy to see that the above protocol is no longer secure if C_i can collude with any server, since the corrupted server can thus easily pass the *consistency check* by holding both the global MAC key shares α^0 and α^1. In [24], C_i has to set up the secure channels with both servers and perform a Vector Oblivious Linear Evaluation (vOLE) functionality $\mathcal{F}^{\mathsf{vOLE}}$ (described in Fig. 14) twice in the online stage. To optimize the communication overhead for C_i, we apply an asymmetric setting in $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ and decompose the computation into $\alpha^1 \cdot \mathbf{x}^0 + \alpha^1 \cdot \mathbf{x}^1 + \alpha^0 \cdot (\mathbf{x}^0 + \mathbf{x}^1)$. Now, the first term $\alpha^1 \cdot \mathbf{x}^0$ can be computed between servers without involving C_i and can thus be moved to the preprocessing stage. The second term $\alpha^1 \cdot \mathbf{x}^1$ can be locally computed by S_1. C_i only has to participant in a single $\mathcal{F}^{\mathsf{vOLE}}$ in the online stage with S_0 to compute the third term $\alpha^0 \cdot \mathbf{x}$. Finally, servers apply *consistency check* to verify the correctness of the computed authentication MACs.

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

Private inputs: A client C_i holds $\mathbf{x}$, where $\mathbf{x} = \{x_0, ..., x_{t-1}\} \in \mathbb{Z}^t_{2^{k+s}}$ and $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$.
Public inputs: Public parameters k, s and sid.
Outputs: S_j outputs $[\mathbf{x}]^j \leftarrow (\mathbf{x}^j, \mathbf{m}^j) \in (\mathbb{Z}^t_{2^{k+s}}, \mathbb{Z}^t_{2^{k+s}})$, where $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$. C_i outputs $(\mathbf{x}^0, \mathbf{x}^1)$.

Initialize: S_0 chooses $\alpha^0 \xleftarrow{\$} \mathbb{Z}_{2^s}$. Then S_1 chooses $\alpha^1 \xleftarrow{\$} \mathbb{Z}_{2^s}$. S_0 initialize 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 $\mathbf{x}^0 \xleftarrow{\$} \mathbb{Z}^t_{2^{k+s}}$, $x^0_t \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$.
2. S_0 sets $\tilde{\mathbf{x}}^0 \leftarrow (\mathbf{x}^0, x^0_t) \in \mathbb{Z}^{t+1}_{2^{k+s}}$.
3. S_0 and S_1 call their $\mathcal{F}^{\mathsf{vOLE}}$ instance (See Fig. 14) 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 $\mathbf{a}^0 = \mathbf{b}^0 + \alpha^1 \cdot \tilde{\mathbf{x}}^0 \mod 2^{k+2s}$.

Protocol:

1. C_i calls its $\mathcal{F}^{\mathsf{CR}}$ instance (with S_0), receives $\mathbf{x}^0 \xleftarrow{\$} \mathbb{Z}^t_{2^{k+s}}$, $x^0_t \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$.
2. C_i chooses $x^1_t \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, then computes $\mathbf{x}^1 \leftarrow \mathbf{x} - \mathbf{x}^0$ and $x_t \leftarrow x^0_t + x^1_t \mod 2^{k+s}$. It sets $\tilde{\mathbf{x}} \leftarrow (\mathbf{x}, x_t) \in \mathbb{Z}^{t+1}_{2^{k+s}}$.
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 $\mathbf{a}^1 = \mathbf{b}^1 + \alpha^0 \cdot \tilde{\mathbf{x}} \mod 2^{k+2s}$.
5. C_i sends $(\mathbf{x}^1, x^1_t, \mathbf{b}^1)$ to S_1, which sets $\tilde{\mathbf{x}}^1 \leftarrow (\mathbf{x}^1, x^1_t) \in \mathbb{Z}^{t+1}_{2^{k+s}}$.
6. The h-th MAC share is defined as follows:
   - S_0: $m^0_h \leftarrow \mathbf{a}^1[h] - \mathbf{b}^0[h]$.
   - S_1: $m^1_h \leftarrow \mathbf{a}^0[h] - \mathbf{b}^1[h] + \alpha^1 \cdot \tilde{\mathbf{x}}^1[h]$

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

Figure 3: Input commitment protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ in the dishonest majority setting

4.2 Square Correlation Generation between Malicious Servers

To compute the squares of secretly shared values, it is more efficient to use square correlations in the online stage compared to Beaver triples [36, 76]. In the honest majority setting, we can let C_i generate $([\mathbf{a}], [\mathbf{d}], [\mathbf{a}'], [\mathbf{d}'])$ by invoking the protocol Π^{InCom}, where $d_h = a_h \cdot a_h$ and $d'_h = a'_h \cdot a'_h$. Servers can then apply the correlation sacrifice step to verify the correctness of the generated correlation [24, 47, 48, 76]. In this section, we forcus on the dishonest majority setting. C_i is no longer allowed to distribute the shares. Instead, servers have to generate the shares via 2PC. First, we would like to avoid *homomorphic encryption* and *zero knowledge proofs* computation like in [36, 47, 48]. Then we notice that in [24, 47],

Protocol Π^{SqGen}

Public inputs: Public parameters k, s and sid.
Outputs: S_j outputs $([\mathbf{a}]^j, [\mathbf{d}]^j)$, where $j \in \{0,1\}$, $(\mathbf{a}, \mathbf{d}) \in (\mathbb{Z}^t_{2^{k+s}}, \mathbb{Z}^t_{2^{k+s}})$ and $d_h \leftarrow a_h \cdot a_h \mod 2^k$.
Initialize: S_j send $(\mathsf{Init}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{TripGen}}$, receives α_j.
Protocol:

1. S_j call $\mathcal{F}^{\mathsf{TripGen}}$ (See Fig. 13), receives $([\mathbf{a}]^j, [\mathbf{b}]^j, [\mathbf{c}]^j)$, where $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{Z}^t_{2^{k+s}}$ and $c_h \leftarrow a_h \cdot b_h \mod 2^k$.
2. Servers use **Open** and **MAC check** (See Fig. 19) to reconstruct $\mathbf{e} \leftarrow [\mathbf{b}] - [\mathbf{a}]$.
3. If the check passes, servers locally compute h-th share as $[d_h] \leftarrow [c_h] - e_h \cdot [a_h]$ and output $([\mathbf{a}], [\mathbf{d}])$.

Figure 4: Square correlation generation protocol Π^{SqGen}

the generated triple will be "rerandomized" by performing a linear **Combine** procedure to maintain the security guarantee of the sacrificing step. Let $([a_h], [b_h], [c_h])$ denote the h-th generated triple. In other word, parties do not have any control on the final output $[a_h]$, since only $[b_h]$ is locally sampled by parties. As a result, we cannot trivially adapt the triple generation protocol to a square correlation generation protocol. We now introduce our protocol Π^{SqGen} as shown in Fig. 4, which is constructed based on $\mathcal{F}^{\mathsf{TripGen}}$ described in Fig. 13. Our idea is very simple: we let servers first generate the normal Beaver triples, then efficiently convert those to the square correlations. As mentioned in [24], the **MAC Check** step can be postponed to the output reconstruction stage and be efficiently executed in batch. We place the **BatchCheck** procedure [2] in Fig. 19 and show the correctness of Π^{SqGen} as follows:

$$\begin{aligned} c_h &\equiv_k a_h \cdot b_h \equiv_k a_h \cdot (a_h + (b_h - a_h)) \\ &\equiv_k a_h \cdot a_h + a_h \cdot (b_h - a_h) \\ \Leftrightarrow a_h \cdot a_h &\equiv_k c_h - a_h \cdot \underbrace{(b_h - a_h)}_{e_h} \end{aligned}$$

4.3 Silent Select Protocol

Recap of current L_2-Norm checks. Both Flame [71] and RoFL [59] have proven that applying a dynamic L_2-Norm bound β achieves a better filter performance compared to using a fixed bound [3]. Current works such as Flame [71], RoFL [59] and Elsa [76] assume that β can be publicly determined. The servers must hold a separate training dataset to compute the bound [59], or β is essentially reconstructed on the server side [76]. However, if a separate training dataset is not available, β will then be computed based on the real-time gradient updates of the clients. This makes β an intermediate result, and a direct reconstruction of β (or β^2) will thus leak information to the adversary. In addition, if we allow one server to collude with multiple clients, simply hiding β^2 from servers does not solve the problem. In Elsa [76], although the comparison between $(L_2(\mathbf{u}_i))^2$ and β^2 is secretly performed, the comparison result s_i will be reconstructed. If $s_i = 0$, it indicates that $(L_2(\mathbf{u}_i))^2 \geq \beta^2$ and servers should reject C_i for further computation. The drawback of such an approach is that the adversary is able to guess β^2 several times and infer the range of β^2 by having $(L_2(\mathbf{u}_i))^2$ in plaintext.

Protocol Π^{SiSelect}

Private inputs: Servers hold $[\mathbf{x}]$ and $[y]_2$, where $\mathbf{x} \in \mathbb{Z}^t_{2^{k+s}}$ and $y \in \mathbb{Z}_2$. Additionally, $\mathcal{F}^{\mathsf{TripGen}}$ is already initialized and S_j holds α_j.
Public inputs: Public parameters k, s and sid.
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 $(\mathsf{BitTripGen}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{TripGen}}$, receives $([\mathbf{a}]^j, [\mathbf{b}]^j, [\mathbf{c}]^j)$, where $\mathbf{a}, \mathbf{c} \in \mathbb{Z}^t_{2^{k+s}}$, $\mathbf{b} \mod 2^k \in \mathbb{Z}^t_2$, $c_h \leftarrow a_h \cdot b_h \mod 2^k$.
2. Let b^j_h and $m^j_{b_h}$ be S_j's share and MAC share of $\mathbf{b}[h]$. S_j defines $[\mathbf{b}']^j_2$, where $b'^j_h \leftarrow b^j_h \mod 2^{1+s}$ and $m^j_{b'_h} \leftarrow m^j_{b_h} \mod 2^{1+s}$.

Protocol:

1. Servers extend $[y]_2$ to $[\mathbf{y}]_2$. They run **Open** and **MAC check** to reconstruct $\mathbf{e} \leftarrow [\mathbf{x}] - [\mathbf{a}]$ and $\mathbf{f} \leftarrow [\mathbf{y}]_2 + [\mathbf{b}']_2$.
2. If $f_h = 0$, S_j sets $[z_h]^j \leftarrow [c_h]^j + e_h \cdot [b_h]^j$.
If $f_h = 1$, S_j sets $[z_h]^j \leftarrow j \cdot e_h + [a_h]^j - [c_h]^j - e_h \cdot [b_h]^j$

Figure 5: Silent select protocol Π^{SiSelect}

Silent select protocol. We address the the above issue by executing the select ideal functionality $\mathcal{F}^{\mathsf{Select}}$ to "select" only **valid** gradient updates. And we propose the silent select protocol Π^{SiSelect} to minimize the online communication overhead compared to the classic select protocol. We first briefly recap $\mathcal{F}^{\mathsf{Select}}$. Initially, servers hold $[\mathbf{x}]$ and $[y]_2$. $\mathcal{F}^{\mathsf{Select}}$ outputs $[\mathbf{z}]$, where $z_h = x_h$ if $y = 1$ and $z_h = 0$ otherwise. A classic select protocol realizing $\mathcal{F}^{\mathsf{Select}}$ works as follows: servers first convert $[y]_2$ to $[y]$ by executing a Boolean-to-Arithmetic protocol (B2A) then compute the multiplication between $[x_h]$ and $[y]$ with a Beaver Triple. However, such an implementation is not optimal. It requires **two** rounds in the online stage and $2t \cdot (k+s) + (s+1)$ bits of communication overhead (regardless of **MAC check**).

We now describe the protocol Π^{SiSelect}, which consists of a preprocessing (offline) phase and an online phase. For clarity, we only consider the server-client collusion case, which indicates that all correlated randomness must be generated via executing secure MPC protocols between servers.

a) Preprocessing: The core idea of Π^{SiSelect} to accelerate the online computation is to generate so called *Select Correlations* in the preprocessing phase. First, servers generate multiplication triples $([\mathbf{a}], [\mathbf{b}], [\mathbf{c}])$ by calling $\mathcal{F}^{\mathsf{TripGen}}$ described in Fig. 13, where $c_h \leftarrow a_h \cdot b_h \bmod 2^k$ and $b_h \bmod 2^k \in \mathbb{Z}_2$. Compared to a traditional

[2] We notice that the *consistency check* described in Fig. 2 and Fig. 3 requires parties to additionally compute (x_t, m_t) compared to **BatchCheck**. The reason is that parties must use (x^j_t, m^j_t) as a mask to hide the distribution of v^j and d^j (step 6 in Fig. 2). We refer to the security proof in SPD$\mathbb{Z}_{2^k}$ [24] for more details.

[3] In this work, β is set to the mean of all clients' L_2 norms instead of the median as suggested in Flame [71] and RoFL [59]. We leave the choice of a dynamic L_2-Norm bound as an orthogonal research.

Beaver triple, we restrict b_h to be either 1 or 0 over $\mathbb{Z}_{2^k}$. Note that in the Beaver triple generation protocol Π^{Triple} [24], parties first sample $b_h \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$ then determine a_h and c_h via **Combine**. Now in order to generate an authenticated random bit $[b_h]$, parties execute $\Pi^{\mathsf{RanBitGen}}$ (described in Fig. 21) instead of sampling b_h randomly. The crucial step of restricting $b_h \in \mathbb{Z}_2 \mod 2^k$ 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. After generating the triple, servers convert the arithmetically shared $[b_h]$ to boolean shared $[b'_h]_2$ by executing the Arithmetic-to-Boolean protocol (A2B) provided in [26]. The *Select Correlations* are essentially defined as $([\mathbf{a}], [\mathbf{b}], [\mathbf{c}], [\mathbf{b}']_2)$, where $b_h \bmod 2^k \in \mathbb{Z}_2$.

b) Online phase: In the online phase, servers only have to reconstruct **e** and **f** in a **single** round, then locally compute the shared output $[\mathbf{z}]$ supported by *Select Correlations*. The communication overhead of Π^{SiSelect} (regardless of **MAC check**) is $t \cdot (k+2s+1)$ bits, which cuts the communication of the classic protocol approximately in half. We show the correctness of Π^{SiSelect} as follows:

$$\begin{aligned}
&(1-f_h)\cdot(c_h + e_h\cdot b_h)+\\
&f_h\cdot(e_h + a_h - c_h - e_h\cdot b_h)\\
\equiv_k\ & f_h\cdot(e_h+a_h) + (1-f_h)\cdot(c_h+e_h\cdot b_h)\\
\equiv_k\ & f_h\cdot x_h + b_h\cdot x_h - 2f_h\cdot b_h\cdot x_h\\
\equiv_k\ & x_h\cdot(f_h - 2f_h\cdot b_h + b_h)\\
\equiv_k\ & x_h\cdot s_h
\end{aligned}$$

5 Security Analysis

In this section, we first analyze the collusion case of the input commitment protocol variant implemented in [46]. Then we formally prove that the probability of successfully introducing an error into MACs is negligible while executing Π^{InCom} and $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$. We also show a subtlety while modeling ideal functionalities for SPD$\mathbb{Z}_{2^k}$ regarding the global MAC key extraction. Finally, we provide theorems and proofs for the proposed protocols.

5.1 A Collusion Case Analysis for the Input Commitment Protocol in [46]

In [46], servers first generate a Beaver triple $([a], [b], [c])$, then reconstruct (a, b, c) on the client side, who checks the correlation of the opened triple and broadcasts $x - a$ to servers. We discuss the current implementation where the triple generation is implemented via executing the protocol Π^{TripGen} [24] and $\mathcal{F}^{\mathsf{MAC}}$ is implemented via executing the protocol Π^{Auth} [24]. While the whole scheme is secure in the honest majority setting, it is vulnerable to the server-client collusion. Note that the only secret of the collusion case is the global MAC key share of the honest party.

For the following proofs, let S_0 be the honest server, let S_1 and C_i be corrupted by the adversary $\mathcal{A}$. We first consider notations used in Π^{TripGen} [24]. During the execution of Π^{TripGen}, servers have to compute and verify the MACs of both the output triple (a, b, c) and the sacrificed triple $(\hat{a}, \hat{c})$. Since now the triple (a, b, c) is reconstructed at C_i, $\mathcal{A}$ receives (a^0, b^0, c^0) from S_0. Again, since $\rho \leftarrow t\cdot[a] - [\hat{a}]$ and $\sigma \leftarrow t\cdot[c] - [\hat{c}] - \rho\cdot[b]$ are opened in the sacrifice step, $\mathcal{A}$ receives ρ^0 and σ^0 from S_0 and can thus compute $\hat{a}^0 = \rho^0 - t\cdot a^0 \mod 2^{k+s}$ and $\hat{c}^0 = t\cdot c^0 - \rho\cdot b^0 - \sigma^0 \mod 2^{k+s}$ (t is public). For clarity, we first assume that all values $a^0, b^0, c^0, \hat{a}^0, \hat{c}^0 \in \mathbb{Z}_{2^{k+s}}$ known to $\mathcal{A}$ are the exact intermediate values in Π^{TripGen} without masking the first s bits. Let $(m_a^0, m_b^0, m_c^0, m_{\hat{a}}^0, m_{\hat{c}}^0)$ denote the MAC shares of S_0, $\mathcal{A}$ can decompose each MAC share into

$$\begin{aligned}
m_x^0 &= (x^0 + x^1)\cdot(\alpha^0+\alpha^1) - m_x^1 \mod 2^{k+2s}\\
&= x\cdot\alpha^1 - m_x^1 + x\cdot\alpha^0 \mod 2^{k+2s},
\end{aligned}$$

where $x \in \{a, b, c, \hat{a}, \hat{c}\}$. We now use notations in Fig. 2 for consistency check, which is consistent with the steps in Π^{Auth} to verify the correctness of the generated MACs. Upon receiving v^0 from S_0, $\mathcal{A}$ computes x_t^0 as:

$$x_t^0 = v^0 - \sum_{h=0}^{t-1} x_h^0\cdot r_h \mod 2^{k+2s},$$

where $x_h \in \{a, b, c, \hat{a}, \hat{c}\}$ and all r_h are public. $\mathcal{A}$ can then decompose d^0 into

$$d^0 = \underbrace{\sum_{h=0}^{t}(x_h\cdot\alpha^1 - m_{x_h}^1)\cdot r_h}_{\gamma} + \underbrace{\sum_{h=0}^{t} x_h\cdot r_h}_{\beta}\cdot\alpha^0 \mod 2^{k+2s}$$

Thus, $\mathcal{A}$ can solve α^0 via the following equation:

$$\begin{aligned}
z^0 &= d^0 - v\cdot\alpha^0 \mod 2^{k+2s}\\
&= (\gamma + \beta\cdot\alpha^0) - v\cdot\alpha^0 \mod 2^{k+2s}\\
&= \gamma + (\beta - v)\cdot\alpha^0 \mod 2^{k+2s}
\end{aligned}$$

In the real protocol execution, instead of receiving $a^0, b^0, c^0, \hat{a}^0, \hat{c}^0 \in \mathbb{Z}_{2^{k+s}}$, $\mathcal{A}$ actually receives $a^0, b^0, c^0, \hat{a}^0, \hat{c}^0 \in \mathbb{Z}_{2^k}$, where the first s bits are masked by some randomness. However, $\mathcal{A}$ can still infer α^0, since the equations above still hold for the last k bits. Thus, we conclude that the current implementation in [46] is not secure against the server-client collusion.

5.2 Consistency Check Details

In Π^{InCom}. We now analyse 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 [24] 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:

$$\gamma_h = m_h - \alpha\cdot x_h \mod 2^{k+s} \tag{1}$$

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

$$\sum_{h=0}^{t-1} d_h = \sum_{h=0}^{t-1}(\alpha\cdot x_h + \gamma_h)\cdot r_h + \alpha\cdot x_t + \gamma_t \mod 2^{k+2s} \tag{2}$$

Now different from [24], $\mathcal{A}$ cannot introduce any error to c, since both honest S_0 and S_1 will reconstruct $v = v^0 + v^1 \bmod 2^{k+2s}$ correctly. To pass the consistency check, there must be an error

$e \in \mathbb{Z}_{2^{k+2s}}$ introduced to the equation to compensate the errors γ_h we have defined above, such that:

$$\begin{aligned} 0 &= z^0 + z^1 + e \mod 2^{k+2s} \\ \Leftrightarrow -e &= \sum_{h=0}^{t-1} d_h - \alpha \cdot (\sum_{h}^{t-1} x_h \cdot r_h + x_t) \mod 2^{k+2s} \\ &= \sum_{h=0}^{t-1} \gamma_h \cdot r_h + \gamma_t \mod 2^{k+2s} \end{aligned} \tag{3}$$

Claim 5.1. *Suppose there is at least one non-zero component γ_h mod 2^{k+s}, then the probability of passing the check is no more than 2^{-s}.*

Proof. Without loss of generality, we suppose $\gamma_0 \neq 0 \bmod 2^{k+s}$, then we have:

$$\gamma_0 \cdot r_0 = \underbrace{\sum_{h=1}^{t-1} \gamma_h \cdot r_h + \gamma_t}_{S} \mod 2^{k+2s} \tag{4}$$

Let 2^v be the largest power of two dividing γ_0, we know that $v < k+s$, since $\gamma_0 \neq 0 \bmod 2^{k+s}$. Therefore, we know that $\frac{\gamma_0}{2^v}$ is odd and has multiplicative inverse modulo $k + 2s - v$. We have:

$$\begin{aligned} r_0 \cdot \frac{\gamma_0}{2^v} &= \frac{S}{2^v} \mod 2^{k+2s-v} \\ r_0 &= \frac{S}{2^v} (\frac{\gamma_0}{2^v})^{-1} \mod 2^{k+2s-v} \end{aligned} \tag{5}$$

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}. □

In $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$. We then analyse the consistency check of protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ in the *dishonest majority* setting. 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 value to be α^j and $\mathbf{x}^j$, then we define errors as:

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

Note that a corrupted C_i can introduce an error $\delta = \mathbf{x} - \mathbf{x}^0 - \mathbf{x}^1$ to the protocol. Without loss of generality, we always set this error as $\delta^1 = \hat{\mathbf{x}}^1 - \mathbf{x}^1$, if S_0 is not corrupted at the same time. Otherwise if both C_i and S_0 are corrupted, we set this error as $\delta^0 = \hat{\mathbf{x}}^0 - \mathbf{x}^0$. We observe following corruption cases:

1. S_0 is corrupted: $\mathcal{A}$ can introduce both type of errors to the equation by using incorrect $\hat{\alpha}^0$ and $\hat{\mathbf{x}}^0$ while executing $\mathcal{F}^{\mathsf{vOLE}}$ with S_1 and C_i.
2. S_1 is corrupted: $\mathcal{A}$ can only use an incorrect $\hat{\alpha}^1$ while executing $\mathcal{F}^{\mathsf{vOLE}}$ with S_0.
3. C_i is corrupted: $\mathcal{A}$ can send an incorrect $\hat{\mathbf{x}}^1$ to S_1.
4. S_0 and C_i are corrupted: $\mathcal{A}$ can use an incorrect $\hat{\mathbf{x}}^0$ while executing $\mathcal{F}^{\mathsf{vOLE}}$ with S_1.
5. S_1 and C_i are corrupted: $\mathcal{A}$ can use an incorrect $\hat{\alpha}^1$ while executing $\mathcal{F}^{\mathsf{vOLE}}$ with S_0.

Let c denote the indexes of corrupted parties. We summarize errors to the sum of MAC shares:

$$\begin{aligned} \mathbf{m}^0 + \mathbf{m}^1 &= \alpha^1 \cdot \mathbf{x}^1 + \sum_j \mathbf{a}^j - \mathbf{b}^j + \sum_{j \notin c} \gamma^j \cdot \mathbf{x}^j + \sum_{j \notin c} \alpha^j \cdot \delta^j \\ &= \alpha \cdot \mathbf{x} + \sum_{j \notin c} \gamma^j \cdot \mathbf{x}^j + \sum_{j \notin c} \alpha^j \otimes \delta^j \end{aligned} \tag{6}$$

which ends up with the same error analysis provided in [24]. So we directly derive two claims from [24]:

Claim 5.2. *If at least one $\gamma^j \neq 0$ where $j \notin c$, then the probability of passing the check is no more than $2^{-s+\log 2}$.*

Claim 5.3. *Suppose $\gamma^j = 0$ for all $j \notin c$, and δ^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 $2^{-s+\log s}$.*

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

While modeling ideal functionalities for SPD$\mathbb{Z}_{2^k}$, we found a subtlety that the global MAC key α must be explicitly chosen by the functionality, otherwise the functionality is not able to compute the authentication MAC of the output. In $\mathcal{F}^{\mathsf{InCom}}$ (and $\mathcal{F}^{\mathsf{MAC}}$ described in Fig. 12), α is chosen at the initialization stage by the functionality. We show an example by modeling a B2A functionality $\mathcal{F}^{\mathsf{B2A}}$ (or any functionality $\mathcal{F}$) without the initialization stage. Now $\mathcal{F}^{\mathsf{B2A}}$ must extract α by itself after receiving the input $[x]_2$ as $\alpha \leftarrow (m^0_x + m^1_x) \cdot (x^0 + x^1)^{-1} \bmod 2^{1+s}$, where $(x^j, m^j_x) \in (\mathbb{Z}_{2^{1+s}}, \mathbb{Z}_{2^{1+s}})$. However, α is unique only if $x \leftarrow (x^0 + x^1) \in \mathbb{Z}_{2^{1+s}}$ has a multiplicative inverse over $\mathbb{Z}_{2^{1+s}}$. Since we know that there exists at least an α satisfying the equation $\alpha \cdot (x^0 + x^1) \equiv_{1+s} (m^0_x + m^1_x)$, a "bad" case is that there exists $\{\alpha_i\}$ whose elements all satisfy the above equation. Thus, the simulation will fail if $\mathcal{F}^{\mathsf{B2A}}$ chooses the wrong α to compute the MAC of the output. In addition, simply add the initialization stage to each functionality does not solve the problem, since this would allow parties to initialize inconsistent α for different functionalities. In [24, 26], a crucial modeling is to summarize all randomness generation functionalities in a single preprocessing functionality $\mathcal{F}^{\mathsf{Pre}}$ to manage the global MAC key generation and leave functional computation in the main protocol. In AlphaFL, we build a wrapper functionality $\mathcal{F}^{\mathsf{Wrap}}$ described in Fig. 16, which accepts commands as defined in $\mathcal{F}^{\mathsf{InCom}}$, $\mathcal{F}^{\mathsf{SqGen}}$ and $\mathcal{F}^{\mathsf{TripGen}}$ (formally defined in Fig. 7, Fig. 8 and Fig. 13, respectively). In addition, we include $\mathcal{F}^{\mathsf{RanBitGen}}$ described in Fig. 15 into $\mathcal{F}^{\mathsf{Wrap}}$, where $\mathcal{F}^{\mathsf{Wrap}}$ outputs authenticated bit shares to servers. We show an overview of different protocols and functionalities in Fig. 6.

5.4 Theorems and Proofs

We formally describe the functionality $\mathcal{F}^{\mathsf{InCom}}$ in Fig. 7 and the functionality $\mathcal{F}^{\mathsf{SqGen}}$ in Fig. 8. We use "$\star$" 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. Due to the subtlety above, we directly invoke Π^{SiSelect} and Π^{MSB} in Fig. 9 (like in [26]) without abstracting them as functionalities. Due to space limitations, we present the theorems in this section and refer the reader to Appendix C for detailed proofs.

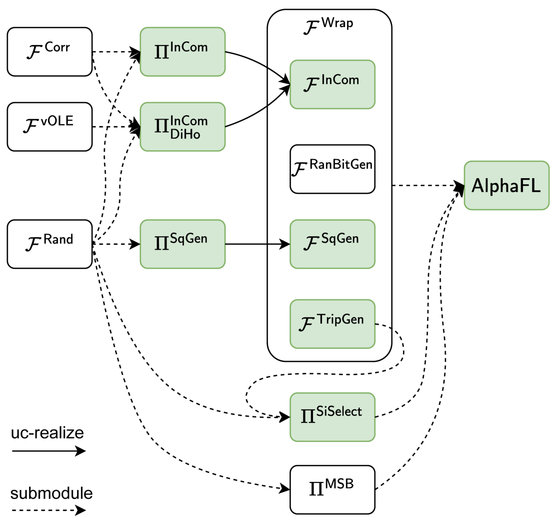

Figure 6: Relationship between protocols and functionalities. Discussed components in this work are highlighted in green. $\mathcal{F}^{\mathsf{Rand}}$ is a submodule of all protocols due to consistency check or Open and MAC check procedure, respectively.

Theorem 5.4. *Protocol Π^{InCom} shown in Fig. 2 uc-realizes $\mathcal{F}^{\mathsf{InCom}}$ described in Fig. 7 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 $\mathsf{C}_c \subseteq \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$ or a server $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$, with static corruption.*

Theorem 5.5. *Protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ shown in Fig. 3 uc-realizes $\mathcal{F}^{\mathsf{InCom}}$ described in Fig. 7 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.*

Theorem 5.6. *Protocol Π^{SqGen} shown in Fig. 4 uc-realizes $\mathcal{F}^{\mathsf{SqGen}}$ described in Fig. 8 in the $\mathcal{F}^{\mathsf{TripGen}}$, $\mathcal{F}^{\mathsf{Rand}}$-hybrid model, in the presence of a malicious adversary who can corrupt a server $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$, with static corruption.*

6 Federated Learning with Malicious Security

AlphaFL is built on four core components: preprocessing, input commitment, filtering, aggregation and MAC check. We represent detailed outline of the AlphaFL procedure during each training round in Fig 9. For clarity, we only discuss the more complicated case in the remaining part, where we allow a malicious server to collude with multiple clients.

a) Preprocessing. The preprocessing stage is independent of the real-time input of the online stage. For each C_i, suppose that the local gradient vector has t elements, servers generate $w \cdot t$ pairs of random arithmetically and boolean shared bits $([b_i], [b'_i]_2)$ to support the B2A protocol. Servers then do the same to generate n pairs of shared bits $([p_i], [p'_i]_2)$. Besides, servers generate t pairs of square correlations $([a_h], [d_h])$ to support the L_2-Norm

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

Initialize: For each $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$, upon receiving $(\mathsf{Init}, \mathsf{P}_i, \mathsf{sid})$ from $\mathsf{P}_i \in \{\mathsf{C}_i, \mathsf{S}_0, \mathsf{S}_1\}$:

1. If $\mathsf{P}_c \in \{\mathsf{S}_0, \mathsf{S}_1\}$, 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 $\alpha \leftarrow \alpha^c + \alpha^{c-1} \mod \mathbb{Z}_{2^{k+s}}$.
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 $\mathbf{m} \leftarrow \mathbf{x} \cdot \alpha \mod 2^{k+s}$.
2. Wait to receive $\mathbf{m}^c$ from $\mathcal{S}$, then set $\mathbf{m}^{c-1} \leftarrow \mathbf{m} - \mathbf{m}^c$.

InCom: For each $\mathsf{C}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}\}$, upon receiving $(\mathsf{InCom}, \mathsf{P}_i, \mathsf{sid})$ where $\mathsf{P}_i \in \{\mathsf{C}_i, \mathsf{S}_0, \mathsf{S}_1\}$:

1. If $\mathsf{P}_c \in \{\mathsf{S}_0, \mathsf{S}_1\}$, wait to receive $(\mathbf{x}^c, \mathbf{m}^c) \in (\mathbb{Z}^t_{2^{k+s}}, \mathbb{Z}^t_{2^{k+s}})$ from the adversary and $(\mathbf{x}, \mathsf{C}_i, \mathsf{sid})$ from C_i where $\mathbf{x} \in \mathbb{Z}^t_{2^{k+s}}$, set $\mathbf{x}^{c-1} \leftarrow \mathbf{x} - \mathbf{x}^c$.
2. If C_i is corrupted (individually or simultaneously), wait to receive $(\mathbf{x}^0, \mathbf{x}^1)$ from the adversary, compute $\mathbf{x} \leftarrow \mathbf{x}^0 + \mathbf{x}^1 \mod 2^{k+s}$.
3. Send $\mathbf{x}^j$ 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. Run $\mathsf{MACGen}(\mathbf{x})$. Send $\mathbf{m}^j$ to S_j.

Figure 7: Input commitment functionality $\mathcal{F}^{\mathsf{InCom}}$

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

The functionality $\mathcal{F}^{\mathsf{SqGen}}$ has all the same features as $\mathcal{F}^{\mathsf{InCom}}$, with the additional command:

Square Correlation Generation: Upon receiving $(\mathsf{SqCoGen}, \mathsf{P}_i, \mathsf{sid})$ from $\mathsf{P}_i \in \{\mathsf{C}_i, \mathsf{S}_0, \mathsf{S}_1\}$ (or $\mathsf{P}_i \in \{\mathsf{S}_0, \mathsf{S}_1\}$ in 2PC):

1. If $\mathsf{P}_c \in \{\mathsf{S}_0, \mathsf{S}_1\}$, 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 $a \leftarrow a^c + a^{c-1} \mod \mathbb{Z}_{2^k}$. Compute $d \leftarrow a \cdot a \mod \mathbb{Z}_{2^k}$.
4. Sample $r \xleftarrow{\$} \mathbb{Z}_{2^s}$, compute $d \leftarrow d + 2^k \cdot r \mod 2^{k+s}$, set $d^{c-1} \leftarrow d - d^c$.
5. Send (a^j, d^j) to S_j.
6. Run $\mathsf{MACGen}(\{a, d\})$. Send (m^j_a, m^j_d) to S_j.

Figure 8: Square correlation generation functionality $\mathcal{F}^{\mathsf{SqGen}}$

AlphaFL

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.
Output: 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 $\mathbf{b} \mod 2^k \in \mathbb{Z}_2^{w*t}$. 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 $b_i'^j \leftarrow b_i^j \mod 2^{1+s}$ and $m_{b_i'}^j \leftarrow m_{b_i}^j \mod 2^{1+s}$.
2. S_j sends $(\mathsf{RanBitGen}, \mathsf{S}_j, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, receives $[\mathbf{p}]^j$ where $\mathbf{p} \mod 2^k \in \mathbb{Z}_2^n$. 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 $p_i'^j \leftarrow p_i^j \mod 2^{1+s}$ and $m_{p_i'}^j \leftarrow m_{p_i}^j \mod 2^{1+s}$.
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 $d_h \leftarrow a_h \cdot a_h \mod 2^k$.
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 $\gamma \leftarrow \delta \cdot \delta \mod 2^k$.

Input Commitment: For each client C_i:

1. C_i locally computes the gradient update $\mathbf{u}_i \leftarrow \{u_0, ..., u_{t-1}\}$, where $u_h \leftarrow (u_{h,0}, ..., u_{h,w-1}) \in \mathbb{Z}_2^w$.
2. S_0, S_1 and C_i send $(\mathsf{InCom}, \mathsf{P}_i, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{Wrap}}$, servers receive $[\mathbf{u}_i]_2 \in \mathbb{Z}_2^{w*t}$.

L_∞-Norm and B2A: For each C_i:

1. Servers run the **Open** phase of **BatchCheck** to reconstruct $\mathbf{c} \leftarrow [\mathbf{u}_i]_2 + [\mathbf{b}']_2$, 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 $[u_{h,i}] \leftarrow c_{w\cdot h+i} + [b_{w\cdot h+i}] - 2 \cdot c_{w\cdot h+i} \cdot [b_{w\cdot h+i}]$.
3. Servers finally compute $[u_h] \leftarrow \sum_{i=0}^{w-2} 2^i \cdot [u_{h,i}] + \sum_{i=w}^{k-1} 2^i \cdot [u_{h,w}]$.

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 $f_h \leftarrow [u_h] - [a_h]$.
2. S_0 computes $[v]^0 \leftarrow \sum_{h=0}^{t-1} [d_h]^0 + 2f_h \cdot [a_h]^0 - f_h \cdot f_h$, S_1 computes $[v]^1 \leftarrow \sum_{h=0}^{t-1} [d_h]^1 + 2f_h \cdot [a_h]^1$.

L_2-Norm Check: Servers compute $[\beta] \leftarrow \frac{\sum [v_i]}{n}$ for $i \in \{0, ..., n-1\}$ and $[\beta^2]$ using $([\delta], [\gamma])$ as above, then for each C_i:

1. Servers compute $[y] \leftarrow [v] - [\beta^2]$.
2. Servers run Π^{MSB} with input $[y]$ to extract the authenticated shared sign bit $[s]_2$ of y ($s = 1$ indicates that $v < \beta^2$).

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 $e_i \leftarrow [s_i]_2 + [p_i']_2$. Servers compute $[s_i] \leftarrow e_i + [p_i] - 2 \cdot e_i \cdot [p_i]$.
3. Servers run the **Open** phase of **BatchCheck** to reconstruct $\tau' \leftarrow \sum_{i=0}^{n-1} [s_i]$. If $\tau' < \tau$, servers abort the computation.
4. Otherwise, servers compute $[\mathcal{U}_q] \leftarrow \frac{1}{\tau'} \cdot \sum_{i=0}^{n-1} [\mathbf{z}_i]$, where $\mathbf{z}_i = \{0\}$ 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 explained in Fig. 20.

Figure 9: Maliciously secure aggregation protocol in AlphaFL

computation. Note that $\Pi_{\mathsf{Diho}}^{\mathsf{InCom}}$, Π^{MSB} and Π^{SiSelect} also involve preprocessing computation, we refer to Fig. 3, Fig. 22 and Fig. 5 for more details.

b) Input Commitment. During this stage, each client C_i submits its authenticated input to servers. We set aside the straightforward method where C_i would simply share its input with the servers. This is flawed since there is no guarantee that a malicious server will use the precise input share received from C_i. We present the solutions in Section 4.1.

c) Filtering. As mentioned in Section 1, we apply L_∞-Norm and L_2-Norm checks to filter malicious gradient update sent by compromised clients. Recall that by applying L_∞-Norm check, each element u_h of C_i's local gradient update $\mathbf{u}_i$ is bounded by 2^{w-1}. In the **Input Commitment** stage, C_i shares overall $w \cdot t$ authenticated bits to servers for $\mathbf{u}_i$ with size t, where each u_h is decomposed into w bits. Thus, the L_∞-Norm is automatically maintained while executing $\Pi_{\mathsf{Diho}}^{\mathsf{InCom}}$ (with $k = 1$), where the last authenticated bit is considered to be the sign bit of u_h. In order to perform the L_2-Norm check, servers have to first perform a boolean-to-arithmetic conversion (B2A) supported by the pre-computed authenticated random bits and then compute the square of the L_2-Norm of $\mathbf{u}_i$. Finally, servers execute the protocol Π^{MSB} described in Fig. 22 to securely extract the most significant bit of $[y] \leftarrow [v] - [\beta^2]$.

d) Aggregation and MAC Check. After completing the L_2-Norm check, servers run Π^{SiSelect} to compute $[\mathbf{z}_i]$ and run the B2A protocol to compute $[s_i]$ from $[s_i]_2$. If $\tau' > \tau$, servers run the **MAC check** phase of **BatchCheck** to verify all values that have been opened so far. If the MAC check passes, servers proceed to run the **SingleCheck** to reconstruct the authenticated aggregation result $\mathcal{U}_q$ and use $\mathcal{U}_q$ to update the global model $\mathcal{M}_{q-1}$. They finally send the latest version of the model $\mathcal{M}_q$ to all clients.

7 Evaluation

In this section, we quantify the computational and communication overhead of AlphaFL. We do not benchmark the robustness of norm-based poisoning defense, as it is tangential to this work and has already been evaluated in RoFL [59]. We focus on the input commitment and the secure aggregation tasks. Similar to prior work, we omit the local training phase and do not include preprocessing time in any of the benchmark results in this section.

7.1 Experiment Setup

We evaluated all the tasks on an `Ubuntu 24.04.1 LTS` VM with 48 vCPUs and 128GB RAM, hosted by a workstation with 2 `Intel(R) Xeon(R) Gold 5317` CPUs. All clients and servers are executed as separate processes. The network is configured with `1ms` round-trip latency and `10Gbps` bandwidth. We consider the gradient updates as 32-bit values, and perform the L_2-Norm computation and aggregation over 64-bit ring, i.e., $w = 32$, $k = 64$, $s = 63$. We choose $s = 63$ instead of $s = 64$ for better memory alignment in our input commitment implementation.

7.2 Implementation

We implemented AlphaFL in two parts, both based on MP-SPDZ `v0.3.9` [46]. The most secure aggregation building blocks are implemented in Python, with MP-SPDZ high-level interface, except the novel input commitment protocols, which are written in C++, using MP-SPDZ lowest-level interface, as the required functionalities are not available at higher level. The source code is publicly available at https://github.com/Barkhausen-Institut/AlphaFL.

7.3 Comparison against a Single Aggregator

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

7.3.1 With Client-Poisoning Resilience.

Baseline. We consider RoFL [59] at commit `c1a0c13` for the one-server setting with $w = 16$ as a showcase. The total runtime is calculated by summing up the recorded gradient encryption, proof generation, sending time at one client with the aggregation and proof verification time at the server. The total data sent are the sum of the data sent by each client.

Table 1: End-to-End runtime and total data sent comparison against the one-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

Parameter Sizes. For this setting, we consider three parameter sizes, each associated with a distinct dataset: a) `LeNet5` [52] trained on `CIFAR10-S`, b) `ResNet18` [37] trained on the `CIFAR10` [51] and c) `LSTM` [38] trained on the `Shakespeare` [18]. We only use $n = 4$ clients in our setup, since RoFL [59] crashes with more clients during our evaluation.

Comparison. Table 1 shows the results of our end-to-end benchmarks. RoFL [59] has an advantage in traffic volume thanks to its one-server setting. In the two-server setting like AlphaFL, each client needs to communicate with two servers at the same time. Besides, there are also communications between two servers. It is not surprising that the total data sent is more than double of that in the one-server setting.

However, regardless of 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. When we take a closer look at RoFL runtime, we notice that the majority of time is spent on generating ZKPs at the client side. ZKPs are still too expensive for this task.

7.3.2 Without Client-Poisoning Resilience.

Baselines. For completeness, we also benchmark two state-of-the-art one-server frameworks, which support output integrity verification without poisoning resilience:

- VeriFL [34] at commit `8235a87` with the same settings as in their paper: bit length 24, half of the number of clients as the threshold of secret sharing, batch size 1, no dropouts. The total runtime is calculated by summing up all recorded duration at both client and server. The total data sent is calculated by summing all recorded data at each client and multiplying that by the number of clients.
- e-SeaFL [8] at commit `41ede38` in the malicious setting with 3 assisting nodes as suggested. The total runtime is calculated by summing up all recorded duration at assisting node, client and server. Similarly, the total data sent includes the outbound traffic at all assisting nodes, clients and server. Setup phase is always excluded.

Parameter Sizes. For this setting, we consider three parameter sizes: a) gradient size $t = 62k$ with $n = 4$ clients, corresponding to `LeNet5` [52] trained on `CIFAR10-S`, referring to the rows in Table 1,

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

Table 2: End-to-End runtime and total data sent comparison against the one-server framework VeriFL and e-SeaFL

#Cli	#Par	Alpha-Ho	Alpha-DiHo	VeriFL	e-SeaFL
		Runtime (second)			
4	62k	0.58	7.76	5.78	97.20
20	100k	3.6	56.14	9.39	163.17
40	100k	7.15	111.11	9.50	194.18
		Total Data Sent (MB)			
4	62k	200	8,201	2.37	2.60
20	100k	1,610	66,122	19.25	16.41
40	100k	3,218	132,242	38.88	31.67

b) gradient size $t = 100k$ with $n \in \{20, 40\}$ clients, referring to the rows in Table 3 and Table 4. We only use these two gradient sizes since VeriFL [34] crashes with larger sizes during our evaluation.

Comparison. Table 2 shows the results of our end-to-end benchmarks. Both VeriFL [34] and e-SeaFL [8] have extreme communication efficiency. The one-server setting, short bit length of secret sharing and efficient masking techniques contribute to that.

However, despite of low communication, both VeriFL and e-SeaFL are slower than AlphaFL-Ho. e-SeaFL is even much slower than AlphaFL-DiHo. Both VeriFL and e-SeaFL are hindered by the verification in the end, which is highly related to the gradient size, less affected by the number of clients.

7.4 Comparison in the Two-server Setting

Baselines. We benchmark AlphaFL-Ho and AlphaFL-DiHo along with two state-of-the-art two-server frameworks:

- Elsa [76] at commit eabcfd2 for the two-server setting 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+ [1] at commit eabcfd2 (provided by Elsa [76]) for the two-server setting with $w = 32$, $l = 64$. The total runtime and the total data sent are evaluated in the same way as in Elsa [76].

Parameter Sizes. In order to analyze the scalability, we vary both the size of the gradient vectors and the number of clients while benchmarking different frameworks. We use two gradient sizes (#Params) $t \in \{100\text{k}, 300\text{k}\}$ to capture the scale of trained model. And we set the number of clients (#Clients) to $n \in \{10, 20, 30, 40\}$.

Comparison. We show the end-to-end runtime in Table 3. In general, running all protocols with malicious security in AlphaFL-Ho consumes more runtime than Elsa, but less than Prio+:

- Compared to Elsa, AlphaFL-Ho brings 34% − 79% more runtime overhead for $t = 100\text{k}$ and 25% − 75% more for $t = 300\text{k}$.
- Compared to Prio+, AlphaFL-Ho is 3.32 − 6.30× as fast for $t = 100\text{k}$ and 5.70 − 9.32× as fast for $t = 300\text{k}$.

Furthermore, parties execute the silent select protocol in both AlphaFL-Ho and AlphaFL-DiHo, which is skipped in Elsa and Prio+, resulting in additional runtime. Since the aggregation protocol executed in 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, even though our protocol cuts the online computation in half (Section 4.1.2).

Table 3: End-to-End runtime (in Seconds) comparison in 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	1.9	28.67 (24.84)	1.42	11.97
20	100k	3.6	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	16.1	250.68 (230.63)	9.6	N/A
40	300k	21.39	338.21 (311.63)	12.23	N/A

Table 4: End-to-End total data sent (in GBs) comparison in 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

We also show the communication overhead in Table 4. We observe that the communication volume required in AlphaFL-Ho is very close to Elsa and Prio+, while AlphaFL-DiHo requires $\sim 40\times$ communication compared to AlphaFL-Ho. This is due to the execution of the vOLE functionality $\mathcal{F}^{\mathsf{vOLE}}$ within $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$. We will elaborate more in the next section.

7.5 Breakdown

To better analyze the performance of each modular protocol executed in AlphaFL-Ho and AlphaFL-DiHo, we provide the breakdown of runtime and traffic in Fig. 10 and Fig. 11. We first set the gradient size to 100k and vary the number of clients, then we set the number of clients to 20 and vary the gradient size.

In the honest majority setting, the runtimes of the input commitment protocol, the boolean-to-arithmetic protocol (B2A) and the L_2-Norm check are almost identical as shown in (a) and (c) of Fig. 10, while the communication overhead of Π^{InCom} dominates all other modular protocols as shown in (b) and (d) of Fig. 10. This indicates that the local computation of the B2A protocol and the L_2-Norm check takes up a significant portion of their total runtime.

In the dishonest majority setting, the computation and communication overhead of $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ dominates all other parts as shown in Fig. 11. Regarding the runtime, the execution of $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ takes approximately 96% of the total runtime. The time proportion for executing $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ remains almost unchanged for different gradient

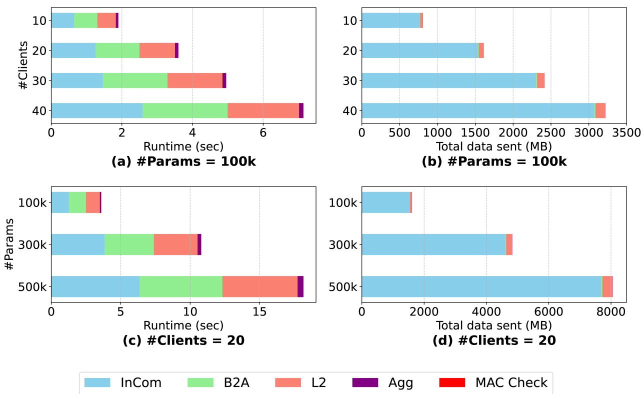

Figure 10: Runtime and total data sent breakdown of AlphaFL-Ho

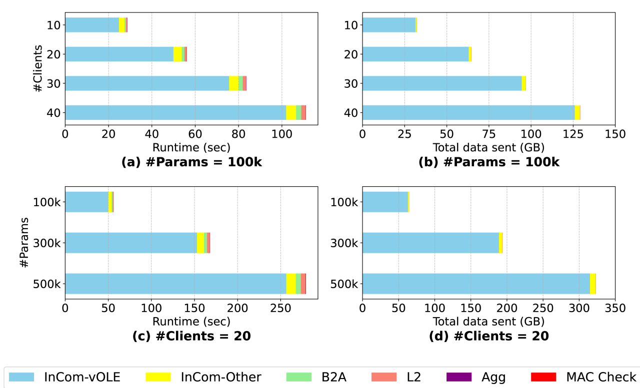

Figure 11: Runtime and total data sent breakdown of AlphaFL-DiHo

sizes and number of clients. Meanwhile, the absolute runtime and communication volume of $\Pi^{\text{InCom}}_{\text{Diho}}$ increases proportionally to the parameter sizes and the number of clients, while the communication cost of the rest can be almost ignored.

When we further take $\Pi^{\text{InCom}}_{\text{Diho}}$ apart, we notice that vOLE consumes about 96% of the runtime and the traffic (we label the cost of vOLE as InCom-vOLE in Fig. 11). This also makes vOLE the most consuming part in the end-to-end evaluation, as shown in the AlphaFL-DiHo column in Table 3 and Table 4. We have already applied some transmission optimization by sending in smaller trunks, which boosts local processing and increases the network utilization a lot. We note that when executing vOLE, the network bandwidth is almost always 100% occupied. This means that further optimization needs to be done at the vOLE protocol level to reduce the traffic volume, which is not a main focus of our work.

7.6 Microbenchmarks

To show the efficiency improvement of our input commitment protocols and the silent select protocol, we compare our protocols against the original ones implemented in MP-SPDZ and provide microbenchmarks in Table. 5 and Table. 6. To reduce the side effect

Table 5: Input commitment comparison. N/A means that the compilation failed due to memory requirement.

#Params	Alpha-Ho	Alpha-DiHo	MP-SPDZ
	Runtime (second)		
62k CIFAR10-S	0.15	4.79	0.55
273k CIFAR10-L	0.61	24.07	2.27
818k SHAKESPEARE	1.95	70.74	N/A
	Total Data Sent (MB)		
62k CIFAR10-S	48	2,048	253
273k CIFAR10-L	211	9,063	1,124
818k SHAKESPEARE	628	27,009	N/A

Table 6: Select protocol comparison. The unit for runtime is second, and the unit for communication is MB.

#Params	Silent Select		Classic Select	
	Runtime	Comm.	Runtime	Comm.
62k CIFAR10-S	0.03	1.00	0.08	1.99
273k CIFAR10-L	0.11	4.42	0.41	8.81
818k SHAKESPEARE	0.32	13.18	1.00	26.27

of parallel execution, we analyze the single client case and vary the parameter size to benchmark the total runtime and total data sent.

Input commitment protocol. We note that Damgård et al. [25] propose input and output protocols for non-computing parties in the honest majority setting, which is highly related to our input commitment protocol Π^{InCom}. Marcel Keller [46] now provides a modified version in MP-SPDZ, which we simply label as MP-SPDZ in Table 5. Π^{InCom} is at least 3.7× as fast as MP-SPDZ, and requires less than $\frac{1}{5}$ of MP-SPDZ's communication. On the other hand, although in the dishonest majority setting $\Pi^{\text{InCom}}_{\text{Diho}}$ achieves a better efficiency than the input protocol in [24] as shown in Section 4.1.2, $\Pi^{\text{InCom}}_{\text{Diho}}$ is still 31.93 to 39.46 × as slow as Π^{InCom}.

Silent select protocol. We implement and benchmark our silent select protocol Π^{SiSelect} against the classic select protocol as implemented in MP-SPDZ. As shown in Table. 6, Π^{SiSelect} is $2.67 - 3.72\times$ as fast as the classic select. In addition, the communication cost required in Π^{SiSelect} is half of what the classic select requires.

8 Conclusion

In this work, we propose AlphaFL: an efficient aggregation protocol in two-server setting with malicious security and poisoning resilience. We design efficient input commitment protocols, and we propose an efficient silent select protocol to reduce online computation cost. We further introduce a simple way to generate square correlation on ring. We prove our protocol secure in the UC framework, and we showcase a subtlety while modeling functionalities for the SPD$\mathbb{Z}_{2^k}$ scheme. Aiming at achieving complete malicious security, AlphaFL exhibits a similar efficiency compared to state-of-the-art frameworks in the non-collusion case and stimulates more future work in the collusion case.

Acknowledgments

Yufan Jiang: This work was supported by funding from the subtopic Methods for Engineering Secure Systems (46.23.01) under the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs. The authors from the Barkhausen Institut have been supported by the Federal Ministry of Research, Technology, and Space of Germany through the SEMECO project (Grant No. 03ZU1210AA). They are also financed based on the budget passed by the Saxonian State Parliament in Germany.

We would like to thank Marcel Keller (MP-SPDZ [46]) and Mayank Rathee (Elsa [76]) for their prompt and helpful email responses to our technical queries. Their support was valuable to our work.

References

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

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

[3] Sana Awan, Bo Luo, and Fengjun Li. 2021. 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, 455–475.

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

[5] Soumya Banerjee, Sandip Roy, Sayyed Farid Ahamed, Devin Quinn, Marc Vucovich, Dhruv Nandakumar, Kevin Choi, Abdul Rahman, Edward Bowen, and Sachin Shetty. 2024. 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, 635–640.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] Xiaoyu Cao, Minghong Fang, Jia Liu, and Neil Zhenqiang Gong. 2020. Fltrust: Byzantine-robust federated learning via trust bootstrapping. *arXiv preprint arXiv:2012.13995* (2020).

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

[22] Amrita Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. 2022. EIFFeL: Ensuring Integrity for Federated Learning. 2535–2549. https://doi.org/10.1145/3548606.3560611

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

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

[25] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. 2017. Confidential Benchmarking Based on Multiparty Computation. In *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, 169–187.

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

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

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

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

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

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

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

[33] Xueluan Gong, Yanjiao Chen, Qian Wang, and Weihan Kong. 2022. Backdoor attacks and defenses in federated learning: State-of-the-art, taxonomy, and future directions. *IEEE Wireless Communications* 30, 2 (2022), 114–121.

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

[35] Changhee Hahn, Hodong Kim, Minjae Kim, and Junbeom Hur. 2021. Versa: Verifiable secure aggregation for cross-device federated learning. *IEEE Transactions on Dependable and Secure Computing* 20, 1 (2021), 36–52.

[36] Sebastian Hasler, Toomas Krips, Ralf Küsters, Pascal Reisert, and Marc Rivinius. 2023. Overdrive LowGear 2.0: Reduced-Bandwidth MPC without Sacrifice. *Cryptology ePrint Archive* (2023).

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

[38] S Hochreiter. 1997. Long Short-term Memory. *Neural Computation MIT-Press* (1997).

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

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

[41] Chao Huang, Yanqing Yao, Xiaojun Zhang, Da Teng, Yingdong Wang, and Lei Zhou. 2022. 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, 582–589.

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

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

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

[45] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2021. Advances and open problems in federated learning. *Foundations and trends® in machine learning* 14, 1–2 (2021), 1–210.

[46] Marcel Keller. 2020. 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, 1575–1590.

[47] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 830–842.

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

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

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

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

[52] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.

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

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

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

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

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

[58] Fucai Luo, Saif Al-Kuwari, and Yong Ding. 2022. SVFL: Efficient secure aggregation and verification for cross-silo federated learning. *IEEE Transactions on Mobile Computing* 23, 1 (2022), 850–864.

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

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

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

[62] Yunlong Mao, Xinyu Yuan, Xinyang Zhao, and Sheng Zhong. 2021. 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, 476–496.

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

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

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

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

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

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

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

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

[71] 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. 2022. {FLAME}: Taming backdoors in federated learning. In *31st USENIX Security Symposium (USENIX Security 22)*. 1415–1432.

[72] 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. 2021. Flguard: Secure and private federated learning. *Crytography and Security* Preprint (2021).

[73] Jaehyoung Park and Hyuk Lim. 2022. Privacy-preserving federated learning using homomorphic encryption. *Applied Sciences* 12, 2 (2022), 734.

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

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

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

[77] Yanli Ren, Yerong Li, Guorui Feng, and Xinpeng Zhang. 2022. Privacy-enhanced and verification-traceable aggregation for federated learning. *IEEE Internet of Things Journal* 9, 24 (2022), 24933–24948.

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

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

[80] Virat Shejwalkar, Amir Houmansadr, Peter Kairouz, and Daniel Ramage. 2022. 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, 1354–1371.

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

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

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

[84] Gan Sun, Yang Cong, Jiahua Dong, Qiang Wang, Lingjuan Lyu, and Ji Liu. 2021. Data poisoning attacks on federated machine learning. *IEEE Internet of Things Journal* 9, 13 (2021), 11365–11375.

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

[86] Vale Tolpegin, Stacey Truex, Mehmet Emre Gursoy, and Ling Liu. 2020. 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, 480–501.

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

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

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

[90] Ning Wang, Yang Xiao, Yimin Chen, Yang Hu, Wenjing Lou, and Y Thomas Hou. 2022. 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*. 946–958.

[91] Yong Wang, Aiqing Zhang, Shu Wu, and Shui Yu. 2022. VOSA: Verifiable and oblivious secure aggregation for privacy-preserving federated learning. *IEEE Transactions on Dependable and Secure Computing* 20, 5 (2022), 3601–3616.

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

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

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

[95] Guowen Xu, Hongwei Li, Sen Liu, Kan Yang, and Xiaodong Lin. 2019. VerifyNet: Secure and verifiable federated learning. *IEEE Transactions on Information Forensics and Security* 15 (2019), 911–926.

[96] Guowen Xu, Hongwei Li, Yun Zhang, Shengmin Xu, Jianting Ning, and Robert H Deng. 2020. Privacy-preserving federated deep learning with irregular users. *IEEE Transactions on Dependable and Secure Computing* 19, 2 (2020), 1364–1381.

[97] Jian Xu, Shao-Lun Huang, Linqi Song, and Tian Lan. 2022. Byzantine-robust federated learning through collaborative malicious gradient filtering. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1223–1235.

[98] Gang Yan, Hao Wang, Xu Yuan, and Jian Li. 2023. 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. 10711–10719.

[99] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. 2018. Byzantine-robust distributed learning: Towards optimal statistical rates. In *International conference on machine learning*. Pmlr, 5650–5659.

[100] Haiyang Yu, Runtong Xu, Hui Zhang, Zhen Yang, and Huan Liu. 2023. EV-FL: Efficient Verifiable Federated Learning With Weighted Aggregation for Industrial IoT Networks. *IEEE/ACM Transactions on Networking* (2023).

[101] Chengliang Zhang, Suyi Li, Junzhe Xia, Wei Wang, Feng Yan, and Yang Liu. 2020. {BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 493–506.

[102] Zaixi Zhang, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. 2022. 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*. 2545–2555.

[103] Huadi Zheng, Haibo Hu, and Ziyang Han. 2020. Preserving user privacy for machine learning: Local differential privacy or federated machine learning? *IEEE Intelligent Systems* 35, 4 (2020), 5–14.

[104] Tianhang Zheng and Baochun Li. 2022. Poisoning attacks on deep learning based wireless traffic prediction. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 660–669.

[105] Yin Zhu, Junqing Gong, Kai Zhang, and Haifeng Qian. 2024. Malicious-Resistant Non-Interactive Verifiable Aggregation for Federated Learning. *IEEE Transactions on Dependable and Secure Computing* (2024).

A Ideal Functionalities

A.1 MAC Functionality $\mathcal{F}^{\mathsf{MAC}}$.

See Fig 12.

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

$\mathcal{F}^{\mathsf{MAC}}$ generates shares of a global MAC key and, on input shares of a value, distributes MAC shares of this value. Let P_c denote the corrupted party, and c is the index of the corrupted party.

Initialize: Upon receiving $(\mathsf{Init}, \mathsf{S}_j, \mathsf{sid})$ from $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$:

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 $\alpha \leftarrow \alpha^c + \alpha^{c-1} \mod \mathbb{Z}_{2^{k+s}}$.
3. Send α^j to S_j.

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

1. Compute $\mathbf{m} \leftarrow \mathbf{x} \cdot \alpha \mod 2^\ell$.
2. Wait to receive $\mathbf{m}^c$ from the adversary, then set $\mathbf{m}^{c-1} \leftarrow \mathbf{m} - \mathbf{m}^c$.

Authentication: Upon receiving $(\mathsf{MAC}, \ell, k, \mathsf{S}_j, \mathsf{sid})$ from $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$, where $x \in \mathbb{Z}_{2^k}$ and $\ell \geq k$:

1. Wait for the adversary to send a message $(\mathsf{guess}, \mathsf{S}_j, B_j)$ 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 $\mathsf{Auth}(\ell, \mathbf{x} = \{x_0, ..., x_{t-1}\})$ and then wait for the adversary to send either OK or Abort. If the adversary sends OK then send the MAC shares $\mathbf{m}^j$ to party S_j, otherwise abort.

Figure 12: Functionality $\mathcal{F}^{\mathsf{MAC}}$ [24]

A.2 Triple Generation Functionality $\mathcal{F}^{\mathsf{TripGen}}$.

See Fig. 13.

A.3 Vector Oblivious Linear Evaluation Functionality $\mathcal{F}^{\mathsf{vOLE}}$.

See Fig. 14.

A.4 Random Bit Generation Functionality $\mathcal{F}^{\mathsf{RanBitGen}}$.

See Fig. 15.

A.5 Wrap Functionality $\mathcal{F}^{\mathsf{Wrap}}$.

See Fig. 16.

A.6 Correlated Randomness Functionality $\mathcal{F}^{\mathsf{CR}}$.

See Fig. 17.

A.7 Correlated Randomness Functionality $\mathcal{F}^{\mathsf{CR,glo}}$.

See Fig. 18.

B Protocols

B.1 Batch Check

See Fig. 19.

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 $a \leftarrow a^c + a^{c-1} \mod \mathbb{Z}_{2^k}$ and $b \leftarrow b^c + b^{c-1} \mod \mathbb{Z}_{2^k}$. Compute $c \leftarrow a \cdot b \mod \mathbb{Z}_{2^k}$
3. Sample $r \xleftarrow{\$} \mathbb{Z}_{2^s}$, compute $c \leftarrow c + 2^k \cdot r \mod 2^{k+s}$, set $c^{c-1} \leftarrow c - c^c$.
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 $b \leftarrow b^c + b^{c-1} \mod 2^k \in \mathbb{Z}_2$.
2. Compute $a \leftarrow a^c + a^{c-1} \mod \mathbb{Z}_{2^k}$ and $b \leftarrow b^c + b^{c-1} \mod \mathbb{Z}_{2^k}$. Compute $c \leftarrow a \cdot b \mod \mathbb{Z}_{2^k}$
3. Sample $r \xleftarrow{\$} \mathbb{Z}_{2^s}$, compute $c \leftarrow c + 2^k \cdot r \mod 2^{k+s}$, set $c^{c-1} \leftarrow c - c^c$.
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 13: Triple generation functionality $\mathcal{F}^{\mathsf{TripGen}}$

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 $\mathbf{a} \leftarrow \mathbf{b} + \alpha \cdot \mathbf{x} \mod 2^\ell$.
3. If P_i is corrupted, receive $\mathbf{a} \in \mathbb{Z}_{2^\ell}^t$ from the adversary and compute $\mathbf{a} \leftarrow \mathbf{b} + \alpha \cdot \mathbf{x} \mod 2^\ell$.
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 14: Vector oblivious linear evaluation functionality $\mathcal{F}^{\mathsf{vOLE}}$ [24]

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

The functionality $\mathcal{F}^{\mathsf{RanBitGen}}$ has all the same features as $\mathcal{F}^{\mathsf{MAC}}$, with the additional command:

Random Bit Generation: Upon receiving $(\mathsf{RanBitGen}, \mathsf{S}_j, \mathsf{sid})$ from $\mathsf{S}_j \in \{\mathsf{S}_0, \mathsf{S}_1\}$:

1. Wait to receive $b^c \in \mathbb{Z}_{2^{k+s}}$ from the adversary, sample random $b^{c-1} \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, where $b \leftarrow b^c + b^{c-1} \mod 2^k \in \mathbb{Z}_2$.
2. Send b^j to S_j.
3. Run $\mathsf{MACGen}(b)$. Send m_b^j to S_j.

Figure 15: Random bit generation functionality $\mathcal{F}^{\mathsf{RanBitGen}}$

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

Initialize: Same as $\mathcal{F}^{\mathsf{InCom}}$.
Macro $\mathsf{MACGen}(\mathbf{x})$: Same as $\mathcal{F}^{\mathsf{InCom}}$.
InCom: Same as $\mathcal{F}^{\mathsf{InCom}}$.
Square Correlation Generation: Same as $\mathcal{F}^{\mathsf{SqGen}}$.
Random Bit Generation: Same as $\mathcal{F}^{\mathsf{RanBitGen}}$.
Bit Triple Generation: Same as $\mathcal{F}^{\mathsf{TripGen}}$.

Figure 16: Wrapper functionality $\mathcal{F}^{\mathsf{Wrap}}$

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

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 $(\mathsf{CRGen}, \mathsf{P}_i, \mathsf{sid})$ from $\mathsf{P}_i \in \{\mathsf{C}_i, \mathsf{S}_j\}$, compute $r \leftarrow \mathsf{PRF}_k(\mathsf{sid})$, send r to all P_i.

Figure 17: Correlated randomness functionality $\mathcal{F}^{\mathsf{CR}}$

Functionality $\mathcal{F}^{\mathsf{CR,glo}}$

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 $(\mathsf{CRGen}, \mathsf{P}_i, \mathsf{sid})$ from $\mathsf{P}_i \in \{\mathsf{C}_0, ..., \mathsf{C}_{n-1}, \mathsf{S}_j\}$, compute $r \leftarrow \mathsf{PRF}_k(\mathsf{sid})$, send r to all P_i.

Figure 18: "Global" correlated randomness functionality $\mathcal{F}^{\mathsf{CR,glo}}$

B.2 Single Check

See Fig. 20.

B.3 Protocol $\Pi^{\mathsf{RanBitGen}}$

See Fig. 21.

BatchCheck

Open: To open a value x_h:

1. S_j samples $r_h^j \in \mathbb{Z}_{2^s}^t$, then call $\mathcal{F}^{\mathsf{MAC}}$ to obtain $[r_h]^j$.
2. Servers then compute $[\tilde{x}_h] \leftarrow [x_h] + 2^k[r_h]$. 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 $\mathcal{F}^{\mathsf{Rand}}$, receive $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_{2^s}^t$.
2. Servers then compute $v \leftarrow \sum_{h=0}^{t-1} r_h \cdot \tilde{x}_h \mod 2^{k+s}$.
3. S_j computes $\tilde{m}^j \leftarrow \sum_{h=0}^{t-1} r_h \cdot m_h^j \mod 2^{k+s}$ and $z^j \leftarrow \tilde{m}^j - \alpha^j \cdot v \mod 2^{k+s}$.
4. S_j commits and opens z^j, then verifies if $z \leftarrow z^0 + z^1 \mod 2^{k+s}$ is zero. If the check passes, parties accept $x_h = \tilde{x} \mod 2^k$, otherwise they abort.

Figure 19: BatchCheck procedure [24]

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 $z^j \leftarrow m^j - \alpha^j \cdot \tilde{y}$.
3. S_j commits and opens z^j, then verifies if $z \leftarrow z^0 + z^1 \mod 2^{k+s}$ is zero. If the check passes, parties accept $y = \tilde{y} \mod 2^k$, otherwise they abort.

Figure 20: SingleCheck procedure [24]

B.4 Protocol Π^{MSB}

See Fig. 22.

C Security Proofs

For the following proofs of theorem 5.4 and theorem 5.5, we first consider the case in which a single C_i is involved in the protocol execution. We then extend the proof to the case in which multiple clients are involved.

C.1 Proof of Theorem 5.4

Let $\mathcal{A}$ be a malicious, static adversary that interacts with parties performing the protocol Π^{InCom} as shown in 7. 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 $\mathcal{F}^{\mathsf{InCom}}$ or with $\mathcal{S}$ in the ideal process for $\mathcal{F}^{\mathsf{InCom}}$.

Simulating the case when 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:

1. Emulate $\mathcal{F}^{\mathsf{CR,glo}}$, generate (α^0, α^1), send α^0 to $\mathcal{A}$ and $\mathcal{F}^{\mathsf{InCom}}$.

Protocol $\Pi^{\mathsf{RanBitGen}}$

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 $\mathcal{F}^{\mathsf{MAC}}$ with u^j as input, receives $[u]^j$.
3. Servers compute $[a] \leftarrow 2[u] + 1$.
4. Servers compute $[e] \leftarrow [a] \cdot [a]$.
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 $b^j \leftarrow \frac{d^j}{2}$ and $m_b^j \leftarrow \frac{m_d^j}{2}$.
8. Servers call $\mathcal{F}^{\mathsf{MAC}}$ to generate a random value $[r]$, where $r \in \mathbb{Z}_{2^s}$, servers compute $[b] \leftarrow [b] + 2^k \cdot [r]$.
9. S_j outputs $[b]^j \leftarrow (b^j, m_b^j)$.

Figure 21: Authenticated random bit generation $\Pi^{\mathsf{RanBitGen}}$. To ensure that the first s bits are random, we add the step 8 to the original protocol provided in [26].

2. Emulate $\mathcal{F}^{\mathsf{CR}}$, generate and send $\mathbf{x}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}^t$, $x_t^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$ $\mathbf{m}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}^t$ and $m_t^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$ to $\mathcal{A}$.
3. Emulate $\mathcal{F}^{\mathsf{Rand}}$, generate and send $\mathbf{r}$ to $\mathcal{A}$.
4. Act as an honest S_1, receive $\hat{v}^0$ and send c^1 to $\mathcal{A}$.
5. Compute v^0 and z^0 just as $\mathcal{A}$ will do. Set $z^1 \leftarrow (\hat{v}^0 - v^0) \cdot \alpha^1 - z^0 \mod 2^{k+2s}$.
6. Commit and send z^1 to $\mathcal{A}$, receive commitment z^0 from $\mathcal{A}$.
7. Check if $z^0 + z^1 = 0 \mod 2^{k+2s}$, abort as an honest S_1 if not.
8. Otherwise, send previously computed $(\mathbf{x}^0, \mathbf{m}^0)$ to $\mathcal{F}^{\mathsf{InCom}}$ and halt.

Proof. We now prove that $\mathsf{REAL}_{\Pi^{\mathsf{InCom}},\mathcal{A},\mathcal{Z}}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}}}$ is indistinguishable from $\mathsf{IDEAL}_{\mathcal{F}^{\mathsf{InCom}},\mathcal{S},\mathcal{Z}}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}}}$.

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 $v^1 \leftarrow \sum_{i=0}^{t-1} x_i^1 \cdot r_i + x_t^1 \mod 2^{k+s}$ is computed by S_1, while in the ideal execution v^1 is chosen uniformly at random by $\mathcal{S}$. Since x_t^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 (under 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 execution is identical. It remains to argue that the MAC shares output by all parties are identically distributed in both

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 $([a], [b_0], ..., [b_{k-1}])$ where $a, b_i \in \mathbb{Z}_2$.
2. Servers compute $[r] = \sum_{i=0}^{k-1} 2^i \cdot [b_i]$.

Protocol:

1. Servers run **Open** and **Batch check** to reconstruct $c \leftarrow [a] + [r]$.
2. Servers compute $c' \leftarrow c \mod 2^{k-1}$ and $[r'] = \sum_{i=0}^{k-2} 2^i \cdot [b_i]$.
3. Servers run Π^{A2B} with $([b_0], ..., [b_{k-2}])$ as input, receive $([b_0]_2, ..., [b_{k-2}]_2)$ as output.
4. Servers run Π^{BitLT} with $(c', [b_0]_2, ..., [b_{k-2}]_2)$ as input, receive $[p]_2$ as output.
5. Servers run Π^{B2A} with $[p]_2$ as input, receive $[p]$ as output.
6. Servers compute $[x'] \leftarrow c' - [r'] + 2^{k-1}[p]$ and $[d] \leftarrow [x] - [x']$.
7. Servers run **Open** and **Batch check** to reconstruct $e \leftarrow [d] + 2^{k-1}[a]$.
8. Let e_{k-1} be the most significant bit of e. Servers output $[s]_2 \leftarrow e_{k-1} + [a] - 2e_{k-1}[a]$.

Figure 22: Extract MSB protocol Π^{MSB} [26]. Within Π^{MSB}, the A2B protocol Π^{A2B}, the bitwise comparison protocol protocol Π^{BitLT} and B2A protocol Π^{B2A} can be found in [26].

executions. Note that since C_i is honest, the MACs are already correctly computed. The MAC shares of $\mathcal{A}$ is received from $\mathcal{F}^{\mathsf{CR}}$ in the real execution, which is simulated by $\mathcal{S}$ in the ideal execution. Thus, they are chosen uniformly at random in both worlds. The MAC shares of S_1 is correctly computed by an honest C_i in the real execution, where C_i first computes the correct MACs then abstracts the MACs by the shares received from $\mathcal{F}^{\mathsf{CR}}$. In the ideal execution, $\mathcal{A}$'s MAC shares are set by $\mathcal{F}^{\mathsf{InCom}}$ to the exact computed result. Then $\mathcal{F}^{\mathsf{InCom}}$ sets 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 when C_i is corrupted: $\mathcal{S}$ simulates a real execution in which the corrupted C_i controlled by $\mathcal{A}$ delivers message to uncorrupted S_0 and S_1 in the internal (simulated) interaction. The $\mathcal{S}$ works as follows:

1. Emulate $\mathcal{F}^{\mathsf{CR,glo}}$, generate (α^0, α^1), send them to $\mathcal{A}$ and $\mathcal{F}^{\mathsf{InCom}}$.
2. Emulate $\mathcal{F}^{\mathsf{CR}}$, generate $(\mathbf{x}^0, x_t^0, \mathbf{m}^0, m_t^0)$ and send them to $\mathcal{A}$.
3. Compute $\mathbf{m}$ and m_t as $\mathcal{A}$ will do.
4. Act as an honest S_1, receive $(\mathbf{x}^1, x_t^1, \mathbf{m}^1, m_t^1)$ from $\mathcal{A}$.
5. Perform *Consistency Check* just as honest S_0 and S_1 will do, abort if consistency check fails.
6. Otherwise, send $(\mathbf{x}^0, \mathbf{x}^1)$ and $\mathbf{m}^0$ to $\mathcal{F}^{\mathsf{InCom}}$, 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 5.1, we know that if the consistency check passes then parties output correctly generated MAC shares received from $\mathcal{A}$, except with negligible probability. Then, we notice that the shares output by S_0 in the real execution are the exact values received from $\mathcal{F}^{\mathsf{CR}}$, which is emulated by $\mathcal{S}$ in the ideal execution. Thus, they are distributed uniformly at random in both worlds. Then S_1 outputs the (correct) MAC shares received from $\mathcal{A}$ in both real and ideal worlds.

We conclude that the simulation is indistinguishable for $\mathcal{Z}$. □

We now discuss when multiple clients are involved. In both cases above, $\mathcal{S}$ has to repeat the simulation steps prior to the consistency checks until the last client completes its sharing phase. If S_0 is corrupted, then all shares received from the clients remain correct. Therefore, $\mathcal{A}$ gains no additional power except the ability to introduce an error during the consistency check. If a subset of clients are corrupted, $\mathcal{A}$ gains no additional power except the ability to share more $\mathbf{x}$. In both cases, Claim 5.1 still holds, and the probability of passing the consistency check in both executions is still identical, since the consistency check is always performed at the very end of the protocol execution. We thus conclude that the simulation is still indistinguishable for $\mathcal{Z}$.

C.2 Proof of Theorem 5.5

Let $\mathcal{A}$ be a malicious, static adversary that interacts with parties performing the protocol $\Pi^{\mathsf{InCom}}_{\mathsf{Diho}}$ as shown in 7. 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 $\mathcal{F}^{\mathsf{InCom}}$ or with $\mathcal{S}$ in the ideal process for $\mathcal{F}^{\mathsf{InCom}}$.

Simulating the case when 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:

1. Upon receiving α^0 from $\mathcal{A}$ as input to an instance of $\mathcal{F}^{\mathsf{vOLE}}$, send α^0 to $\mathcal{F}^{\mathsf{InCom}}$.
2. Emulate $\mathcal{F}^{\mathsf{CR}}$ instance, sample $\mathbf{x}^0 \xleftarrow{\$} \mathbb{Z}^t_{2^{k+s}}$, $x_t^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, send $(\mathbf{x}^0, x_t)$ to $\mathcal{A}$.
3. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive $\hat{\tilde{\mathbf{x}}}^0 \leftarrow (\hat{\mathbf{x}}^0, \hat{x}_t^0)$ and $\mathbf{b}^0$ from $\mathcal{A}$ as input to $\mathcal{F}^{\mathsf{vOLE}}$. Note that $\hat{\tilde{\mathbf{x}}}^0$ can be different from $\tilde{\mathbf{x}}^0$.
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 $\alpha^1 \xleftarrow{\$} S_1$, honestly compute $\mathbf{a}^0 \leftarrow \alpha^1 \cdot \hat{\tilde{\mathbf{x}}}^0 - \mathbf{b}^0$.
6. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive $\mathbf{a}^0$ from $\mathcal{A}$ as input to $\mathcal{F}^{\mathsf{vOLE}}$.

7. Sample $x_t^1 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, use zero-valued share inputs to set $\tilde{\mathbf{x}}^1 \leftarrow (\mathbf{0}, x_t^1)$. Set $\tilde{\mathbf{x}} \leftarrow \tilde{\mathbf{x}}^0 + \tilde{\mathbf{x}}^1$. Honestly compute $\mathbf{b}^1 \leftarrow \alpha^0 \cdot \tilde{\mathbf{x}} - \mathbf{a}^1$.
8. Emulate $\mathcal{F}^{\mathsf{Rand}}$, send $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_{2^s}^t$ to $\mathcal{A}$.
9. Act as an honest S_1, send v^1 to $\mathcal{A}$, receive back and reconstruct v.
10. Receive and open the commitment z^0 from $\mathcal{A}$. Honestly compute d^1 and open S_1's commitment $z^1 \leftarrow d^1 - c \cdot \alpha^1$.
11. Perform the consistency check. If check fails, abort and terminate.
12. If the check passes then define $\mathcal{A}$'s MAC shares using the received values for $\mathcal{F}^{\mathsf{vOLE}}$. Send $(\mathbf{x}^0, \mathbf{m}^0)$ to $\mathcal{F}^{\mathsf{InCom}}$ and halt.

Proof. We now prove that $\mathsf{REAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}}}_{\Pi^{\mathsf{InCom}}_{\mathsf{Diho}},\mathcal{A},\mathcal{Z}}$ is indistinguishable from $\mathsf{IDEAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}}}_{\mathcal{F}^{\mathsf{InCom}},\mathcal{S},\mathcal{Z}}$.

We first prove that the message distribution are identical in the real and idea execution. First, 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 $v^1 \leftarrow \sum_{i=0}^{t-1} x_i^1 \cdot r_i + x_t^1 \bmod 2^{k+s}$ is computed by S_1, while in the ideal execution v^1 is chosen uniformly at random by $\mathcal{S}$. Since x_t^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. Besides, 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 5.2 and 5.3, 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. $\mathcal{A}$'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 $\mathcal{F}^{\mathsf{vOLE}}$, which serves as random mask and are distributed uniformly at random for $\mathcal{Z}$. After $\mathcal{A}$ obtains its MAC shares, S_1's MAC shares are simply the correct MACs abstract $\mathcal{A}$'s MAC shares We thus conclude that the simulation is indistinguishable for $\mathcal{Z}$. □

Simulating the case when S_1 **is corrupted:** Similar to the case when S_0 is corrupted.

Simulating the case when C_i **is corrupted:** Similar to the case when S_0 is corrupted.

Simulating the case when S_1 **and** C_i **are corrupted:** $\mathcal{S}$ simulates a real execution in which the corrupted S_1 and C_i, controlled by $\mathcal{A}$, deliver messages to the uncorrupted S_0 in the internal (simulated) interaction. $\mathcal{S}$ works as follows:

1. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive α^1 and $\mathbf{a}^0$ from $\mathcal{A}$. Send α^1 to $\mathcal{F}^{\mathsf{InCom}}$.
2. Sample $\mathbf{x}^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}^t$, $x_t^0 \xleftarrow{\$} \mathbb{Z}_{2^{k+s}}$, set $\tilde{\mathbf{x}}^0 \leftarrow (\mathbf{x}^0, x_t^0)$. Honestly compute $\mathbf{b}^0 \leftarrow \alpha^1 \cdot \tilde{\mathbf{x}}^0 - \mathbf{a}^0$.
3. Emulate $\mathcal{F}^{\mathsf{CR}}$ instance, send previously sampled $(\mathbf{x}^0, x_t^0)$ to $\mathcal{A}$.
4. Emulate $\mathcal{F}^{\mathsf{vOLE}}$, receive $\tilde{\mathbf{x}} \leftarrow (\mathbf{x}, x_t)$ and $\mathbf{b}^1$ 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 $\mathcal{F}^{\mathsf{InCom}}$. If $\mathcal{F}^{\mathsf{InCom}}$ aborts, then abort; otherwise, store the set $S_0 = S_0 \cap S$ (where initially $S_0 = \mathbb{Z}_{2^s}$).
6. Sample $\alpha^1 \xleftarrow{\$} S_0$. Honestly compute $\mathbf{a}^1 \leftarrow \alpha^0 \cdot \tilde{\mathbf{x}} - \mathbf{b}^1$.
7. Emulate $\mathcal{F}^{\mathsf{Rand}}$, send $\mathbf{r} \xleftarrow{\$} \mathbb{Z}_{2^s}^t$ to $\mathcal{A}$.
8. Act as an honest S_0, send v^0 to $\mathcal{A}$, receive back and reconstruct v.
9. Receive and open the commitment z^1 from $\mathcal{A}$. Honestly compute d^0 and open S_0's commitment $z^0 \leftarrow d^0 - v \cdot \alpha^0$.
10. Perform the consistency check. If check fails, abort and terminate.
11. If the check passes then define $\mathcal{A}$'s MAC shares using the received values for $\mathcal{F}^{\mathsf{vOLE}}$. Send $(\mathbf{x}^1, \mathbf{m}^1)$ to $\mathcal{F}^{\mathsf{InCom}}$ and halt.

Proof. We now prove that $\mathsf{REAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}}}_{\Pi^{\mathsf{InCom}}_{\mathsf{Diho}},\mathcal{A},\mathcal{Z}}$ is indistinguishable from $\mathsf{IDEAL}^{\mathcal{F}^{\mathsf{CR}},\mathcal{F}^{\mathsf{Rand}}}_{\mathcal{F}^{\mathsf{InCom}},\mathcal{S},\mathcal{Z}}$.

Again, we first prove that the message distribution are identical in both executions. Compared to the case when S_0 is corrupted, the only difference here is that $\mathcal{A}$ is able to send a guess query while executing the second $\mathcal{F}^{\mathsf{vOLE}}$ and the rest of the communications are identical. We conclude that the messages simulated by $\mathcal{S}$ are indistinguishable from those in the real execution. Since now $\tilde{\mathbf{x}}^1$ can be a vector of any arbitrary values, $\mathcal{A}$ can not introduce any error by sending $\tilde{\mathbf{x}}$ to $\mathcal{S}$. While executing the first $\mathcal{F}^{\mathsf{vOLE}}$, $\mathcal{A}$ can still introduce an error by sending an incorrect α^1. However, this does not affect the probability of passing the consistency check in both worlds, since the errors introduced to the ideal execution are the same as those in the real execution.

We still need to prove that the distribution of the MAC shares is identical in both worlds. Again, we know from Claims 5.2 and 5.3 that the MAC shares are correctly computed in the real protocol execution as the MAC shares output by $\mathcal{F}^{\mathsf{InCom}}$ if the consistency check passes, except with negligible probability. While the $\mathcal{A}$'s MAC shares are set by $\mathcal{S}$, S_0's MAC shares are the sum of outputs received from $\mathcal{F}^{\mathsf{vOLE}}$ in the real world, which are the correct MAC shares and thus distributed identically in the ideal world. We thus conclude that the simulation is indistinguishable for $\mathcal{Z}$. □

Simulating the case when S_0 **and** C_i **are corrupted:** Similar to the case when S_1 and C_i are corrupted.

We now analyze the case when multiple clients are involved. The main effect is that $\mathcal{A}$ may now be able to send multiple guess queries to $\mathcal{F}^{\mathsf{vOLE}}$ instances. We assume that $\mathcal{A}$ has corrupted q clients along with S_1. During the simulation, $\mathcal{S}$ always forwards guess queries to $\mathcal{F}^{\mathsf{InCom}}$. As long as $\mathcal{F}^{\mathsf{InCom}}$ aborts, $\mathcal{S}$ simulates the corresponding $\mathcal{F}^{\mathsf{vOLE}}$ instance aborting. If all guesses succeed, $\mathcal{S}$ draws $\alpha \in \bigcap_{i=0}^{q-1} S_{0,i}$. The probability of passing the consistency check in both words remains identical. We thus conclude that the simulation is still indistinguishable for $\mathcal{Z}$.

C.3 Proof of Theorem 5.6

Proof Sketch: We construct an adversary $\mathcal{S}$ for the ideal execution such that the environment machine $\mathcal{Z}$ cannot distinguish between the real execution of protocol Π^{SqGen} with $\mathcal{A}$ ($\mathbf{REAL}_{\Pi^{\mathsf{SqGen}},\mathcal{A},\mathcal{Z}}$)

and the ideal execution of $\mathcal{F}^{\mathsf{SqGen}}$ with $\mathcal{S}$ ($\mathbf{IDEAL}_{\mathcal{F}^{\mathsf{SqGen}},\mathcal{S},\mathcal{Z}}$). We observe that the simulator only needs to simulate the **Open** and **MAC check** procedures to reconstruct $\mathbf{e}$. It is easy to see that the distribution of $\mathbf{e}$ is identical in the ideal and real executions, since both $\mathbf{a}$ and $\mathbf{b}$ are just random vectors chosen by $\mathcal{F}^{\mathsf{TripGen}}$ in the real execution and emulated by $\mathcal{S}$ in the ideal execution. Following the proof of [24], we also conclude that the probability of passing the check is the same in both executions. It remains to show that the square correlation shares output by the servers are identically distributed in both executions. Due to Equation 4.2, we know that the square shares are correctly computed if $\mathbf{e}$ is correctly reconstructed, except with negligible probability. $\mathcal{S}$ will set the exact computed result of $\mathcal{A}$ as its output from $\mathcal{F}^{\mathsf{SqGen}}$, then the honest party's output will be chosen by $\mathcal{F}^{\mathsf{SqGen}}$ as a random mask to the correct square correlation, which distributed identically in both executions. We thus conclude that ideal and real executions are indistinguishable to the environment machine $\mathcal{Z}$.