ELSA: Secure Aggregation for Federated Learning with Malicious Actors

Mayank Rathee[1], Conghao Shen[1,2], Sameer Wagh[1,3], and Raluca Ada Popa[1]
University of California, Berkeley[1]
Stanford University[2]
Devron Corporation[3]

***Abstract*—Federated learning (FL) is an increasingly popular approach for machine learning (ML) in cases where the training dataset is highly distributed. Clients perform local training on their datasets and the updates are then aggregated into the global model. Existing protocols for aggregation are either inefficient, or don't consider the case of malicious actors in the system. This is a major barrier in making FL an ideal solution for privacy-sensitive ML applications. We present ELSA, a secure aggregation protocol for FL, which breaks this barrier - it is efficient and addresses the existence of malicious actors at the core of its design. Similar to prior work on Prio and Prio+, ELSA provides a novel secure aggregation protocol built out of distributed trust across two servers that keeps individual client updates private as long as one server is honest, defends against malicious clients and is efficient end-to-end. Compared to prior works, the distinguishing theme in ELSA is that instead of the servers generating cryptographic correlations interactively, the clients act as *untrusted* dealers of these correlations without compromising the protocol's security. This leads to a much faster protocol while also achieving stronger security at that efficiency compared to prior work. We introduce new techniques that retain privacy even when a server is malicious at a small added cost of 7-25% in runtime with negligible increase in communication over the case of semi-honest server. Our work improves end-to-end runtime over prior work with similar security guarantees by big margins - single-aggregator RoFL by up to 305x (for the models we consider), and distributed trust Prio by up to 8x.**

1. Introduction

Federated learning (FL) [9, 83] is an emerging approach in privacy-conscious machine learning (ML) that enables efficient training over large and highly distributed datasets. In a typical workflow, the application developer runs a server to maintain a global ML model; the server iteratively updates this model by computing an aggregate of the gradient updates sent by the clients after local training [9].

In a utopian world with no bad actors, FL allows the clients to retain full ownership of their data and the application developer to efficiently train high-quality ML models. However, both the clients and the servers can be corrupted by adversaries leading to a range of attacks [24, 96, 97, 99] against the system. Recently, governments have been incentivizing research in building robust and private FL solutions for tasks like financial crime prevention [11], and pandemic response and forecast [12] with prizes worth $800,000. Over the years, a rich line of work [19, 21, 22, 26, 35, 37, 40, 57, 62, 66, 70, 82, 84, 102, 104, 107, 114] has identified two security properties as desirable:

- *Privacy of individual gradients.* It is well-known that the gradients submitted by the clients leak information about their local datasets [19, 24, 26, 34, 83, 105]. Therefore, to be usable in privacy-sensitive applications, it is imperative that FL preserves the privacy of *individual* gradients (and thereby of the datasets)[1].
- *Filtering out boosted gradients from malicious clients.* Aggregation, which is at the heart of FL, is quite sensitive to out-of-proportion values. Without any defenses in place, even a *single* malformed gradient (e.g., with a very high norm) can arbitrarily bias the global model. Malicious clients can boost their gradients (scale up to a large norm) to corrupt the global model, and many model poisoning attacks [17, 53, 98, 99] rely on this strategy. An effective FL solution must defend against boosted gradients. Recent prior work [99, 104] has shown that filtering gradients based on their ℓ_2 norm (a.k.a. ℓ_2 defense) is effective against a large number of sophisticated poisoning attacks under realistic threat models for production FL[2].

Most existing research either provides privacy [19, 21, 26, 35, 57, 62, 66, 70, 82, 102, 107] or defends against malformed gradients [22, 37, 84, 104, 114]. Some attempts have since been made to achieve both the properties, but they are either quite inefficient [34, 40, 41, 101] or resort to weak threat models [13, 61, 64, 88]. These protocols can be divided based on whether they operate in the single-aggregator model where a central server facilitates gradient aggregation (RoFL [34], EIFFeL [40], [88, 101]), or if they distribute the trust of the central aggregator into two servers hosted behind separate trust domains (Prio [41], Prio+ [13], [61, 64]).

We present ELSA (**E**fficient **L**earning with **S**ecure **A**ggregation), a secure aggregation protocol for FL which uses distributed trust and guarantees (Table 1):

1. The global update computed after aggregation of the submitted gradient updates has to be revealed, but individual updates should stay hidden.
2. No defenses are powerful enough to completely stop all poisoning attacks, and all existing defenses [99] rely on some heuristics.

Federated Learning Protocols	Efficient Parties	Malicious Privacy	Poisoning Resilience	Trust Model
FedAvg [83]	✓	✗	✗	○
SecAgg [19, 26]	✓	✓	✗	○
Defence [37, 104, 114]	✓	✗	✓	○
RoFL*[34], [40]	✗	✓	✓	○
Prio [41]	✗	✓	✓	○ ●
Prio+ [13], [61, 64]	✓	✗	✓	○ ●
ELSA (This work)	✓	✓	✓	○ ●

* It appears that RoFL should provide malicious privacy (not considered in their paper); we give them the advantage here.

TABLE 1: Qualitative comparison of FL protocols. Only representative works are shown. Single-aggregator model is represented by ○, and ○ ● refers to distributed trust with two servers behind separate trust domains. Poisoning resilience refers to some defense against malformed gradients.

- *Malicious privacy*. Honest clients' gradients stay private even in the face of a strong collusion between malicious clients and at most one malicious server. As long as one server is honest, privacy is guaranteed.
- *Resilient to boosted gradients*. We employ the commonly used ℓ_2 defense [1, 104] to filter out boosted gradients. Alongside its efficacy [99], the relative simplicity of this defense makes it ideal for privacy-preserving systems. In this defense, gradient updates with ℓ_2 norms much larger than usual are discarded.
- *Efficiency through lightweight protocols*. Our server and client-side protocols are lightweight, and free of expensive cryptographic and public-key operations. This translates to a simpler implementation and highly efficient end-to-end secure FL. In terms of total runtime, we outperform the single-aggregator and the distributed trust state-of-the-art by 146-305x and 6-8x (with up to 16x improvement in servers), respectively, while also requiring lower communication from the clients.

In addition, ELSA has a number of other desirable properties. Unlike RoFL [34], EIFFeL [40] and Prio [41] who can't support bandwidth-constrained clients, ELSA stays efficient even when a subset of clients have strict bandwidth constraints. In ELSA, a few malicious clients cannot block output delivery (RoFL lacks this property), and it can withstand client dropouts without runtime degradation (unlike [19, 26, 70, 102]). ELSA can use an input-independent offline phase to significantly boost end-to-end runtimes for when the gradients become available. Unlike many single-aggregator prior works [19, 26, 40, 70, 102], in our protocol, clients don't need to talk to each other; this makes communication for clients much simper. Moreover, the communication from client to server is one-shot (single message) after which clients don't need to be online.

Other applications. Our techniques are more generally applicable to realizing other defences in FL as long as they operate independently on each client's submission (e.g. ℓ_∞ defense [34]), and also achieving malicious privacy in applications like privacy-preserving telemetry [13, 41].

1.1. Technical Overview

In the single-aggregator setting, RoFL [34] simultaneously achieves malicious privacy and enforces norm-based defenses (ℓ_2 and ℓ_∞) by using expensive zero-knowledge proofs (Bulletproofs [33]) which makes them quite inefficient. We don't see a way to circumvent this limitation in their setting, and therefore, focus on distributed trust instead.

Prio+ overview. We start with the design of Prio+ [13] which operates in the distributed trust setting with two servers. Their design yields an efficient protocol which (heuristically) weeds out malformed updates by a "relaxed" ℓ_∞ defense, but doesn't guarantee privacy in the presence of a malicious server. The high-level idea behind their construction is to have each client send Boolean secret shares of its gradient update to the servers, and the servers use the bit-length of the shares as a proxy to enforce a weaker form of ℓ_∞ defense. For example, restricting the magnitude of values in the update to be at most 7 can be enforced by allowing bit-lengths of 3. This prevents malicious clients from sending boosted updates [104] which have large magnitudes to overpower honest updates and poison the model. Servers then engage in interactive 2-party computation (2PC) using oblivious transfer (OT) [68, 86] to convert Boolean shares to arithmetic, so they can be aggregated.

Malicious privacy challenge in Prio+. If one of the two servers in Prio+ is malicious, privacy of individual gradients can't be preserved. An example malicious strategy would be that errors introduced in OT messages propagate as a function of the secret. A direct use of techniques from prior works on malicious-secure Boolean to arithmetic conversion [44, 51, 93] might seem promising[3], but such protocols are more than an order of magnitude more expensive in communication than their semi-honest counterparts.

New insights for almost *free* malicious privacy. We uncover new insights into this problem and build a solution which provides malicious privacy, essentially for *free*. Our idea relies on two important observations. First, to ensure the privacy of gradients, we only need to safeguard the steps that servers execute on each client's input shares excluding the final aggregation step where gradients of all clients are added together; let's call them "client-specific steps". This is because the aggregation step only admits additive errors which don't depend on individual honest gradients [39], and therefore, don't affect privacy. Second, for a client c, knowing the servers' internal state beforehand for c's client-specific steps neither gives it any advantage in successfully mounting a poisoning attack, nor reveals any information about other clients' gradients. Therefore, each client can share random tapes with the servers which, along with the clients input shares, makes the messages exchanged between the servers totally deterministic. This enables clients to locally generate a digest of the transcript of server interaction, and send it to the servers helping the honest server catch any

3. Each client generates its own MAC key and uses that to send authenticated Boolean shares of its gradient. Servers use separate instances of malicious-secure 2PC to process shares of each client.

malicious behavior that could violate privacy. We formally define malicious privacy in Definition 1.

Extending to ℓ_2 defense. Stemming from Prio+, the current design only supports the relaxed ℓ_∞ defense. A more commonly used and better studied [1, 34, 99, 104] defense enforces an upper bound on the ℓ_2 value of gradient updates. This can be supported by using Beaver triples [18] to compute the sum of squares of values within each gradient vector. However, as a consequence of operating over finite rings, the ℓ_2 defense needs to be paired with a component-wise upper bound within each gradient vector to maintain soundness [34]; there should be no overflows in the ℓ_2 computation. We refer to this *augmented* ℓ_2 defense as $\ell_2^{\square}$. To address this, servers first convert Boolean shares to arithmetic (ensures component-wise upper bound similar to relaxed ℓ_∞ in Prio+), and then use beaver triples to compute shares of ℓ_2.

Tapping into cheaper *untrusted* randomness. To make our end-to-end protocol more efficient, we realize that clients can serve as cheap untrusted sources of the cryptographic material (correlations) needed by the servers. This material, which is interactively generated by the servers by engaging in expensive 2PC, can be *locally* generated by the client. The client just samples some random numbers subject to simple constraints. Once generated, these correlations can be secret shared between the servers to provide them with a significant boost. However, malicious clients can't be trusted with generating correct correlations, so the servers need an efficient way to verify these correlations. In our protocol, servers need OT correlations for Boolean to arithmetic conversions and Beaver triples for computing shares of ℓ_2. We use ideas from Keller et al. [72] to efficiently verify *all* OT correlations while just communicating two field elements, and for Beaver triples, we use the SPDZ sacrifice technique [43, 45, 73, 74]. This approach greatly reduces the total communication and the entire workload of servers, while also *easing* the computational effort of clients (transcript emulation becomes simpler). The only downside is the increase in communication from clients, and therefore, we let heavily bandwidth-constrained clients to individually opt out of this optimization, and still largely retain our end-to-end efficiency. Note that all the clients (including regular ones) in our protocol communicate less than the three prior works (RoFL, EIFFeL and Prio) which provide similar security guarantees, and none of these prior works support bandwidth-constrained clients.

Achieving one-shot clients. By using clients as untrusted correlation sources, there is a part of the random tape of the servers which can't be known apriori to the clients. One can think of this as corresponding to the "random challenge" which needs to be hidden from the client until it sends its correlations to the servers (for soundness of the defense against malformed gradients). This leaves us with a two-round client because the transcript for a part of the server-server interaction can't be generated by the client until it submits the correlations and receives the random challenge. Casting our protocol as a public-coin interactive protocol and using the distributed variant of the Fiat-Shamir transform [27, 54], we achieve single-round clients.

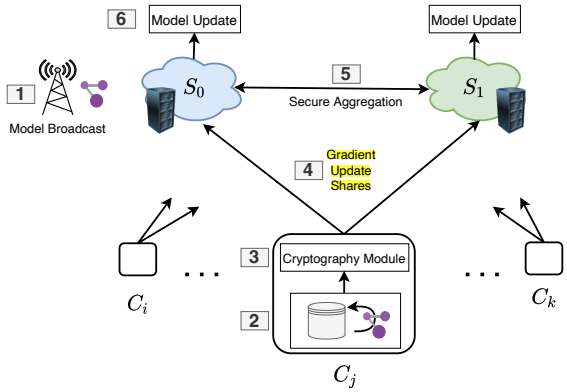


Figure 1: ELSA FL pipeline. S_0, S_1 denotes servers in different trust domains who each hold a copy of the current global model. C_i represents the ith client who performs local training before using its cryptography module to enable secure aggregation. Numbers in boxes represent steps.

2. Preliminaries

ELSA's design considers two types of actors - servers and clients. Clients, who can be users running mobile devices (cross-device) or large organizations (cross-silo), have personalized datasets which they don't want to reveal to anyone else, and the servers wish to train an ML model on this large distributed dataset. Typical to FL, this is achieved by choosing a subset of clients during each training iteration, and having them contribute to updating the global model by training it on their local datasets and sending the updates back to the servers. Our design considers the distributed trust setup where two servers deployed in separate trust domains collectively and securely emulate the task of the central coordinating server [9, 26], i.e., collecting all updates and integrating them into the global model.

We present our pipeline in Fig. 1. Each training iteration starts with S_0 and S_1 selecting a common subset of clients (round participants) and broadcasting the current global model to each of them (each server knows the global model in clear). These clients then use their datasets to train the global model for a few epochs locally, and then use the cryptography module to *securely* share the generated gradient update between the two servers such that each server individually receives some "random-looking" update. Servers then engage in an interactive protocol between themselves to aggregate the updates and finally, integrate them into the global model.

2.1. Problem Setup

In this work, we primarily focus on constructing the cryptography module that is run on each client, and the secure aggregation protocol that servers engage in. We assume that clients generate gradient updates using an ML black-box, and consider its specifics tangential to our work.

The computation of interest in secure FL is the following: we have a set of clients $\mathcal{C}$ chosen for the current iteration, where client i holds a gradient vector x_i of size m (a.k.a parameter size or #params), and the goal is to compute the average of these gradient vectors[4] [25, 71, 83], i.e., $\nabla = \frac{1}{|\mathcal{C}|}\sum_{i=1}^{|\mathcal{C}|} x_i$. We simply refer to this computation as *aggregation*, ignoring division by $|\mathcal{C}|$ as this is a public constant in our system.

Sensitive gradients. It has long been established [19, 24, 26, 34, 83, 105] that gradients often leak sensitive information about clients' local datasets, and therefore, there is a need to keep them individually private. To achieve this goal in ELSA, we use techniques from secure multi-party computation [13, 48] to ensure that all computation done by the servers happens in a privacy-preserving fashion, such that only the final aggregated update is learned by the servers.

Utility of the service. The utility of an FL service relies on the quality of the gradients x_i. A single ill-formed gradient, if left unchecked, can totally alter ∇. Malicious clients can boost their gradients to bias the training process [26, 34, 79, 104, 109]. Therefore, to limit the scope of such attacks, it is imperative to check for ill-formed gradient updates. An effective strategy that is used by prior FL protocols [1, 13, 34, 41, 99, 104] is for the servers to enforce a norm bound on the gradients before accepting them. We consider the problem of selecting an appropriate bound orthogonal to our work and refer the reader to prior work [34]. Additionally, we make the common assumption that the chosen bound is publicly known to everyone.

2.2. Security Guarantees and Threat Model

We consider a strong threat model where corrupted clients are assumed to be malicious and up to one server is corrupt. We distill corruption of server into two cases (with different guarantees): the corrupt server is semi-honest or malicious. Malicious parties can arbitrarily deviate from the protocol specification in an attempt to break its guarantees, while semi-honest parties follow the protocol specification, but can try to infer as much private information as possible from the transcript. Like prior works on distributed trust [13, 41, 42, 46, 47, 108], we assume that both servers aren't simultaneously compromised, and therefore, at most one of them is corrupt. Informally, ELSA guarantees:

- *For semi-honest server (with malicious clients).* Individual gradients of honest clients maintain privacy (not revealed to anyone other than the source client itself), and the computation done by the servers, i.e., $\ell_2^{\square}$ checks and aggregation, is correct (thus both privacy and correctness are maintained).
- *For malicious server (with malicious clients).* Honest clients' gradients enjoy privacy, but the computation at servers can be incorrect (thus only privacy is guaranteed). By incorrect computation, we mean no guarantees for both - the final aggregate and the $\ell_2^{\square}$ defense. Since the malicious server can censor all-but-one honest client in an attempt to recover its private gradient (from the final aggregate), we maintain a threshold parameter τ which denotes the minimum number of gradients that need to be aggregated before the final result is revealed; τ can be appropriately set from an estimated upper bound (can be adjusted for confidence) on the relative prevalence of malicious clients to thwart this privacy attack (w.h.p.). The other honest server ensures τ is met and aborts otherwise.

We formally prove these properties in theorems A.1 and B.1. Similar to prior work [19, 26, 34, 70, 102] on secure FL, we consider it a meaningful use of our system when there are at least two honest clients so that the output of aggregation doesn't trivially reveal the honest gradient.

Setting up distributed trust. There has been an explosion in recent privacy-preserving systems which are based on distributed trust. Apart from the numerous academic works [13, 28, 41, 42, 46, 47, 52, 60, 89, 108], it has been adopted in many real-world deployments including Firefox telemetry [2], COVID-19 exposure notification analytics [8], oblivious DNS [3], and cryptocurrency wallets [4, 5]. In ELSA, the application developer is responsible for setting up distributed trust[5], and we refer the reader to prior work (e.g., [2, 8]) on how to do this properly. Given our security guarantees, this ensures that honest clients no longer have to put their entire trust in one server for privacy; as long as the other server is working fine, privacy is guaranteed.

2.3. Building Blocks

Notation. We denote the servers (a.k.a. aggregators) by S_b for $b \in \{0,1\}$ and the set of all participating clients by $\mathcal{C}$. $[n]$ denotes the set $\{0, 1, \ldots, n-1\}$. The set (or vector) $\{x_0, x_1, \ldots, x_{n-1}\}$ is denoted by $\{x_i\}_{i=0}^{n-1}$. When it is clear from the context, we will drop n from $\{x_i\}_{i=0}^{n-1}$ and represent it as $\{x_i\}_i$ for brevity. We use $\oplus, \wedge$ for bitwise XOR and AND, respectively. Our protocols support aggregations over $\mathbb{Z}_L$, where $L = 2^\ell$. $x \xleftarrow{\$} \mathbb{Z}_L$ denotes that x is uniformly randomly sampled from $\mathbb{Z}_L$. For a vector y, we denote its entries by y_i for $i \in [|y|]$ and all operations performed on vectors are component-wise. We use $\leftarrow$ to set a variable. $||$ denotes string concatenation. λ, κ are the computational and statistical security parameters, respectively.

Vector norms and defenses. The Euclidean norm, or ℓ_2 norm, of a vector $(x_1, \ldots, x_n)$ is defined as $\sqrt{x_1^2 + \cdots + x_n^2}$, and forms the main defence in our work against boosted gradients. When working with cryptographic primitives over finite rings, an upper bound on the ℓ_2 norm of a vector is ineffective in containing the magnitudes of individual components because overflowed values wrap around the modulus (as observed in RoFL [34]). Therefore, we augment the ℓ_2 bound with a component-wise upper

4. For the ease of exposition, we consider all clients are assigned equal weight. Our techniques straightforwardly extend to the general case.

5. Compared to works like RoFL in the single-aggregator model which base their security on the hardness of solving discrete log, distributed trust (theoretically) is a somewhat weaker notion of security, but leads to much more practical solutions.

bound (using bitlength) which works similarly to a relaxed ℓ_∞ bound; we refer to this additional bound as $\ell_\sqcap$ and the combined defense as $\ell_2^\sqcap$. ℓ_∞ norm of a vector $(x_1, \ldots, x_n)$ is defined as $\max_i |x_i|$, and an $\ell_\sqcap$ bound allows this value to be at most $2^w - 1$ for some $w \in \mathbb{N}$. For our case, ℓ_2 is faithful as long as $\ell_\sqcap$ is set such that $n \cdot (\ell_\sqcap)^2 \leqslant L'$ (implying no overflows), where L' is the ring modulus under which ℓ_2 computation happens [34]. In our $\ell_2^\sqcap$ defense, we typically choose a large $\ell_\sqcap$ (L' can be appropriately adjusted) to leave enough slack for honest gradients; setting it to be close to the ℓ_2 bound (say μ) is ideal[6] because the definition of ℓ_2 norm implicitly restricts the magnitude of any individual value in the vector to be at most μ. Moreover, rather than bounding ℓ_2, we bound ${\ell_2}^2$ (with the bound value denoted by μ^2), but for notational simplicity refer to it as ℓ_2. Note that imposing an $\ell_\sqcap$ bound already implies an ℓ_2 bound of $n \cdot (\ell_\sqcap)^2$, but that is quite coarse-grained and therefore, an explicit bound (μ^2) on the ℓ_2 norms is required.

Arithmetic/Boolean secret sharing. In arithmetic secret sharing [95], a value $x \in \mathbb{Z}_L$ is split into a pair of shares $x^{(0)}, x^{(1)}$ such that $x^{(0)} + x^{(1)} = x \pmod L$. In our setting, typically, the client who owns the value x creates its shares as: $x^{(0)} \xleftarrow{\$} \mathbb{Z}_L$ and $x^{(1)} \equiv x - x^{(0)} \pmod L$, and sends $x^{(0)}$ to S_0 and $x^{(1)}$ to S_1. Given shares of two secret values x and y, shares of $z \equiv x + y \pmod L$ can be locally computed by each server S_b setting $z^{(b)} \leftarrow x^{(b)} + y^{(b)} \pmod L$. When $L = 2$, we get Boolean secret sharing.

Oblivious Transfer (OT). An ℓ-bit "t-choose-1" OT computation [15, 68, 75], denoted by $\binom{t}{1}$-OT$_\ell$, allows the receiver (OTRc) holding a "choice" input $j \in [t]$ to receive the message m_j from a set of ℓ-bit messages $\{m_i\}_{i=0}^{t-1}$ held by the sender (OTSn). The sender learns nothing during the protocol and receiver learns no message other than m_j. For this work, the case of $t = 2$ is the most relevant. Correlated OT (COT) [16, 91] is a variant of OT where the sender has an input m, receiver has a choice bit j, and the protocol outputs $(r, r + m)$ to the sender and $(r + j \cdot m)$ to the receiver, where $r \xleftarrow{\$} \{0,1\}^\ell$. OT extension [30, 68] protocols generate polynomially many OTs given λ base OTs.

Beaver triples. Given secret shares of values $x, y \in \mathbb{Z}_L$, computing the shares of $x \cdot y$ requires interaction. A commonly used approach for this secure multiplication is using a Beaver triple [18] which consists of three elements (α, β, γ) such that $\gamma \leftarrow \alpha \cdot \beta$, and $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_L$. Secret shares of a beaver triple can be used to compute shares of $z \leftarrow x \cdot y$ with each party only communicating 2 ring elements [18, 48, 91].

3. ELSA: Secure Federated Learning

In this section, we describe our protocol for secure FL. We begin by constructing a simple protocol that enforces ℓ_2 bounds, but is neither efficient nor guarantees malicious privacy. We then show how to address each of these issues sequentially, and finally arrive at ELSA.

6. Larger than this would waste protocol resources.

3.1. Norm Bounding with Semi-Honest Privacy

In Prio+ [13], clients use Boolean secret sharing to share their gradient updates between the two servers, and the servers then engage in an interactive protocol to convert Boolean shares to arithmetic shares which can then be locally aggregated. This allows the servers to efficiently enforce an $\ell_\sqcap$ bound on client submissions. In this work, we follow a similar strategy. Let's consider the aggregation task at hand is to compute sum over the ring $\mathbb{Z}_L$ and we want to ensure that each individual value is at most $2^w - 1$ ($= \ell_\sqcap$) in magnitude, for $w \leqslant \ell$. Each client locally bit decomposes the values in its gradient vector and sends Boolean secret shares to the servers. The servers reject all client submissions which have more than w bit-shares per component of the gradient update, thereby retaining only the shares of values which are bounded by 2^w (refer to Appendix C for a discussion on negative values). These bit-shares are then converted back to arithmetic shares of the original values by using an COT-based bit composition protocol [13, 48, 91] (opposite of bit decomposition), and finally aggregated locally by the servers to get shares of the final aggregate. This result is *opened* by the servers through reconstruction (exchanging arithmetic share with each other) to yield the output.

As the first step towards building ELSA, we focus on ℓ_2 norm bounding [1, 104] to filter out boosted gradients from malicious clients. RoFL [34], the state-of-the-art single-aggregator protocol for secure FL, also considers ℓ_2 bounding as one of their prime defenses. The relative simplicity of this defence compared to others [22, 84, 114] makes it suitable when working over secret shares. To realize this defense over arithmetic secret-shared gradients (post bit composition), two secure computations need to be performed. First, arithmetic secret shares of the ℓ_2 value needs to be computed, and second, this value needs to be securely compared against the upper bound such that the only information that is revealed is the comparison output, i.e., whether the ℓ_2 value is within bound or not. We refer to the first step as "ℓ_2 computation", and the second one as "ℓ_2 enforcement". We now describe how both steps can be performed, and later in this section, optimize this solution. Note that when working over finite rings, ℓ_2 defense needs to be paired with a per-component bound, like $\ell_\sqcap$, to ensure soundness (see Section 2.3). In our case, the two steps of ℓ_2 defense are only performed after the bit composition step, ensuring that each value[7] is individually bounded by 2^w.

ℓ_2 computation. To compute shares of the ℓ_2 value, all we need is to perform secure multiplication of arithmetic shares of each component in the gradient update with itself, and finally add the resulting shares locally. This can be achieved using Beaver triples; one triple for each value in the vector. The ring over which aggregation happens ($\mathbb{Z}_L$) need not be the same as the one over which ℓ_2 is computed. The ℓ_2 ring, $\mathbb{Z}_{2^u}$, can be larger than $\mathbb{Z}_L$. Since shares of a value

7. We consider the same bound on each value for simplicity of exposition. Our protocol straightforwardly extends to the general case.

less than L over the ring $\mathbb{Z}_{2^u}$ can be converted to shares over $\mathbb{Z}_L$ by local modulo reduction with L [90], servers can perform bit composition to output shares over the larger $\mathbb{Z}_{2^u}$, perform ℓ_2 computation and checks, and then during aggregation convert them back to $\mathbb{Z}_L$. Shares of the required Beaver triples can be generated between the two servers using either COT (requires $2u$ instances of $\binom{2}{1}$-COT_u per triple) or homomorphic encryption [48].

ℓ_2 enforcement. Once the ℓ_2 computation is done, the next step is to check whether the value obeys the upper bound μ^2. Opening the ℓ_2 value directly, and locally comparing with μ^2 would leak extra information than needed, i.e., the precise ℓ_2 value. Therefore, this comparison needs to happen using a secure sub-protocol which only reveals whether the bound is violated or not. In this work, we perform this step by having the servers first locally compute arithmetic shares of $z \leftarrow y - \mu^2 \bmod 2^u$, where y is the ${\ell_2}^2$ value, and then extract its most-significant bit (MSB) by evaluating a secure adder [48]; secure adders can be computed using COTs [48, 91] with each AND gate requiring two $\binom{2}{1}$-COT_1. The adder outputs Boolean shares of the sum, and the MSB is then opened to reveal the output; zero implies bound violation.

3.2. Designing an Efficient Protocol

To achieve an efficient end-to-end protocol, we propose a novel redistribution of secure computation across parties.

3.2.1. Cheaper Sources of Correlations

We now analyze the current construction, identify the bottleneck, and propose a technique that *completely dissolves* this bottleneck. For this analysis, we focus on OT-based primitives instead of homomorphic encryption (HE), and use the IKNP OT extension (with its derivatives) [16, 68]. Recent PCG-based OT extension protocols [30, 112] and HE trade-off lower communication for much higher computation compared to IKNP which makes them trickier to theoretically analyze. Moreover, as long as the servers have a good bandwidth connection with each other, these solutions aren't too different in end-to-end performance [92, 112], and therefore, IKNP suffices for our analysis.

Bit composition and ℓ_2 computation are the only two phases where the communication between the servers grows linearly with the size of the gradient updates and the number of participating clients; all the other phases require only a small fraction of this communication. Of these two, ℓ_2 computation is more heavy because generating Beaver triples is quite expensive. In particular, the communication cost of ℓ_2 computation is dominated by $m \cdot 2u \cdot (\lambda + u)$, while bit composition incurs $m \cdot w \cdot (\lambda + u)$ bits. Since u is at least $2w + \log m$, there is $>$ 4x difference in communication. This difference is even more pronounced when a commonly used technique called probabilistic quantization [76] is used to reduce client communication[8]. Hence, ℓ_2 computation is the bottleneck in our current protocol, and now we make this phase significantly cheaper.

8. This reduces w, while u stays the same. Boolean shares of gradients are compressed during transit, and unpacked at the servers to their full size.

More efficient sources of triples and OTs. Generation of Beaver triples between the servers makes ℓ_2 computation quite expensive. If we could find cheaper ways to source Beaver triples, then the cost of this phase can be brought down significantly to the point that it no longer is the bottleneck. To this end, we realize that each client can act as an "untrusted" (clients can be malicious) source of triples corresponding to the ℓ_2 computation that servers need to do for its gradient update. Since the soundness of the ℓ_2 computation relies on the correctness of these triples, servers need to first validate them before they can be used. For verification, we use the well-known sacrifice technique [43, 45, 73, 74] from SPD$\mathbb{Z}_{2^k}$ [43] which sacrifices one triple to verify the correctness (statistically) of the other. For gradient updates of m elements, each client prepares $2m$ triples and secret shares them between the servers, where each triple $(\alpha_i, \beta_i, \gamma_i)$ is generated by sampling $\alpha_i, \beta_i \xleftarrow{\$} \mathbb{Z}_{2^u}$ and setting $\gamma_i \leftarrow \alpha_i \cdot \beta_i$. The servers then sacrifice m of these triples to verify the other m, and finally use the surviving triples for ℓ_2 computation phase. The computational work of both clients and servers is minimal. In terms of communication, each client sends $6mu$ bits to each server, and each server communicates $2mu$ bits for verification[9], followed by $2mu$ bits for computing the squares. The communication cost from clients to the servers can be further reduced to just $2mu$ bits (total) by using a shared pseudo-random generator (PRG) seed [78] to generate both shares of α_i, β_i, and one share of γ_i such that the other share of γ_i is appropriately set by the client to ensure that the relation $\gamma_i = \alpha_i \cdot \beta_i$ holds, and is the only value that needs to be communicated. Talking about the end-to-end protocol, this approach increases client communication[10] (mw to $mw + 2mu$) by almost the same multiplicative factor as it reduces server communication $(m \cdot (2u + w) \cdot (\lambda + u)$ to about $mw \cdot (\lambda + u))$, however, given that the clients communicate only a small fraction of what servers communicate, we gain substantially in overall efficiency of the protocol.

Having clients generate Beaver triples not only makes our protocol significantly more efficient, but also makes it much simpler (requiring no OT extensions for generating triples). We now extend this idea to the COTs used in the bit composition phase. Although unlike ℓ_2 computation, this won't improve the efficiency of our end-to-end protocol by a big margin, it still makes it simpler, and further reduces the communication and computational load of the servers while increasing client communication. We default to this arrangement for a better distribution of work across parties, and later in this section, discuss the case of bandwidth-constrained clients.

Recall that the bit composition phase relies on OTs to convert Boolean shares of gradient updates to arithmetic shares. If clients generate random OTs (ROTs), i.e., OTs of the form $(m_0, m_1) \xleftarrow{\$} \mathbb{Z}_{2^u}$ given to the server acting

9. Performing the zero-check as a part of triple verification can be batched across all m checks by using a collision-resistant hash function.

10. For communicating Boolean shares of gradient update, the PRG trick can be used.

Algorithm 1 Bit Multiplication $\Pi^k_{\mathsf{BitMultUA}}$

Input: Bit shares $x^{(0)}, x^{(1)} \in \mathbb{Z}_2$ to multiply. If participating as the sender (OTSn role), then additional inputs include $(m_0, m_1) \in \mathbb{Z}_{2^k}$. If participating as the receiver (OTRc role), additional inputs include (j, m_j), where $j \in \{0,1\}$.
Output: $y^{(b)} \in \mathbb{Z}_{2^k}$ s.t. $x^{(0)} \wedge x^{(1)} = y^{(0)} + y^{(1)} \pmod{2^k}$

Protocol: Between Servers (Assuming S_0 is OTSn)

1: S_1 sends $d \leftarrow j \oplus x^{(1)}$ to S_0.
2: S_0 computes the pair $(v_0, v_1) \leftarrow (u_0 - r, u_1 - r + x^{(0)})$ and sends it to S_1, where $r \xleftarrow{\$} 2^k$, and $(u_0, u_1) \leftarrow (m_0, m_1)$ if d is 0 (swapped otherwise).
3: S_0 sets $y^{(0)} \leftarrow r$, S_1 sets $y^{(1)} \leftarrow v_{x^{(1)}} - m_j \pmod{2^k}$.

as OTSn and $(j \xleftarrow{\$} \{0,1\}, m_j)$ to OTRc, then servers can use them to do bit composition. Consider a bit x that the servers want to convert from its Boolean shares $(x^{(0)}, x^{(1)})$ to arithmetic form. We make use of the following equation:

$$\text{(Arithmetic)}\quad x = x^{(0)} \oplus x^{(1)} = x^{(0)} + x^{(1)} - 2(x^{(0)} \wedge x^{(1)})$$

where additions and subtractions are in the arithmetic domain. Notice that the only non-local operation needed is to compute $x^{(0)} \wedge x^{(1)}$, called bit multiplication. We describe the protocol $\Pi_{\mathsf{BitMultUA}}$ to do bit multiplication $x^{(0)} \wedge x^{(1)}$, where S_b holds $x^{(b)}$ for $b \in \{0,1\}$, in Algo. 1; this consumes an ROT in a straightforward way. This provides a way for the servers to use client-supplied OTs for bit composition, but leaves an issue wide-open - how do the servers verify if the OTs are correct? ROTs cannot be efficiently verified by the servers, so we rather have clients send a special kind of COTs, called Δ-COTs, which admit a batch-verification procedure [72], and convert them back to ROTs at the servers. In a Δ-COT, every pair of messages held by OTSn satisfies the constraint $m_0 \oplus m_1 = \Delta$ for a fixed but random $\Delta \in \{0,1\}^\lambda$. A batch of Δ-COTs $\{(m_i, m_i \oplus \Delta), (j_i, m_{j_i})\}_i$ can be verified by checking their random linear combination for the Δ-offset property [72], and each of them can then be converted to ROTs at the servers by the local operation of a hash function [72] which breaks the correlation and makes the two messages in each OT look random (just like an ROT). To reduce the communication from clients to servers, we again use shared PRG seeds to only communicate m_{j_i} to OTRc, and all other values m_i, Δ, j_i are generated by expanding the PRG at respective parties. When servers generate the OTs themselves (on the fly), they incur a communication of $mw(\lambda+u)$ bits for bit composition which is now reduced to $mw(2u+1)$, and clients now communicate $mw\lambda$ additional bits. We present an optimization in Section 3.2.2 which further cuts the server communication by half, thereby achieving the same total communication for bit composition as on-the-fly OT case.

Bandwidth-constrained clients. In situations where clients are mobile or edge devices who might be bandwidth-constrained (because of their geographic location, for example), sending Δ-COTs for bit composition can be quite demanding; each client needs to send $m(w + 2u + w\lambda)$ bits which is completely dominated by $mw\lambda$ from Δ-COTs. In these cases, clients can opt out of sending Δ-COTs and have the servers generate them on the fly as mentioned in Section 3.1. This frees up their bandwidth burden significantly while mostly maintaining the end-to-end efficiency of our protocol, and should be enough for most cases. In the very rare situation where clients can only communicate the bare minimum, we let them opt out of even sending the Beaver triples. This however, comes at the cost of increased work (computation and communication) at the servers. As long as only a small fraction of clients are severely bandwidth-constraint, our protocol still largely maintains its efficiency guarantees. We experimentally confirm this in Section 4.3. We next discuss an approach to reduce client communication without increasing server communication, but at the cost of increased computation for both roles.

Using Pseudorandom Correlation Generators. Two-party pseudorandom correlation generators (PCGs) [30, 31, 94] are defined by two algorithms - Gen and Expand. The former generates a pair of succinct seeds (k_0, k_1) which are distributed between the two servers (S_b gets k_b). The servers use the Expand algorithm to *locally* expand their seeds to generate a large number of correlations. Prior work [30, 31, 94] has presented PCG constructions for both Δ-COTs and beaver triple[11] correlations. In our setting, each client can generate PCG seeds for the correlations it wants to send, and the servers can then expand them out and validate them as they would validate regular correlations. This reduces client communication significantly. Taking Δ-COT as an example which currently forms the communication bottleneck of the clients, with Boyle et al.'s [30] (resp. Schoppmann et al. [94]) silent-OT seeds, clients would communicate less than a bit (resp. 6-7 bits) per COT compared to λ (e.g. 128) bits otherwise. Schoppmann et al. [94] has higher communication than Boyle et al. [30], but requires much smaller computation, and computation is what matters more for our setting because clients generate the seeds locally. We leave it to future work to develop more efficient PCG-based OT extension protocols (like Ferret [112]) in the trusted dealer model [29] like ours.

3.2.2. Optimizing Correlation Usage

We saw how servers can benefit from getting correlations like Beaver triples and OTs from the clients. We now propose further optimizations for using *both* these correlations which reduces server communication by another $2\times$.

Pre-aligned Δ-COTs. Bit multiplication using Δ-COTs (hashed to ROTs) of k bits (Algo. 1) requires $2k+1$ bits of communication across 2 rounds. We can bring this down to k bits and a single round by proposing the use of *pre-aligned* Δ-COT correlations. In a pre-aligned Δ-COT, the client sets the choice bit to be same as OTRc's bit-share which consumes that COT during bit multiplication. In particular, by setting $j \leftarrow x^{(1)}$, the first message d is always

11. Servers can also use OTs to generate beaver triples [48].

Algorithm 2 Aligned Bit Multiplication $\Pi^{j}_{\mathsf{BitMult}}$

Input: Bit shares $x^{(0)}, x^{(1)} \in \mathbb{Z}_2$ to multiply. If participating as the Sender (OTSn role) then additional inputs include Δ, $q \in \mathbb{F}_{2^\lambda}$. If participating as the Receiver (OTRc role) additional inputs include $t \in \mathbb{F}_{2^\lambda}$. Let $H : [\ell] \times \mathbb{F}_{2^\lambda} \to \mathbb{Z}_{2^\lambda}$ be a hash [72] in the random-oracle model.
Output: $y^{(b)} \in \mathbb{Z}_{2^j}$ s.t. $x^{(0)} \wedge x^{(1)} = y^{(0)} + y^{(1)} \pmod{2^j}$

Protocol: Between Servers (Assuming S_0 is OTSn)

1: S_0 computes $v_0 \leftarrow H(c||q)$ and $v_1 \leftarrow H(c||q+\Delta)$ and sets $y^{(0)} \leftarrow -v_0 \pmod{2^j}$, where c is a global counter.
2: S_0 sends $u \leftarrow v_0 + v_1 + x^{(0)} \pmod{2^j}$ to S_1.
3: S_1 computes $v \leftarrow H(c||t)$.
4: S_1 sets $y^{(1)} \leftarrow x^{(1)}u + (-1)^{x^{(1)}}v \pmod{2^j}$

Algorithm 3 Local OT Correlation Generation LocalOT

Input: Choice bits $x \in \mathbb{Z}_2^n$ and an offset $\Delta \in \mathbb{F}_{2^\lambda}$.
Output: Q and T s.t. $Q = T + x \cdot \Delta \in \mathbb{F}_{2^\lambda}^n$

Protocol: Locally performed at each Client $c \in \mathcal{C}$

1: **for** $j \in [n]$ **do**
2: Sample $q_j \in \mathbb{F}_{2^\lambda}$. Let $x_j \in \mathbb{F}_2$ be the j^{th} bit of x.
3: Set $t_j \leftarrow q_j + x_j \cdot \Delta \in \mathbb{F}_{2^\lambda}$
4: **end for**
5: Set $Q \leftarrow \{q_0, \ldots, q_{n-1}\}$ and $T \leftarrow \{t_0, \ldots, t_{n-1}\}$.

zero, and therefore, doesn't need to be sent. Moreover, the two messages (v_0, v_1) sent by OTSn can also be reduced to a single message by using ideas from COT extension protocols [16, 72]. We refer to such Δ-COT correlations as being *pre-aligned*, and the optimized bit multiplication which uses them to be *aligned bit multiplication*, Π_{BitMult}, presented in Algo. 2. For completeness, in Algo. 2, we assume Δ-COTs as input and include hashing to ROTs as part of the protocol.

Square correlations. Beaver triples allow multiplication of any pair of secret-shared values x, y. For our protocol, we only require computation of squares of secret-shared values, i.e., when $x = y$, and therefore, using Beaver triples is sub-optimal[12]. We introduce *square correlations* as the optimized alternative for computing squares (part of ℓ_2 computation). A pair of values (α, γ) form a square correlation over $\mathbb{Z}_{2^u}$ if $\gamma = \alpha^2$ and $\alpha \xleftarrow{\$} \mathbb{Z}_{2^u}$. Square correlations can be used in a straightforward way (similar to Beaver triples) to compute squares over secret-shared values. However, the tricky part is their verification. Following an approach similar to SPDZ sacrifice [43, 45, 73, 74] for Beaver triples, soundness boils down to the distribution of quadratic residues in the finite ring. We prove in Lemma 2 that using an odd value as the random challenge in the verification step can provide soundness with just a 3-bit loss in statistical security.

12. When PCG-based OTs are used to generate Beaver triples, triple generation becomes the bottleneck step. Replacing triples with square correlations improves the efficiency of this step by 2x.

Algorithm 4 Bit Composition $\Pi^{w,\ell}_{\mathsf{BitComp}}$

Input: Bit shares $x^{(b)} \in \mathbb{Z}_2^w$ to convert, where $b \in \{0, 1\}$. If participating as the Sender (OTSn role) then additional inputs include Δ, $Q \in \mathbb{F}_{2^\lambda}^w$. If participating as the Receiver (OTRc role) additional inputs include $T \in \mathbb{F}_{2^\lambda}^w$.
Output: $z^{(b)} \in \mathbb{Z}_L$ where $L = 2^\ell$ such that

$$z^{(0)} + z^{(1)} = \sum_{i=0}^{w-1} 2^i (x_i^{(0)} \oplus x_i^{(1)})$$

Protocol: Between Servers (Assuming S_0 is OTSn)

1: S_b sets $z^{(b)} \leftarrow 0 \in \mathbb{Z}_L$, where $b \in \{0, 1\}$.
2: **for** $i \in \{0, \ldots, w-1\}$ **do**.
3: Let $\ell' = \ell - (i+1)$.
4: S_0 sets $y^{(0)} \leftarrow \Pi^{\ell'}_{\mathsf{BitMult}}(x_i^{(0)}, \Delta, Q_i)$.
S_1 sets $y^{(1)} \leftarrow \Pi^{\ell'}_{\mathsf{BitMult}}(x_i^{(1)}, T_i)$.
5: Set $z^{(b)} \leftarrow z^{(b)} + 2^i(x_i^{(b)} - 2y^{(b)}) \bmod L$, where $x^{(b)}, y^{(b)}$ are considered as elements of $\mathbb{Z}_L$.
6: **end for**

End-to-end protocol. We present our end-to-end semi-honest private protocol for FL in Algo. 5. Details about the round complexity of our protocol are deferred to Appendix D.

Algorithm 5 Secure Aggregation for FL (Π_{Agg})

Input: Gradient vectors of size m. Let $n = m \cdot w + 2u + \lambda + \kappa$. τ denotes the minimum no. of clients whose gradients need to be aggregated each round. For the $\ell_2^{\sqcap}$ defense, μ (resp. $2^w - 1$) is the ℓ_2 (resp. $\ell_\sqcap$) bound to enforce. We assume that Server S_0 is the OTSn and S_1 is the OTRc.
Output: Global aggregate vector with m values, where co-positioned w-bit values across clients' vectors are aggregated over $\mathbb{Z}_L$.

Input Sharing Phase: (Locally at each Client)

1: For each client $c \in \mathcal{C}$, let x denote the data of this client (clipped to the current ℓ_2 bound, μ) which consists of a vector of size m with w-bit values. Let $x_{i,j} \in \mathbb{Z}_2$ where i refers to the vector index and $i \in \{1, 2, \cdots, m\}$ and $j \in [w]$ is the bit-index.
2: Generate shares $x^{(0)}, x^{(1)}$ of x over $\mathbb{Z}_2^{m \times w}$ and send $x^{(b)}$ to server b.

OT Generation: (Locally at each Client)

1: Each client $c \in \mathcal{C}$ samples $\Delta \xleftarrow{\$} \mathbb{F}_{2^\lambda}$ and $r \xleftarrow{\$} \mathbb{Z}_2^{2u+\lambda+\kappa}$.
2: Each client generates OT correlations Q, T and Q', T' using the LocalOT sub-routine on inputs $x^{(1)} \in \mathbb{Z}_2^{m \times w}$ (flattened) and Δ and r, Δ respectively:

$$\begin{aligned} (Q, T) &\leftarrow \mathsf{LocalOT}(x^{(1)}, \Delta) \\ (Q', T') &\leftarrow \mathsf{LocalOT}(r, \Delta) \end{aligned} \tag{1}$$

3: Each client sends Δ and $Q \leftarrow (Q||Q')$ to the S_0 (OTSn Server), and r and $T \leftarrow (T||T')$ to S_1.

Square Correlation Generation: (Locally at each Client)

1: Set $v \leftarrow u + \kappa + 3$[13].
2: Each client $c \in \mathcal{C}$ samples $\{a_i\}_{i=1}^{2m} \xleftarrow{\$} \mathbb{Z}_{2^v}$.
3: Each client generates arithmetic shares $W^{(0)}, W^{(1)}$ of $\{(a_i, d_i)\}_{i=1}^{2m}$, where $d_i = a_i^2 \bmod 2^v$ and sends them to the respective server.

OT Verification: (Between Servers)

Servers perform the following steps for each $c \in |\mathcal{C}|$:

1: Servers S_0, S_1 collectively sample random values $\{\chi_1, \ldots, \chi_n\} \in \mathbb{F}_{2^\lambda}^n$. S_1 (OTRc role) parses $\hat{x} \leftarrow (x^{(1)}||r) \in \mathbb{F}_2^n$ and computes:

$$\tilde{x} \leftarrow \sum_{j=1}^{n} \hat{x}_j \cdot \chi_j \quad \text{and} \quad \tilde{t} \leftarrow \sum_{j=1}^{n} T_j \cdot \chi_j \tag{2}$$

where $T_j \in \mathbb{F}_{2^\lambda}$ is the jth correlation in T.
2: S_1 sends $\tilde{x}, \tilde{t}$ to S_0 and S_0 computes

$$\tilde{q} \leftarrow \sum_{j=1}^{n} Q_j \cdot \chi_j \tag{3}$$

where $Q_j \in \mathbb{F}_{2^\lambda}$ is the jth correlation in Q.
3: S_0 checks if $\tilde{t} = \tilde{q} + \tilde{x} \cdot \Delta$ and reject the client if it fails.
4: S_0, S_1 discard the last $(\lambda + \kappa)$[14] OT correlations. They further split the remaining correlations into two sets $(Q^A, T^A), (Q^B, T^B)$ with mw in the first set. They parse Q^A, T^A to their folded form $\in \mathbb{F}_{2^\lambda}^{m \times w}$.

Square Correlation Verification: (Between Servers)

Servers perform the following steps for each $c \in |\mathcal{C}|$:

1: S_b sets $\hat{W}^{(b)} \leftarrow \{\}$.
2: **for** each pair of correlations $(a^{(b)}, d^{(b)}), (\hat{a}^{(b)}, \hat{d}^{(b)}) \in W^{(b)}$ **do**
3: Collectively sample an random odd value $t \in \mathbb{Z}_{2^v}$
4: Servers open $e \leftarrow ta - \hat{a}$.
5: S_0 computes $t^2 d^{(0)} - \hat{d}^{(0)} - 2tea^{(0)} + e^2$
S_1 computes $t^2 d^{(1)} - \hat{d}^{(1)} - 2tea^{(1)}$
6: Servers check that they computed shares of zero; If yes, $\hat{W}^{(b)} \leftarrow \hat{W}^{(b)}||(a^{(b)}, d^{(b)})$ else reject client.
7: **end for**
8: Servers parse $\hat{W}^{(b)}$ as $\{(a_i^{(b)}, d_i^{(b)})\}_i$.

Bit Composition Phase: (Between Servers)

Servers perform the following steps for each $c \in |\mathcal{C}|$:

1: **for** $i \in [m]$ **do**
2: S_0 sets $z_i^{(0)} \leftarrow \Pi_{\mathsf{BitComp}}^{w,u}(x_i^{(0)}, \Delta, Q_i^A)$
S_1 sets $z_i^{(1)} \leftarrow \Pi_{\mathsf{BitComp}}^{w,u}(x_i^{(1)}, T_i^A)$
3: **end for**

13. Square correlations are generated over v bits by the clients, then verified over v bits by the servers, and finally, upper $\kappa + 3$ bits are locally dropped to yield *validated* correlations over u bits. Refer to Lemma 2 for more details.

14. Extra $\lambda + \kappa$ random OTs were needed to prevent any private information leakage from $\tilde{x}$. This follows from lemma 2 [72] which states (informally) that a random $(\lambda + \kappa) \times \lambda$ matrix over $\mathbb{F}_2$ is full rank except negligible probability in κ.

ℓ_2 Computation Phase: (Between Servers)

Servers perform the following steps for each $c \in |\mathcal{C}|$:

1: S_b sets $z^{(b)} \leftarrow 0 \in \mathbb{Z}_{2^u}$, where $b \in \{0, 1\}$ and $i \in [m]$.
2: **for** $i \in [m]$ **do**
3: Servers open $e \leftarrow z_i - a_i$.
4: S_0 sets $z^{(0)} \leftarrow z^{(0)} + d_i^{(0)} + 2ez_i^{(0)} - e^2 \bmod 2^u$
S_1 sets $z^{(1)} \leftarrow z^{(1)} + d_i^{(1)} + 2ez_i^{(1)} \bmod 2^u$
5: **end for**

ℓ_2 Enforcement Phase: (Between Servers)

Servers perform the following steps for each $c \in |\mathcal{C}|$:

1: S_0 sets $z^{(0)} \leftarrow z^{(0)} - \mu^2$.
2: S_0, S_1 extract the sign bit of z by evaluating an adder securely ([48]) with inputs $z^{(b)}$. For AND computations, S_0 uses Q^B, Δ and S_1 uses r, T^B. Each AND uses two $\Pi_{\mathsf{BitMultUA}}^1$ invocations (Algo. 6).
3: If the sign bit is zero, reject the client.

Aggregation Phase: (Between Servers)

1: For $i \in [m]$ and $b \in \{0, 1\}$, S_b adds the $z_i^{(b)}$ values of all the clients together into $y_i^{(b)}$.
2: Reconstruct y_is from shares if inputs from more than τ fraction of clients got aggregated. Otherwise output $\perp$.

3.3. Achieving Malicious Privacy

The protocol that we have built so far is efficient, supports $\ell_2^\square$ defense, and guarantees privacy against a semi-honest server. In this section, we present new ideas for guaranteeing privacy in the face of a maliciously corrupt server while incurring an extremely low overhead compared to our semi-honest protocol. Unlike a semi-honest server, a malicious server can send malformed protocol messages to violate privacy of individual gradients. As an example, consider aligned bit multiplication (Algo. 2) where S_0 is malicious. To ensure privacy, we require that S_0 learns nothing about the bit $x^{(1)}$ held by S_1. Since a malicious S_0 can send malformed messages, it can construct its message as $u \leftarrow v_0 + v_1 + x^{(0)} + \delta$, where δ is some error that it introduces. Now, δ will contribute to the final aggregate only if $x^{(1)}$ was one because this is only when u is used by S_1. Therefore, the final result (which is opened to both servers) reveals $x^{(1)}$ to S_0. This is just one way in which a malicious server may compromise gradient privacy. We need to eliminate these attacks to reach our goal of malicious privacy. We start by looking at existing MPC techniques which are commonly used to achieve malicious security.

A naïve way to achieve malicious privacy would be to directly use a malicious-secure arithmetic black box [43, 44, 51] for all the operations that we need in our protocol. However, this will require all clients in $\mathcal{C}$ and the two servers to act as (full) parties in the MPC protocol, making this solution completely impractical for many reasons - MPC protocols don't scale well with so many parties, each client's communication becomes too high, and they have to interact with each other and the servers across multiple rounds for just one invocation of secure aggregation. Additionally, this

solution assumes that each client is as resource-capable as the servers, which doesn't hold for many setups like cross-device FL. A better solution would be to do malicious-secure 2PC between the two servers, and have the clients send authenticated bit-shares of their gradients to the servers. If we ignore collusion between the malicious server and a subset of clients for a moment, then a global authentication key (MAC key [43, 51, 87]) can be kept secret-shared between the servers, while the clients can know the key in clear. This allows the clients to generate authenticated Boolean shares of their gradients locally. However, under collusion, this approach breaks down because the malicious server can easily learn the key through any corrupted client. To withstand this, clients can be allowed to interact with the servers to authenticate their Boolean shares, but this is quite inefficient, and moreover, gives a single malicious client the power to abort the entire protocol by adding inconsistencies during authentication. Although servers can introduce more steps to check for valid authentication, this approach has a big overhead over semi-honest.

Distilling privacy-sensitive steps. To build a satisfactory solution, we observe that not all steps in the protocol need to be maliciously secure; the final aggregation phase (Algo. 5) is a step which doesn't need to be malicious secure. This is because any error δ introduced by the malicious server during reconstruction of the final aggregate is unconditionally reflected in the output, and therefore, doesn't leak any private information. In other words, the aggregation phase only admits an additive error attack [39] which doesn't affect privacy. However, it does affect correctness of the output, and that is why, we don't guarantee correctness of aggregation when a server is malicious. Given this insight, if we consider using information-theoretic MACs for malicious privacy, then we can have each client generate its own MAC key, use it to *locally* generate authenticated Boolean shares of its gradient update, and send them to the server along with arithmetic shares of the MAC key. The servers can then use malicious-secure 2-party Boolean-to-arithmetic conversion protocols [44, 51] and secure multiplication using authenticated Beaver triples [43] to perform the $\ell_2^\square$ defense. The final step of aggregation can ignore the MACs because, firstly, they vary across clients and can't be aggregated, and secondly, we don't need to catch malicious tampering in this phase. Despite being the most efficient solution that uses MACs which we have drafted so far, it still puts a communication overhead of more than an order of magnitude over the semi-honest base owing to the Boolean-to-arithmetic conversion step [51].

In this work, we take a different direction for guaranteeing malicious privacy. We observe that all the steps (Algo. 5) run by the servers which require malicious protection (OT verification, square correlation verification, bit composition, ℓ_2 computation, and ℓ_2 enforcement) are run independently for each client; the aggregation phase, on the other hand, doesn't have this independence property, but can be safely ignored for malicious protection. It will be convenient to think of it this way - the servers start $|\mathcal{C}|$ separate and independent instances implementing the $\ell_2^\square$ defense where each instance is dedicated to a specific client. As long as we can guarantee that for each instance, the corresponding client can know apriori all the messages that the servers send to each other without affecting soundness of the $\ell_2^\square$ defense, then we can achieve malicious privacy by *transcript emulation*. For each instance, we let the corresponding client emulate the entire transcript of server interaction, and send it to both the servers. The honest server can now cross-check the transcript it observes while interacting with the other server against the transcript from the client, and easily detect malicious tampering. If tampering is detected, we stop any further processing (censor the client) on the inputs of the concerned client to maintain their privacy and carry on with the rest of the protocol. If the fraction of censored clients go beyond a configured threshold τ (e.g., all but one clients get censored), the honest server aborts the protocol.

Coming back to the assumption we made above that knowing protocol messages doesn't affect soundness of the defense, we now argue why this holds. Consider the $\ell_2^\square$ defense outside the secure computation context. It takes bits as inputs, converts them to elements of a larger ring, computes sum of their squares and checks if they obey an upper bound. Nowhere in this process do we rely on any secret value (like a random challenge) which needs to be hidden from the client. The computation being done by the servers is deterministic from the client's perspective. As we shift to secure computation, the only change that happens is that all these operations are replaced by their secure counterparts which consume cryptographic correlations like Δ-COTs and square correlations. If these correlations are valid, we have soundness. In our protocol, since we get these correlations from the clients, their validity is questionable, and therefore, servers need to perform validity checks on them. Unlike the $\ell_2^\square$ defense, these validity checks base their soundness on a random *challenge* (χ_i in OT verification, and t in square correlation verification) being hidden from the client until it submits the correlations. Therefore, we split the client protocol into two rounds - in the first round, clients send Boolean shares and cryptographic correlations to the servers, and on receiving the random challenge back from the servers, in the second round, clients send the transcript of server-server interaction for all steps except the final aggregation phase.

Reducing transcript communication. We can reduce client communication by having them send a short digest of the protocol transcript; this can be implemented using a collision-resistant hash function. However, naïvely using this optimization doesn't provide malicious privacy because the honest server cannot cross-check the observed transcript against the digest until the protocol has reached the end of ℓ_2 enforcement phase, and the results of intermediate checks (OT, square correlation verification, and ℓ_2 enforcement) can reveal sensitive information. We fix this issue by deferring the opening of result of these intermediate checks until the ℓ_2 enforcement phase is complete. We change the transcript digest to only include the messages in the privacy-sensitive

phases which don't convey results of intermediate checks. For example, in square correlation verification, all messages until the zero-check (conveys result) are part of the digest and the messages in zero-check are ignored for the digest[15]. After receiving the digests from the clients, the servers first cross-check the digest, and only if it passes, they execute the opening step of intermediate checks.

We now dive deeper into how transcript emulation happens, and later discuss how we can squish the rounds of clients from two to one, thereby achieving one-shot clients.

3.3.1. Transcript Emulation

For a client to emulate the interaction between the servers, all we need is for the server protocol to be deterministic in the client's view. Recall from Section 3.2 that our protocol has two variants - one where clients supply cryptographic correlations to the servers, and other where the servers generate them on the fly. We now discuss how clients emulate server interaction for both these cases.

Client-generated correlations. When clients supply OT and square correlations to the servers, all privacy-sensitive steps in the server protocol except the OT and square correlation verification can be determined locally by the client because there is no external randomness involved (the Boolean input shares also come from the client). OT and square correlation verification rely on a random challenge that is generated by the servers and can only be known by the client once it submits the correlations. By making the client protocol two-round, we ensure that after the first round, the client learns this random challenge from the servers, and therefore, it can generate the entire transcript of the server side computation by the second round.

Server-generated correlations. When a subset of clients are bandwidth-constrained, a part or all of the correlations for processing their inputs are interactively generated by the servers. Given that this interactive sub-protocol (such as the IKNP OT extension [68]) uses randomness unknown to the clients, they cannot directly generate the transcript. To address this, we maintain a unique common random tape for each client-server pair which is used in the interactive correlation-generating sub-protocol between the servers. Each server has its own common random tape with each client, and given this tape, all messages sent by the server are deterministic in the client's view. In practice, large random tapes can be efficiently shared using PRGs.

One-Shot Clients. As a final optimization, we now make our client protocol single-round (one-shot clients). This is desirable in settings (like cross-device FL) where the availability of client devices is quite uncertain. Moreover, our solution has the additional benefit of obviating the need for a secure coin-flipping protocol to sample common random values between S_0, S_1 (steps 1, 4 in Algo. 5).

Our current protocol divides the client-server interaction into two rounds because the soundness of the correlation verification phases rely on the client not knowing the random challenge sampled by the servers. Once the client has already submitted the correlations, the random challenge no longer needs to be hidden. If the clients can somehow generate the random challenge locally without breaking the soundness of our verification phases, then we get one-shot clients. We begin by observing that this part of our protocol can be cast as the so-called *public-coin* protocol [27, 106] where the clients act as provers, and the servers collectively act as a distributed verifier. The Fiat-Shamir transform [54] (analyzed in the random-oracle model) provides a way for the prover to locally generate verifier's challenge while still maintaining soundness. The idea behind the transform is to generate the challenge by applying a secure hash function to the protocol inputs and transcript observed so far. Since the verifier in our protocol is a virtual party distributed across the two servers, we can't apply the Fiat-Shamir transform directly, and appeal to the distributed variant introduced by Boneh et al. [27]. We now describe in more detail how we use this transform. We focus on the generation of random challenge for OT verification, and our arguments straightforwardly extend to square correlation verification.

The random challenge in OT verification are the χ_i values which are collectively sampled by S_0, S_1 by invoking a coin-flip sub-protocol. When using the Fiat-Shamir transform with the hash function $H : \{0,1\}^* \to \mathbb{F}_{2^\lambda}$, we define $\chi \leftarrow H(\chi^{(0)}, \chi^{(1)})$, where $\chi^{(b)} \leftarrow H(b, x^{(b)}, p, q, W^{(b)}, \theta_b)$, θ_b is a blinding variable sampled by the client (and communicated to S_b) to ensure that $\chi^{(b)}$ doesn't leak information about S_b's share to S_{1-b}, and $(p, q) \leftarrow (\Delta, Q)$ if b is zero, and (r, T) otherwise. Servers can then generate $\chi^{(b)}$ locally and share it with the other server to get a seed χ, which is then expanded using a PRG to yield $\{\chi_i\}_i$. Notice that with the use of this transform, the servers no longer need the coin-flip sub-protocol to sample $\{\chi_i\}_i$.

4. Evaluation

In our evaluation, we focus on the secure aggregation task[16], and answer the following questions:

- *How does* ELSA *compare to state-of-the-art FL solutions which consider malicious actors? (Section 4.1)*
- *What is the breakdown of performance of individual components in our protocol? (Section 4.2)*
- *How is our performance affected when a subset of clients are bandwidth-constrained? (Section 4.3)*

Implementation. We implemented ELSA in Rust and make our code public at https://github.com/ucbsky/elsa. To handle a large amount of simultaneous client sessions, we use Tokio [6] as our communication backend. For finite field multiplication, we employ optimizations from Keller et al. [72] and EMP Toolkit [7]. We use miTCCR [59] as our hash function and hardware-accelerated AES for PRG. When OTs are to be generated between the servers (in Prio+ and for bandwidth-constrained clients in ELSA), we use IKNP [68] OT extension. Most of our evaluation

15. Isolated errors in the result-conveying messages of intermediate checks are only limited to additive attacks [39] which don't violate privacy.

16. We ignore local training since that is tangential to this work.

doesn't include our one-shot clients optimization given its practically negligible impact on performance[17] (as shown in Section 4.2).

Experimental Setup. To emulate a realistic scenario of each client opening a separate connection with the servers, we implement a meta-client; it opens independent connections per client and ensures the allocated bandwidth for each connection stays reasonable. We deploy our meta-client on an r5n.16xlarge AWS instance (Ohio) with 64 vCPUs, and servers on two r5.8xlarge instances (Ohio and N. Virginia) with 32 vCPUs and 10Gbps bandwidth. This is quite similar to RoFL's setup in terms of compute power and bandwidth. We use a t2.medium instance when evaluating the computational overhead of a single client. Throughout our experiments, server time is measured after all communication from the clients for that round is finished. Unless otherwise stated, we consider individual values in the gradients as 32 bits, and perform aggregation and ℓ_2 computation over 64 bits, i.e., $w = 32$, $\ell = u = 64$, $v = 128$ and $\kappa = 61$.

4.1. FL with Malicious Actors

We begin our evaluation by focusing on prior systems for FL which consider malicious parties. The rest of the section is split into two parts depending on the trust model of the baseline system.

Comparison with distributed trust baseline. Prio [41] is a system for privacy-preserving collection of aggregate statistics, and secure aggregation for FL is a subset of its supported functionality. We use the rust crate for Prio [10] and merge it into our framework for a similar multithreading to ELSA. Since its implementation doesn't directly support the $\ell_2^{\sqcap}$ defense, we rather perform this comparison over only the relaxed ℓ_∞ defense (using their count functionality) with the bound as $\ell_{\sqcap}$. Table 2 shows the results of our comparison. The server runtime in ELSA is about 8.5-16x faster than Prio, and the client is 2-3.6x faster, while the total runtime enjoys up to 8x improvement. The runtime of clients increase when more clients are selected per round because of the overhead associated with handling more concurrent active connections at the receiving end (servers) and rationing of slightly lower bandwidth per client by our meta-client. In terms of communication, each client in ELSA communicates a little less than Prio, but the server communication in Prio is negligible (artifact of their proofs) compared to our protocol. The total communication of ELSA and Prio (all clients and servers included), on the other hand, is still comparable with ELSA communicating about 1.5x of Prio.

Comparison with single-aggregator baseline. Next, we compare with RoFL [34], the state-of-the-art single-aggregator FL protocol which is more practical than EIFFeL [40][18]. RoFL supports $\ell_2^{\sqcap}$ defense, and while the authors don't consider malicious privacy as a property of their system, we believe their techniques should provide this property, and give them the advantage here. Given the similarity of our experimental setup with that considered in their paper, we use results from their evaluation section[19]. We use 8-bit probabilistic quantization [76] for gradient values to match their setup. Table 3 shows the comparison for three parameter sizes, each corresponding to a network evaluated by RoFL. *CIFAR-10 S* and *CIFAR-10 L* correspond to LeNet5 [80] and ResNet-18 [63] trained on CIFAR-10 [77], respectively, and *SHAKESPEARE* is an LSTM [65] trained on the Shakespeare dataset [36]. We achieve 146-305x end-to-end runtime improvement while incurring about the same total communication as RoFL. In RoFL, all communication is from clients to servers, and clients in ELSA communicate 1.6-1.8x lower than RoFL.

#Clients	#Params	Prio		ELSA	
		Client	Server	Client	Server
50	100k	14.3 (59.1)	23.3 (0.002)	4.6 (51.6)	2.7 (640)
100	100k	14.8 (59.1)	48.9 (0.005)	7.1 (51.6)	3.8 (1280)
200	100k	16.5 (59.1)	99.5 (0.010)	8.4 (51.6)	6.1 (2560)
50	500k	63.6*(262.2)	102.9 (0.002)	17.5 (258.0)	11.2 (3200)
100	500k	67.7*(262.2)	218.4 (0.005)	23.2 (258.0)	17.3 (6400)
200	500k	78.3*(262.2)	457.9 (0.010)	38.0 (258.0)	31.4 (12800)

* Client ran out of memory. We report underestimates here.

TABLE 2: Comparison of runtime (sec) and data sent (MB in parenthesis; per client and per server) in ELSA vs Prio for relaxed ℓ_∞ defense (with malicious privacy).

#Params	RoFL		ELSA	
	Runtime	Comm.	Runtime	Comm.
62k (CIFAR-10 S)	278	0.8	1.9	0.9 (0.5, 0.4)
273k (CIFAR-10 L)	2229	3.8	7.3	4.0 (2.1, 1.9)
818k (SHAKESPEARE)	4742	11.4*	18.1	12.0 (6.3, 5.7)

* Corrected from what RoFL reported.

TABLE 3: Comparison of ELSA with RoFL for $\ell_2^{\sqcap}$ defense with malicious privacy (only secure aggregation). Values denote end-to-end runtime (sec) and total data sent (GB). Parenthesis show split of communication between all clients and servers, respectively. $|\mathcal{C}| = 48$, $w = 8$ and $\ell = 64$.

4.2. Performance Breakdown of ELSA

In this section, we look into how different parts of our protocol impact performance. First, we consider protocol-level costs starting from our base protocol which provides privacy only against semi-honest server and enforces $\ell_{\sqcap}$ bound and going up to our malicious private protocol with ℓ_2 bounds; we refer to this as the layerwise cost. Second, we look deeper into how different phases of our protocol affect the end-to-end runtime.

Layerwise cost. We call our first layer "ELSA SH $\ell_{\sqcap}$" and it is quite similar to Prio+ with a difference that OTs are all supplied by the clients and validated by the servers before

17. The main benefit of this optimization is in deployment when considering unreliable client availability, and not in performance.

18. EIFFeL clients require about an order of magnitude more bandwidth.

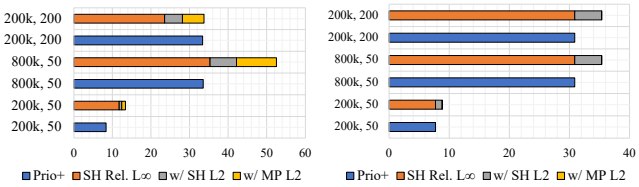
19. We couldn't successfully run their code for our experiments.

using them for bit composition. Then the second (resp. third) layer, called "ELSA SH $\ell_2^{\sqcap}$" (resp. "ELSA MP $\ell_2^{\sqcap}$"), add $\ell_2^{\sqcap}$ defence with semi-honest (resp. malicious) privacy. We present the runtime and communication costs of these layers in Fig. 2. When the number of clients is large (e.g., 200) and gradients are moderately sized (e.g., 200k), our end-to-end runtime with malicious privacy and $\ell_2^{\sqcap}$ defense is comparable to Prio+ even when our defense subsumes theirs and defends against the more powerful malicious server; this is because moderate sized gradients have cheaper transcript emulation than large ones, and we observe that for the same amount of data communicated from the clients, more clients with each sending lesser data is faster than the flip case (this doesn't benefit Prio+ where client communication is already quite small). For malicious privacy, the added overhead is very small ranging from 7-25% of the semi-honest runtime given that it can withstand the much stronger malicious corruption. In terms of communication, our first layer protocol has the same total communication as Prio+ (with IKNP), and the SH $\ell_2^{\sqcap}$ layer incurs an added cost of about 14%. With malicious privacy, owing to our optimization of using transcript digests, the additional overhead on total communication is negligible. Note that although Prio+ can be instantiated with the PCG-based OT extension [30, 31, 112] backends to bring down total communication (at the cost of more compute) to about a third of the IKNP backend, as mentioned in Section 3.2.1, ELSA can also be used in the PCG mode to enjoy similar benefits in total communication. Moreover, in the PCG mode, the client communication of ELSA is comparable to Prio+. We leave the experimental evaluation of this mode for future work. We also compare the runtime of our protocol with and without one-shot clients. For 200k parameters and 200 clients (malicious privacy and $\ell_2^{\sqcap}$), client runtime changes from 19s (two-round clients) to 17s (with one-shot), and server runtime changes from 14s to 19s.

Finer breakdown. We provide a detailed breakdown of the runtime costs of different parts of our protocol in Fig. 3. *Client prepare* and *transcript emulation* refer to the local computation at clients where the former captures everything except generation of the transcript digest of server interaction (captured by the latter). This is followed by *client communication*. *Correlation verification* refers to verification of square correlations at the servers. Other phases are self-explanatory. As expected, the runtimes of the prominent phases are largely governed by the amount of communication they require. The client communication phase is most expensive during which both clients and the servers have to be online, and it is closely followed by bit composition that happens between the servers. Rest of the phases take only a fraction of the total time.

4.3. Clients with Limited Bandwidth

Our work supports bandwidth-constrained clients by shifting the generation of correlated randomness to the servers. We now show how this strategy affects our performance as a function of the fraction of such clients and the degree of their constraint.



(a) End-to-end runtime (sec) (b) Total comm. (GB)

Figure 2: Comparison of different layers of ELSA with Prio+. SH and MP denotes privacy against semi-honest and malicious servers, respectively. Rel. ℓ_∞ enforces just the $\ell_\sqcap$ bound and L2 refers to $\ell_2^{\sqcap}$ defense. Vertical axis denotes parameter size, number of clients.

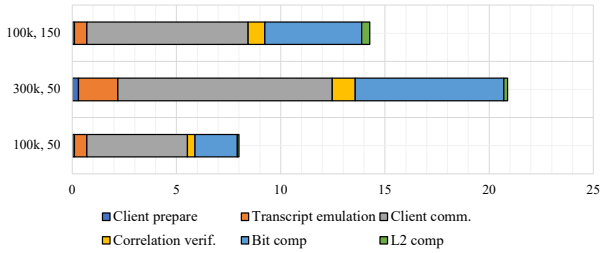


Figure 3: Runtime (sec) breakdown of ELSA with malicious privacy. The phases not shown in the figure have negligible costs. Vertical axis denotes parameter size, no. of clients.

Moderate bandwidth constraint. Clients fall in this case when they have enough bandwidth to assist servers by sending square correlations, and save on communication compared to regular clients by not sending OT correlations. For the rings we consider in our evaluation, this corresponds to 16x reduction in bandwidth requirement for constrained clients. Given such significant savings, most (if not all) constrained clients should be able to fit the requirements of this case. Fig. 4 shows that end-to-end runtime and server communication grow very slowly with increasing fraction of moderately bandwidth-constrained clients. When 10% are constrained, time and server communication increase by just 5% and 15%, respectively. Even in the hypothetical situation where half of the clients are constrained, server

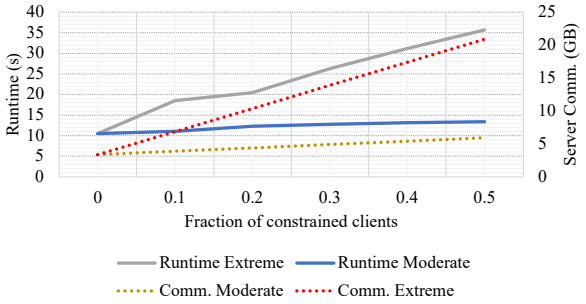


Figure 4: Effect of increasing fraction of bandwidth-constrained clients on performance of semi-honest private variant of ELSA. Extreme refers to the case of severely limited bandwidth where clients send no correlations, and in the moderate case, they only send square correlations.

communication increases by $<$ 2x and total runtime by $<$ 1.3x. This shows that our protocol preserves its efficiency guarantees under such circumstances.

Extreme bandwidth constraint. When clients are extremely bandwidth constrained, our protocol doesn't require them to send any correlations to the servers, resulting in a 137x reduction in bandwidth compared to regular clients for the rings we consider. From Fig. 4, we observe that the effect of increasing fraction of constrained clients on runtime and server communication is much more pronounced in this case. If 10% clients fall in this category, both end-to-end runtime and server communication increase by about 2x, and this rises to 3x for runtime and 6x for communication when half of the clients are constrained (highly unlikely).

Remark. Note that neither of our baselines (RoFL and Prio) work for bandwidth-constrained clients. Moreover, ELSA can use PCGs [30, 31] as another way to support bandwidth-constrained clients (see Section 3.2.1). We leave it to future work to implement this mode in our code. We estimate (through Ferret [112]) that our end-to-end runtime with Schoppmann et al.'s [94] PCG-based OTs will be comparable to our non-PCG approach.

5. Related Work

FL with single aggregator. This model has been adopted in a large class of privacy-preserving systems which are based on secret sharing [19, 26, 55, 70, 102, 110], homomorphic and functional encryption [111, 115], differential privacy (DP) [21, 57, 66], or a combination of these techniques [35, 62, 82, 107]. However, none of these works defend against malformed gradients. Zero-knowledge proofs [20, 23, 56] have been suggested [26, 100] to enforce norm defenses, but at unreasonably high overheads. Recent work RoFL [34] uses Bulletproofs [33] to enforce $\ell_2^\square$ and ℓ_∞ defense, and EIFFeL's [40] defenses use SNIPs (proofs in Prio [41]) where all the clients and the central aggregator do the verification collectively. Both of them, however, are quite inefficient. Moreover, other works that provide both privacy and defenses either assume unrealistic threat models [88], or are largely theoretical (and leak pairwise gradient distances) [101]. Tangential to the privacy-preserving FL solutions, Federated Averaging [83] was the first FL protocol which was improved in subsequent work [22, 37, 84, 104, 114] by adding defenses against malformed gradients.

FL with distributed trust. In this model, existing approaches include specialized systems for FL [61, 64] as well as systems for privacy-preserving collection of aggregate statistics like Prio [41] and Prio+ [13]. Prio uses specialized zero-knowledge proofs (SNIPs) to enforce arbitrary defenses against malformed gradients, and guarantees privacy against at most one malicious server (in the two server case). Rest of the works [13, 61, 64] only provide privacy against a semi-honest server, and therefore, leave much to be desired. Recently, Boneh et al. [27] proposed improvements to the proof size of Prio's SNIPs when the verification circuit has repeating substructures[20]. We estimate that the clients in ELSA are close to an order of magnitude more computationally efficient than their constructions. Moreover, ELSA can achieve significantly reduced client bandwidth (e.g., 64x) compared to [27] by using the PCG mode or our ideas for resource-constrained clients; their clients need to secret share individual bits of gradients as arithmetic shares to efficiently realize the $\ell_2^\square$ defense. Presently, their implementation is limited to languages much simpler than needed for FL, and the soundness of their most efficient proof (FLIOP with Fiat-Shamir) isn't well understood [32].

Other works. To provide some notion of gradient privacy, prior work has employed techniques like encoding gradients to higher dimensions [67] and Gaussian random projections [69]. [38] uses a pair of mixes with central aggregator, and defends against poisoning by clients. Rappor [50] and Privex [49] use DP, where the latter combines it with MPC in Tor to collect stats over anonymous communication.

6. Limitations and Future Work

Our techniques for defending against malformed gradients while achieving malicious privacy are only applicable when the defense operates independently on the gradients of each client. This excludes defenses like trimmed mean and median [114], Krum [22], and Bulyan [84]. Running such complex defenses inside secure computation (2PC) would be completely impractical, and as shown by Shejwalkar et al. [99], ℓ_2 defense performs as good against untargeted poisoning for production FL.

In this work, our main focus has been to protect privacy of individual gradients during aggregation. To limit what the global aggregate might leak, the honest server(s) can add differentially private (DP) noise to the global aggregate before opening it; in the vein of global DP. On the other hand, supporting local DP isn't as straightforward when clients are malicious and defenses are enforced.

ELSA cannot distinguish between a certain malicious server and a malicious client and thus we can't guarantee "fairness", i.e., every honest client's inputs will be used in the computation can't be guaranteed. If a malicious server frequently censors some clients, the honest server can detect that and take action.

We only guarantee malicious privacy in this work, and leave the exploration of malicious security (privacy with correctness) for future work. Efficiently achieving malicious security seems quite challenging given that standard techniques aren't compelling for the large number of parties in our system.

Lastly, as mentioned in Section 3.2.1, in future, we would like to explore the PCG mode in more detail and develop PCG-based OT extension protocols in the trusted dealer model that are specialized for our setting. As of now, some existing efficient constructions like Ferret [112] don't directly extend to this model.

20. This additionally requires a 2PC comparison since doing the entire ℓ_2 check inside the proof violates the repeating substructures property.

Acknowledgements

We thank the anonymous reviewers for their helpful feedback. We also thank students in the Sky security group for feedback that improved the presentation of this paper. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Alibaba, Amazon Web Services, Ant Group, Astronomer, Ericsson, Facebook, Futurewei, Google, IBM, Intel, Lacework, Microsoft, Nexla, Nvidia, Samsung, Scotiabank, Splunk, and VMware. This work is also supported by the Bakar Fellowship.

References

[1] https://www.tensorflow.org/federated/tutorials/tuning_recommended_aggregators.

[2] https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/.

[3] https://blog.cloudflare.com/oblivious-dns/.

[4] https://sepior.com/products/advanced-mpc-wallet.

[5] https://www.fireblocks.com/platforms/mpc-wallet/.

[6] [Online]. Available: https://github.com/tokio-rs/tokio

[7] "Emp toolkit," https://github.com/emp-toolkit.

[8] "Exposure notification privacy-preserving analytics (enpa) white paper," https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

[9] "Federated learning: Collaborative machine learning without centralized training data," https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[10] "libprio-rs," https://github.com/abetterinternet/libprio-rs.

[11] "PETs Prize Challenge: Phase 2 (Financial Crime)." [Online]. Available: https://www.drivendata.org/competitions/105/nist-federated-learning-2-financial-crime-federated/

[12] "PETs Prize Challenge: Phase 2 (Pandemic Forecasting)." [Online]. Available: https://www.drivendata.org/competitions/103/nist-federated-learning-2-pandemic-forecasting-federated/

[13] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou, "Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares," in *IACR ePrint Archive 2021/576*, 2021.

[14] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *CCS*, 2016.

[15] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More efficient oblivious transfer and extensions for faster secure computation," in *CCS*, 2013.

[16] ——, "More efficient oblivious transfer and extensions for faster secure computation," in *2013 ACM SIGSAC CCS'13*, 2013.

[17] G. Baruch, M. Baruch, and Y. Goldberg, "A little is enough: Circumventing defenses for distributed learning," in *NeurIPS*, 2019.

[18] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *CRYPTO*, 1991.

[19] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, "Secure single-server aggregation with (poly)logarithmic overhead," in *CCS*, 2020.

[20] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for C: verifying program executions succinctly and in zero knowledge," in *CRYPTO (2)*, 2013.

[21] A. Bhowmick, J. C. Duchi, J. Freudiger, G. Kapoor, and R. Rogers, "Protection against reconstruction and its applications in private federated learning," *CoRR*, 2018.

[22] P. Blanchard, E. M. E. Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *NeurIPS*, 2017.

[23] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications," in *Providing Sound Foundations for Cryptography*, 2019.

[24] F. Boenisch, A. Dziedzic, R. Schuster, A. S. Shamsabadi, I. Shumailov, and N. Papernot, "When the curious abandon honesty: Federated learning is not private," *CoRR*, 2021.

[25] K. A. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards federated learning at scale: System design," in *MLSys*, 2019.

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

[27] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-knowledge proofs on secret-shared data via fully linear pcps," in *CRYPTO (3)*, 2019.

[28] ——, "Lightweight techniques for private heavy hitters," in *42nd IEEE S&P*, 2021.

[29] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, "Function secret sharing for mixed-mode and fixed-point secure computation," in *EUROCRYPT 2021*, 2021.

[30] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round OT extension and silent non-interactive secure computation," in *CCS*, 2019.

[31] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient pseudorandom correlation generators: Silent OT extension and more," in *CRYPTO (3)*, 2019.

[32] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs," in *CCS*, 2019.

[33] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *IEEE S&P*, 2018.

[34] L. Burkhalter, H. Lycklama, A. Viand, N. Küchler, and A. Hithnawi, "Rofl: Attestable robustness for secure federated learning," *CoRR*, 2021.

[35] D. Byrd and A. Polychroniadou, "Differentially private secure multi-party computation for federated learning in financial applications," in *ICAIF*, 2020.

[36] S. Caldas, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, "LEAF: A benchmark for federated settings," *CoRR*, 2018.

[37] X. Cao, M. Fang, J. Liu, and N. Z. Gong, "Fltrust: Byzantine-robust federated learning via trust bootstrapping," in *NDSS*, 2021.

[38] R. Chen, I. E. Akkus, and P. Francis, "Splitx: high-performance private analytics," in *SIGCOMM*, 2013.

[39] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast large-scale honest-majority MPC for malicious adversaries," in *CRYPTO*, 2018.

[40] A. R. Chowdhury, C. Guo, S. Jha, and L. van der Maaten, "Eiffel: Ensuring integrity for federated learning," *CoRR*, 2021.

[41] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, robust, and scalable computation of aggregate statistics," in *NSDI*, 2017.

[42] H. Corrigan-Gibbs, D. Boneh, and D. Mazières, "Riposte: An anonymous messaging system handling millions of users," in *2015 IEEE S&P*, 2015.

[43] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPDZ2k: Efficient MPC mod 2^k for dishonest majority," in *CRYPTO*, 2018.

[44] I. Damgård, D. Escudero, T. K. Frederiksen, M. Keller, P. Scholl, and N. Volgushev, "New primitives for actively-secure MPC over rings with applications to private machine learning," in *IEEE S&P*, 2019.

[45] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO*, 2012.

[46] E. Dauterman, E. Feng, E. Luo, R. A. Popa, and I. Stoica, "DORY: an encrypted search system with distributed trust," in *USENIX ODSI*, 2020.

[47] E. Dauterman, M. Rathee, R. A. Popa, and I. Stoica, "Waldo: A private time-series database from function secret sharing," in *IEEE S&P*, 2022.

[48] D. Demmler, T. Schneider, and M. Zohner, "Aby-a framework for efficient mixed-protocol secure two-party computation." in *NDSS*, 2015.

[49] T. Elahi, G. Danezis, and I. Goldberg, "Privex: Private collection of traffic statistics for anonymous communication networks," in *CCS*, 2014.

[50] Ú. Erlingsson, V. Pihur, and A. Korolova, "RAPPOR: randomized aggregatable privacy-preserving ordinal response," in *CCS*, 2014.

[51] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for MPC over mixed arithmetic-binary circuits," in *CRYPTO (2)*, 2020.

[52] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh, "Express: Lowering the cost of metadata-hiding communication with cryptographic privacy," in *USENIX Security*, 2021.

[53] M. Fang, X. Cao, J. Jia, and N. Z. Gong, "Local model poisoning attacks to byzantine-robust federated learning," in *USENIX Security*, 2020.

[54] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO '86*, 1986.

[55] A. Fu, X. Zhang, N. Xiong, Y. Gao, and H. Wang, "VFL: A verifiable federated learning with privacy-preserving for big data in industrial iot," *CoRR*, 2020.

[56] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *EUROCRYPT*, 2013.

[57] R. C. Geyer, T. Klein, and M. Nabi, "Differentially private federated learning: A client level perspective," *CoRR*, 2017.

[58] O. Goldreich, S. Micali, and A. Wigderson, "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority," in *ACM STOC*, 1987.

[59] C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu, "Better concrete security for half-gates garbling (in the multi-instance setting)," in *CRYPTO (2)*, 2020.

[60] T. Gupta, N. Crooks, W. Mulhern, S. T. V. Setty, L. Alvisi, and M. Walfish, "Scalable and private media consumption with popcorn," in *13th USENIX NSDI*, 2016.

[61] M. Hao, H. Li, G. Xu, H. Chen, and T. Zhang, "Efficient, private and robust federated learning," in *ACSAC*, 2021.

[62] M. Hao, H. Li, G. Xu, S. Liu, and H. Yang, "Towards efficient and privacy-preserving federated deep learning," in *ICC*, 2019.

[63] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[64] L. He, S. P. Karimireddy, and M. Jaggi, "Secure byzantine-robust machine learning," *CoRR*, 2020.

[65] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, no. 8, 1997.

[66] Z. Huang, R. Hu, Y. Guo, E. Chan-Tin, and Y. Gong, "DP-ADMM: admm-based distributed learning with differential privacy," *IEEE Trans. Inf. Forensics Secur.*, 2020.

[67] M. Imani, Y. Kim, M. S. Riazi, J. Messerly, P. Liu, F. Koushanfar, and T. Rosing, "A framework for collaborative learning in secure high-dimensional space," in *CLOUD*, 2019.

[68] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," in *CRYPTO*, 2003.

[69] L. Jiang, R. Tan, X. Lou, and G. Lin, "On lightweight privacy-preserving collaborative learning for internet-of-things objects," in *IoTDI*, 2019.

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

[71] P. Kairouz, H. B. McMahan, B. Avent, A. Bellet, M. Bennis, A. N. Bhagoji, K. A. Bonawitz, Z. Charles, G. Cormode, R. Cummings, R. G. L. D'Oliveira, H. Eichner, S. E. Rouayheb, D. Evans, J. Gardner, Z. Garrett, A. Gascón, B. Ghazi, P. B. Gibbons, M. Gruteser, Z. Harchaoui, C. He, L. He, Z. Huo, B. Hutchinson, J. Hsu, M. Jaggi, T. Javidi, G. Joshi, M. Khodak, J. Konečný, A. Korolova, F. Koushanfar, S. Koyejo, T. Lepoint, Y. Liu, P. Mittal, M. Mohri, R. Nock, A. Özgür, R. Pagh, H. Qi, D. Ramage, R. Raskar, M. Raykova, D. Song, W. Song, S. U. Stich, Z. Sun, A. T. Suresh, F. Tramèr, P. Vepakomma, J. Wang, L. Xiong, Z. Xu, Q. Yang, F. X. Yu, H. Yu, and S. Zhao, "Advances and open problems in federated learning," *Found. Trends Mach. Learn.*, no. 1-2, 2021.

[72] M. Keller, E. Orsini, and P. Scholl, "Actively secure OT extension with optimal overhead," in *CRYPTO (1)*, 2015.

[73] ——, "MASCOT: faster malicious arithmetic secure computation with oblivious transfer," in *CCS*, 2016.

[74] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *EUROCRYPT (3)*, 2018.

[75] V. Kolesnikov and R. Kumaresan, "Improved OT extension for transferring short secrets," in *CRYPTO (2)*, 2013.

[76] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *CoRR*, 2016.

[77] A. Krizhevsky and G. H. et al., "Learning multiple layers of features from tiny images."

[78] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *IEEE S&P*, 2020.

[79] R. S. S. Kumar, M. Nyström, J. Lambert, A. Marshall, M. Goertzel, A. Comissoneru, M. Swann, and S. Xia, "Adversarial machine learning-industry perspectives," in *SP Workshops*, 2020.

[80] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, no. 4, 1989.

[81] Y. Lindell, "How to simulate it - A tutorial on the simulation proof technique," *IACR ePrint Archive 2016/046*, 2016.

[82] Y. Liu, Z. Ma, X. Liu, S. Ma, S. Nepal, R. H. Deng, and K. Ren, "Boosting privately: Federated extreme gradient boosting for mobile crowdsensing," in *ICDCS*, 2020.

[83] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *AISTATS*, 2017.

[84] E. M. E. Mhamdi, R. Guerraoui, and S. Rouault, "The hidden vulnerability of distributed learning in byzantium," in *ICML*, 2018.

[85] P. Mohassel and P. Rindal, "Aby^3: A mixed protocol framework for machine learning," in *CCS*, 2018.

[86] M. Naor and B. Pinkas, "Efficient oblivious transfer protocols," in *SODA*, 2001.

[87] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra, "A new approach to practical active-secure two-party computation," in *CRYPTO*, 2012.

[88] K. Pillutla, S. M. Kakade, and Z. Harchaoui, "Robust aggregation for federated learning," *IEEE Transactions on Signal Processing*, pp. 1142–1154, 2022.

[89] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein, "Senate: A maliciously-secure MPC platform for collaborative analytics," in *USENIX Security*, 2021.

[90] D. Rathee, M. Rathee, R. K. K. Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi, "SIRNN: A Math Library for Secure RNN Inference," in *IEEE S&P*, 2021.

[91] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *CCS*, 2020.

[92] D. Rathee, T. Schneider, and K. K. Shukla, "Improved multiplication triple generation over rings via rlwe-based AHE," in *CANS*, 2019.

[93] D. Rotaru and T. Wood, "Marbled circuits: Mixing arithmetic and boolean circuits with active security," in *INDOCRYPT 2019*, 2019.

[94] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova, "Distributed vector-ole: improved constructions and implementation," in *CCS*, 2019.

[95] A. Shamir, "How to share a secret," *Communications of the ACM*, no. 11, 1979.

[96] V. Shejwalkar and A. Houmansadr, "Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning," in *NDSS*, 2021.

[97] ——, "Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning," in *NDSS*, 2021.

[98] ——, "Manipulating the byzantine: Optimizing model poisoning attacks and defenses for federated learning," in *NDSS*, 2021.

[99] V. Shejwalkar, A. Houmansadr, P. Kairouz, and D. Ramage, "Back to the drawing board: A critical evaluation of poisoning attacks on federated learning," *CoRR*, 2021.

[100] E. Shi, T. H. Chan, E. G. Rieffel, R. Chow, and D. Song, "Privacy-preserving aggregation of time-series data," in *NDSS*, 2011.

[101] J. So, B. Güler, and A. S. Avestimehr, "Byzantine-resilient secure federated learning," *IEEE J. Sel. Areas Commun.*, no. 7, 2021.

[102] ——, "Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning," *IEEE J. Sel. Areas Inf. Theory*, no. 1, 2021.

[103] W. D. Stangl, "Counting squares in n," *Mathematics Magazine*, vol. 69, no. 4, 1996.

[104] Z. Sun, P. Kairouz, A. T. Suresh, and H. B. McMahan, "Can you really backdoor federated learning?" *CoRR*, 2019.

[105] A. T. Suresh, F. X. Yu, S. Kumar, and H. B. McMahan, "Distributed mean estimation with limited communication," in *ICML*, 2017.

[106] J. Thaler, "Proofs, arguments, and zero-knowledge," 2022.

[107] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou, "A hybrid approach to privacy-preserving federated learning," in *AISec@CCS*, 2019.

[108] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia, "Splinter: Practical private queries on public data," in *NSDI*, 2017.

[109] H. Wang, K. Sreenivasan, S. Rajput, H. Vishwakarma, S. Agarwal, J. Sohn, K. Lee, and D. S. Papailiopoulos, "Attack of the tails: Yes, you really can backdoor federated learning," in *NeurIPS*, 2020.

[110] G. Xu, H. Li, S. Liu, K. Yang, and X. Lin, "Verifynet: Secure and verifiable federated learning," *IEEE Trans. Inf. Forensics Secur.*, 2020.

[111] R. Xu, N. Baracaldo, Y. Zhou, A. Anwar, and H. Ludwig, "Hybridalpha: An efficient approach for privacy-preserving federated learning," in *AISec@CCS*, 2019.

[112] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast extension for correlated OT with small communication," in *CCS*, 2020.

[113] A. C. Yao, "How to generate and exchange secrets (extended abstract)," in *27th Annual Symposium on Foundations of Computer Science*, 1986.

[114] D. Yin, Y. Chen, K. Ramchandran, and P. L. Bartlett, "Byzantine-robust distributed learning: Towards optimal statistical rates," in *ICML*, 2018.

[115] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "Batchcrypt: Efficient homomorphic encryption for cross-silo federated learning," in *USENIX ATC*, 2020.

Algorithm 6 AND of Boolean Shares Π_{AND}

Input: Bit shares of secrets $x = x^{(0)} \oplus x^{(1)}, y = y^{(0)} \oplus y^{(1)} \in \mathbb{Z}_2$. S_0 has additional inputs $\Delta, q, q' \in \mathbb{F}_{2^\lambda}$ and S_1 has additional inputs $t, t' \in \mathbb{F}_{2^\lambda}, r, r' \in \{0,1\}$. Let $H : [\ell] \times \mathbb{F}_{2^\lambda} \to \mathbb{Z}_{2^\lambda}$ be a hash function [72] in the random-oracle model.

Output: Output $z^{(b)} \in \mathbb{Z}_2$ such that

$$z^{(0)} \oplus z^{(1)} = x \wedge y$$

Between Servers (Assuming S_0 is OTSn*)*

1: S_0 computes $m_0 \leftarrow H(c||q)$ and $m_1 \leftarrow H(c||q+\Delta)$, where c is a global counter. Similarly, $m'_0 \leftarrow H(c||q')$ and $m'_1 \leftarrow H(c||q'+\Delta)$.
S_1 computes $m_r \leftarrow H(c||t)$ and $m'_r \leftarrow H(c||t')$.
2: S_0 sets $s^{(0)} \leftarrow \Pi^1_{\mathsf{BitMultUA}}(y^{(0)}, (m_0, m_1))$.
S_1 sets $s^{(1)} \leftarrow \Pi^1_{\mathsf{BitMultUA}}(x^{(1)}, (r, m_r))$.
3: S_0 sets $t^{(0)} \leftarrow \Pi^1_{\mathsf{BitMultUA}}(x^{(0)}, (m'_0, m'_1))$.
S_1 sets $t^{(1)} \leftarrow \Pi^1_{\mathsf{BitMultUA}}(y^{(1)}, (r', m'_r))$.
4: Set $z^{(b)} \leftarrow (x^{(b)} \wedge y^{(b)}) \oplus s^{(b)} \oplus t^{(b)}$.

Appendix A. Semi-Honest Server

We begin by first proving some results about correctness of our sub-protocols and verification phases.

A.1. Correctness of Sub-Protocols

Correctness of $\Pi_{\mathsf{BitMultUA}}$. (Algo. 1)
Case $x^{(1)} = 0$:
When $j = 0$,

$$\begin{aligned} y^{(0)} + y^{(1)} &= r + v_0 - m_0 \\ &= r + (m_0 - r) - m_0 \\ &= 0 = x^{(0)} \wedge x^{(1)} \end{aligned}$$

When $j = 1$,

$$\begin{aligned} y^{(0)} + y^{(1)} &= r + v_0 - m_1 \\ &= r + (m_1 - r) - m_1 \\ &= 0 = x^{(0)} \wedge x^{(1)} \end{aligned}$$

Case $x^{(1)} = 1$:

When $j = 0$,

$$\begin{aligned} y^{(0)} + y^{(1)} &= r + v_1 - m_0 \\ &= r + (m_0 - r + x^{(0)}) - m_0 \\ &= 1 = x^{(0)} \wedge x^{(1)} \end{aligned}$$

When $j = 1$,

$$\begin{aligned} y^{(0)} + y^{(1)} &= r + v_1 - m_1 \\ &= r + (m_1 - r + x^{(0)}) - m_1 \\ &= 1 = x^{(0)} \wedge x^{(1)} \end{aligned}$$

Correctness of Π_{AND}. (Algo. 6) $z = x \wedge y$ can be written in terms of shares of x and y as:

$$= (x^{(0)} \oplus x^{(1)}) \wedge (y^{(0)} \oplus y^{(1)})$$
$$= (x^{(0)} \wedge y^{(0)}) \oplus (x^{(1)} \wedge y^{(1)}) \oplus (x^{(0)} \wedge y^{(1)}) \oplus (x^{(1)} \wedge y^{(0)})$$

Shares of the first two terms can be computed locally by the servers given that S_b knows $x^{(b)}, y^{(b)}$. For the last two terms (cross-terms), $\Pi_{\mathsf{BitMultUA}}$ is invoked with random OTs (generated by hashing Δ-COTs in step 1) as input. Correctness follows from $\Pi_{\mathsf{BitMultUA}}$.

Correctness of Π^j_{BitMult}. (Algo. 2) If $x^{(1)} = 0$, then

$$\begin{aligned} y^{(0)} + y^{(1)} &= -H(c||q) + H(c||t) && \pmod{2^j} \\ &= -H(c||t + x^{(1)}\Delta) + H(c||t) && \pmod{2^j} \\ &= -H(c||t) + H(c||t) && \pmod{2^j} \\ &= 0 = x^{(0)} \wedge x^{(1)} \end{aligned}$$

On the contrary, if $x^{(1)} = 1$, then

$$\begin{aligned} y^{(0)} + y^{(1)} &= -v_0 + v_0 + v_1 + x^{(0)} - H(c||t) && \pmod{2^j} \\ &= H(c||q + \Delta) + x^{(0)} - H(c||t) && \pmod{2^j} \\ &= H(c||t) + x^{(0)} - H(c||t) && \pmod{2^j} \\ &= x^{(0)} \wedge x^{(1)} \end{aligned}$$

Where we have used the fact that the OT correlations have $x^{(1)}$ as the choice bit.

Correctness of $\Pi^{w,\ell}_{\mathsf{BitComp}}$. (Algo. 4) This follows from the correctness of Π_{BitMult}. For each index $i \in \{0, 1, \cdots, w-1\}$, Step 4 of Algo. 4 computes $y^{(0)}, y^{(1)}$ such that:

$$y^{(0)} + y^{(1)} = x_i^{(0)} \wedge x_i^{(1)} \pmod{2^{\ell'}} \tag{4}$$

Thus, Step 5 of Algo. 4 increments the z shares by 2^i times $x^{(0)}, x^{(1)}$ and $-2x^{(0)} \wedge x^{(1)}$ (values modulo $2^{\ell'}$). Since $2^{i+1} \cdot 2^{\ell'} = 2^{\ell}$, this allows us to optimize our calls to Π_{BitMult} by only calling it for ℓ' bits instead of ℓ bits because in the latter, the upper $i+1$ bits would anyways have been removed by the mod L operation. Finally, using this relation between XOR operation and additions/subtractions $x^{(0)} \oplus x^{(1)} = x^{(0)} + x^{(1)} - 2x^{(0)} \wedge x^{(1)}$, we have that z is the correct bit composition modulo 2^{ℓ}.

Correctness of ℓ_2 computation. (Algo. 5) Given a square correlation (a, d) with shares $(a^{(b)}, d^{(b)})$ for $b \in \{0, 1\}$, the shares of the square y of an input value x with shares $x^{(b)}$ can be computed as:

$$\begin{aligned} y^{(b)} &= d^{(b)} + 2ex^{(b)} - b \cdot e^2 \quad \text{where } e \leftarrow x - a \\ &= d^{(b)} + 2xx^{(b)} - 2ax^{(b)} - b \cdot (x^2 + a^2 - 2ax) \end{aligned}$$

Since d is a^2 (assuming correlations are correct), $y = y^{(0)} + y^{(1)} = d + 2x^2 - 2ax - x^2 - a^2 + 2ax = x^2$. Hence, $y^{(b)}$ are shares of x^2.

OT verification. We use clients as untrusted sources of COT_Δ, i.e., OTs where the OT sender gets $(m, \Delta) \in \mathbb{F}_{2^\lambda} \times \mathbb{F}_{2^\lambda}$ and OT receiver gets $(b, m + b \cdot \Delta) \in \{0, 1\} \times \mathbb{F}_{2^\lambda}$, where $+$ is addition in $\mathbb{F}_{2^\lambda}$ which is the same as addition in $\mathbb{F}_2^\lambda$ (i.e. bitwise XOR), and $\cdot$ is multiplication in finite field.

Lemma 1. *If the OT verification in Algo. 5 succeeds, the COT_Δ correlations sent by the client are correct except with probability $2^{-\lambda}$.*

Proof. The random coefficients $\{\chi_0, \ldots, \chi_{n-1}\} \in \mathbb{F}^n_{2^\lambda}$ are sampled uniformly after all the OT correlations have been received. Therefore, the correlations were constructed by the client without the knowledge of $\{\chi_i\}_i$. The check over $\mathbb{F}_{2^\lambda}$ performed by S_0 is (e_j is error injected in Q_j by the client):

$$\tilde{t} = \tilde{q} + \tilde{x} \cdot \Delta = \sum_{j=0}^{n-1} Q_j \cdot \chi_j + \hat{x}_j \cdot \chi_j \cdot \Delta = \tilde{t} + \sum_{j=0}^{n-1} e_j \cdot \chi_j$$

For this check to pass, $\sum_{j=0}^{n-1} e_j \cdot \chi_j$ has to be 0 in $\mathbb{F}_{2^\lambda}$. In the case of incorrect OT correlations, $\exists$ at least one index i such that $e_i \neq 0$. Since $e_i \cdot \chi_i$ is uniformly random in $\mathbb{F}_{2^\lambda}$ (because for any two field elements u, v, if $e_i \cdot u = e_i \cdot v$, then $u = v$), $\sum_{j=0}^{n-1} e_j \cdot \chi_j$ is also uniformly random, and therefore, is zero with probability $2^{-\lambda}$ when OTs are incorrect. □

Square correlation verification.

Lemma 2. *If the square correlation verification in Algo. 5 succeeds, the square correlations selected by the servers are correct modulo 2^u except with probability $2^{-\kappa}$.*

Proof. Let us consider a pair of (potentially erroneous) correlations $(a, d), (\hat{a}, \hat{d})$, where $d \leftarrow a^2 + \delta$ and $\hat{d} \leftarrow \hat{a}^2 + \delta'$. During the check, servers sacrifice the secondary correlation $(\hat{a}, \hat{d})$ to validate the primary one (a, d). To validate the correlations, the servers check if the following is zero: $t^2 d - \hat{d} - 2tea + e^2$ where $e \leftarrow ta - \hat{a}$ and t is a random odd element. Replacing for $e, d, \hat{d}$, we get $t^2 d - \hat{d} - 2tea + e^2 = t^2\delta - \delta'$.

If the primary correlation is incorrect, i.e., $\delta \neq 0 \bmod 2^u$, then we sketch an argument similar to SPD$\mathbb{Z}_{2^k}$ [43]. For servers to pass the check, we require that $t^2\delta \equiv \delta' \bmod 2^v$. Let g be the largest power of two which divides δ. We know that $0 < g < u$ given that $\delta \neq 0 \bmod 2^u$. Since the lower g bits of $t^2\delta$ are zeros, if the check passes, then the upper $v - g$ bits of δ, δ' follow $t^2 \equiv \frac{\delta'}{2^g}\frac{\delta}{2^g}^{-1} \bmod 2^{v-g}$, and therefore, it would mean that the client guessed $v - g$ bits of t^2. Since t is a randomly sampled odd element in $\mathbb{Z}_{2^v}$ (i.e., it is a unit [103]), the distribution of its quadratic residues $t^2 \bmod$

2^{v-g} is uniform and takes 2^{v-g-3} values [103]. Given that we set $v \leftarrow u + \kappa + 3$, passing the check means that the client would have guessed $> \kappa$ random bits. The probability of this happening is at most $2^{-\kappa}$. □

A.2. Correctness and Privacy

We prove security (correctness and privacy) of our protocol in the simulation paradigm of MPC [81] in the $(\mathcal{F}_{\mathsf{CoinFlip}}, \mathcal{F}_{\mathsf{RO}})$-hybrid model [81]. $\mathcal{F}_{\mathsf{CoinFlip}}$ is invoked by a pair of parties and returns the same set of freshly sampled random coins to both. In practice, this can be realized by using a simple commit-and-open sub-protocol [72]. We work in the random oracle model where the hash function H used in our protocol is modeled as a random oracle. The functionality $\mathcal{F}_{\mathsf{RO}}$ facilitates this by returning the output of a randomly chosen function on requested inputs, and in practice, SHA-2 or AES [72] can be used.

Ideal Functionality $\mathcal{F}$. We define a stateful iterative ideal functionality for FL. In each round:

- $\mathcal{F}$ receives gradient updates from clients selected in the current round.
- It enforces ℓ_2 and $\ell_\sqcap$ bounds (by checking the number of received bits) on each submission. All the non-complying submissions are rejected.
- Outputs the aggregate of surviving gradients. If fewer than τ fraction of clients survive, output $\perp$.

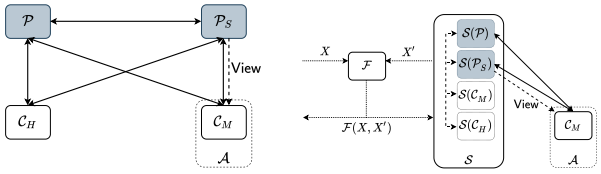
Theorem A.1. *For every non-uniform probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ controlling a set of malicious clients and having access to the view of at most one (semi-honest) server, there exists a non-uniform PPT adversary $\mathcal{S}$ in the ideal world which only interacts with the ideal functionality $\mathcal{F}$ such that the distributions*

$$\left\{\mathsf{Ideal}_{\mathcal{F},\mathcal{S}(z)}(\{X_i\}_{i=1}^{|\mathcal{C}|}, \lambda, \kappa)\right\}_{\{X_i\}_{i=1}^{|\mathcal{C}|}, z, \lambda, \kappa}$$

$$\left\{\mathsf{Real}_{\Pi,\mathcal{A}(z)}(\{X_i\}_{i=1}^{|\mathcal{C}|}, \lambda, \kappa)\right\}_{\{X_i\}_{i=1}^{|\mathcal{C}|}, z, \lambda, \kappa}$$

are indistinguishable except with probability $O\big(|\mathcal{C}_M| \cdot (2^{-\lambda} + 2^{-\kappa})\big)$ in the $(\mathcal{F}_{\mathsf{CoinFlip}}, \mathcal{F}_{\mathsf{RO}})$-hybrid model, where $\mathcal{C}$ is the set of all clients with any inputs $\{X_i\}_{i=1}^{|\mathcal{C}|}$ such that $\forall(i,j)$, $|X_i| = |X_j|$, $z \in \{0,1\}^*$ is an auxiliary input by the adversary to capture malicious strategy, $\mathcal{C}_M$ is the set of malicious clients, and λ, κ are computational and statistical security parameters, resp. $\mathsf{Ideal}_{\mathcal{F},\mathcal{S}(z)}(\{X_i\}, \lambda, \kappa), \mathsf{Real}_{\Pi,\mathcal{A}(z)}(\{X_i\}, \lambda, \kappa)$ denote the output pairs of honest parties and the adversary in ideal and real world, respectively, on protocol inputs $\{X_i\}$ and auxiliary input z.

Proof. To construct a simulator $\mathcal{S}$ in the ideal world which can produce a distribution similar to the one in real world, we give $\mathcal{S}$ black-box access to $\mathcal{A}$, where it "simulates" an interaction with $\mathcal{A}$ that looks like the real world, essentially leading $\mathcal{A}$ to produce the same output as it would when participating in the real protocol execution. $\mathcal{S}$ can then simply output whatever $\mathcal{A}$ outputs.



(a) Real world interaction. (b) Ideal world interaction.

Figure 5: $\mathcal{P}$ and $\mathcal{C}_H$ denote honest server and honest clients, respectively. $\mathcal{P}_S$ and $\mathcal{C}_M$ denote semi-honest server and malicious client, respectively. Parties fully under adversary's control, i.e. $\mathcal{C}_M$, are shown with dashed outline. The simulator $\mathcal{S}$ internally maintains some state for each party shown by $\mathcal{S}(\cdot)$. Shaded parties run server protocol. Solid arrows denote protocol interaction, dashed arrows denote transfer of information from semi-honest server to $\mathcal{A}$ and interactions within the simulator, and dotted arrows denote ideal world interactions. X, X' denote honest and malicious clients' inputs, respectively.

The real and ideal world distributions that we want to show indistinguishability for can be boiled down to showing that the joint distribution of 1) the output of $\mathcal{F}$, and 2) the view (all internal state, and messages sent and received) of the semi-honest corrupt server, say $\mathcal{P}_S$, in the real world and its simulated counterpart in the ideal world, are indistinguishable. This is because the output of $\mathcal{F}$ captures the output of honest parties, and the output of $\mathcal{A}$ is only additionally influenced by the view of $\mathcal{P}_S$. Since clients don't participate in any intermediate stage in the protocol, their view (except their output) is pre-determined by $\mathcal{A}$ and is indistinguishable in the real and ideal worlds. Figures 5a and 5b show the setup. We first describe our simulator construction in the $(\mathcal{F}_{\mathsf{CoinFlip}}, \mathcal{F}_{\mathsf{RO}})$-hybrid model [81], followed by proving indistinguishability using the hybrid argument.

Simulator $\mathcal{S}$. $\mathcal{S}$ starts with a black-box access to $\mathcal{A}$ and given that we work in the $(\mathcal{F}_{\mathsf{CoinFlip}}, \mathcal{F}_{\mathsf{RO}})$-hybrid model, $\mathcal{S}$ provides access of both of these functionalities to $\mathcal{A}$. It then proceeds as follows, where it keeps sending the view of $\mathcal{P}_S$ that it locally generates to $\mathcal{A}$:

Input Sharing, OT and Square Correlation Generation Phases

1) Assuming that the gradient inputs of all honest clients ($\mathcal{C}_H$) are *zero vectors*, $\mathcal{S}$ generates input shares, and OT and square correlation shares as dictated by our protocol to simulate the view of $\mathcal{P}_S$.
2) $\mathcal{A}$ sends shares of input, and OT and square correlations of the malicious clients corresponding to $\mathcal{P}_S$ and $\mathcal{P}$ (server other than $\mathcal{P}_S$), and $\mathcal{S}$ receives them. If shares of a malicious client are missing, or if $\mathcal{A}$ aborts early, proceed by ignoring those clients' participation.

OT and Square Correlation Verification, Bit Composition, ℓ_2 Computation, and ℓ_2 Enforcement Phases

1) $\mathcal{S}$ follows our protocol description to generate the view of $\mathcal{P}_S$. Invocations to the hash function H and sampling of common random values are facilitated by $\mathcal{F}_{\mathsf{RO}}$ and $\mathcal{F}_{\mathsf{CoinFlip}}$, respectively, through $\mathcal{S}$.

Aggregation Phase

1) Excluding the clients in $\mathcal{C}_M$ who failed either the OT verification, the square correlation verification, or the ℓ_2 enforcement in generating $\mathcal{P}_S$'s view in the previous steps, $\mathcal{S}$ reconstructs the gradient inputs of malicious clients from the shares it initially received from $\mathcal{A}$.
2) These inputs are then sent to $\mathcal{F}$ and $\mathcal{S}$ receives the final aggregate a in response.
3) It then extracts the sum a_H of honest clients' gradients from a by subtracting the sum a_M of gradients of malicious clients which it already knows. To compute a_M, for each gradient of a malicious client that $\mathcal{S}$ sends to $\mathcal{F}$, it computes the ℓ_2 value as $\mathcal{F}$ would do, removes the clients who violate the current bound (both ℓ_2 and $\ell_\sqcap$), and takes the sum of the remaining gradients.
4) Follows our protocol description to generate the final message in the view of $\mathcal{P}_S$, but adds a_H to it to correct for the initial assumption that honest clients' inputs are zero vectors.
5) $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Remark. One could also prove security of our protocol by putting $\mathcal{P}_S$ inside $\mathcal{A}$ and having $\mathcal{S}$ extract gradient inputs of malicious clients through $\mathcal{F}_{\mathsf{RO}}$ calls made by $\mathcal{P}_S$. However, as we consider in our proof, the semi-honest nature of $\mathcal{P}_S$ means $\mathcal{S}$ can run its tape for $\mathcal{A}$, and therefore, inputs can be more straightforwardly extracted.

We now proceed by defining a sequence of hybrids to show indistinguishability of the distributions in real and ideal worlds. Note that there are eight messages that constitute server's view (server receives these eight messages) during protocol execution:

1) Shares of gradient vector (step 2), square correlations (step 3) and OT correlations (step 3).
2) Server participating as OTRc sends $\tilde{x}, \tilde{t}$ to OTSn during OT verification (step 2).
3) Opening e (step 4) and checking shares of zero (step 6) during square correlation verification.
4) Server participating as OTSn sends u to OTRc in Π_{BitMult} during bit composition (step 2).
5) Opening e (step 3) during ℓ_2 computation phase and opening the sign bit (step 3) during ℓ_2 enforcement.
6) Messages (v_0, v_1) sent by OTSn in the two calls to $\Pi_{\mathsf{BitMultUA}}$ made by Π_{AND} in ℓ_2 enforcement (step 2).
7) Bit messages d sent by OTRc in the two calls to $\Pi_{\mathsf{BitMultUA}}$ made by Π_{AND} in ℓ_2 enforcement (step 2).
8) Final opening message in the aggregation phase (step 2).

Since our protocol is asymmetric in the roles of S_0, S_1, i.e., it assumes that S_0 is the OTSn for OT verification, Π_{BitComp} and Π_{AND}, for the security argument, we will consider both the cases: when $\mathcal{P}_S$ is P_0 and when it is P_1.

Case 1: $\mathcal{P}_S$ is S_0, i.e., OTSn

Hybrid 0 $\mathcal{H}_0$. We start with the real world as our initial hybrid.

Hybrid 1 $\mathcal{H}_1$. We have honest clients send $\mathcal{P}_S$'s share $x^{(0)}$ of the gradient vector along with Δ directly to $\mathcal{P}$. $\mathcal{P}$ (i.e., S_1) assumes that the gradient vector for all honest clients is a zero vector and samples fresh shares for each keeping the share of $\mathcal{P}_S$ same as $x^{(0)}$, i.e., the shares will be $(x^{(0)}, x^{(1)\prime})$ such that they reconstruct to a zero vector. Now during the OT verification phase of the protocol execution, $\mathcal{P}$ uses Δ and T_j to compute T'_j such that it is consistent with the choice bits in $x^{(1)\prime}$ (recall that T_j was only consistent with $x^{(1)}$); the indices where $x^{(1)\prime}$ and $x^{(1)}$ differ, Δ can be either added or subtracted from T_j to yield T'_j. It then computes $\tilde{x}$ (resp. $\tilde{t}$) by using $x^{(1)\prime}$ (resp. T'_j) instead of $x^{(1)}$ (resp. T_j) and the rest of the protocol proceeds unchanged. The only change in the view of $\mathcal{P}_S$ happens in the OT verification message $(\tilde{x}, \tilde{t})$ it receives from $\mathcal{P}$. Indistinguishability (negligible in κ) between $\mathcal{H}_1$ and $\mathcal{H}_0$ follows from the fact that additional $\lambda + \kappa$ OTs are used in verification and discarded later; formally proved by Keller et al. [72]. These OTs serve the purpose of hiding information about the choice bits of the remaining OTs from OTSn.

Hybrid 2 $\mathcal{H}_2$. In this hybrid, $\mathcal{P}$ replaces $x^{(1)}$ with $x^{(1)\prime}$ at all steps in the protocol execution. This means that the protocol is now essentially treating the inputs of all honest clients as zero vectors. This would normally lead to an incorrect output in the end, but we have $\mathcal{P}$ *correct* for this in the final message it sends to $\mathcal{P}_S$ during aggregation phase. To perform the correction, it first uses $x^{(0)}$ (from $\mathcal{H}_1$) and $x^{(1)}$ to locally reconstruct the original gradient vectors x of honest clients, and x can then be added back into the final message before sending it. $\mathcal{P}_S$'s view remains the same as the previous hybrid. Messages in ℓ_2 computation are masked by randomness from square correlations, and in ℓ_2 enforcement, messages to $\mathcal{P}_S$ in $\Pi_{\mathsf{BitMultUA}}$ are also masked by the random choice bits of OT correlations, while the sign bit opens to 1 (same as clipped honest gradients in the previous hybrid) because zero vectors always follow norm bound. The final message in aggregation phase reconstructs to the same output as $\mathcal{H}_1$. In all other phases, $\mathcal{P}_S$ doesn't receive any message.

Hybrid 3 $\mathcal{H}_3$. This is the ideal world. The only difference in $\mathcal{P}_S$'s view from the previous hybrid comes from the soundness error of OT and square correlation verification. In $\mathcal{H}_2$, the final aggregate is computed based on the OT and square correlations submitted by the clients, while in this hybrid, $\mathcal{F}$ computes the final output (doesn't use client-submitted correlations). Therefore, appealing to Lemmas 1 and 2, the two hybrids only differ in $\mathcal{A}$'s view when *either* of our verification phases are fooled leading to the use of incorrect correlations, which happens with probability at most $|\mathcal{C}_M| \cdot (2^{-\lambda} + 2^{-\kappa})$.

Case 2: $\mathcal{P}_S$ is S_1, i.e., OTRc

Hybrid 0 $\mathcal{H}_0$. We start with the real world as our initial hybrid.

Hybrid 1 $\mathcal{H}_1$. We have honest clients send $\mathcal{P}_S$'s share $x^{(1)}$ of the gradient vector to $\mathcal{P}$. $\mathcal{P}$ (i.e., S_0) assumes that the gradient vector for all honest clients is zero, and samples

new shares of these zero vectors keeping one share same as $x^{(1)}$, i.e., the shares will be $(x^{(0)\prime}, x^{(1)})$ which reconstruct to zero. Now, $\mathcal{P}$ replaces $x^{(0)}$ with $x^{(0)\prime}$ at all steps in the protocol. Similar to the previous case, this translates to using zero vectors as inputs of all honest clients, which are not their original inputs, so a correction needs to be made to keep the view of $\mathcal{P}_S$ indistinguishable compared to $\mathcal{H}_0$. This correction can be made by $\mathcal{P}$ by first reconstructing the original inputs x of honest clients (from $x^{(0)}, x^{(1)}$), and then adding x back into the final message it sends to $\mathcal{P}_S$ during the aggregation phase. Since we work in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model, the view of $\mathcal{P}_S$ is identically distributed in this hybrid compared to the previous because all intermediate messages (where $x^{(0)\prime}$ was newly introduced) received by $\mathcal{P}_S$ are masked by uniformly random values, sign bit opens to 1, and the final message (aggregation phase) reconstructs to the same output as the previous hybrid.

Hybrid 2 $\mathcal{H}_2$**.** This is the ideal world. Similar to the previous case, $\mathcal{A}$'s view differs from the previous hybrid only when the final aggregate from the protocol in $\mathcal{H}_1$ doesn't match $\mathcal{F}$'s output. This happens when either malformed OTs or square correlations fool our verification phases. The probability of such an event is bounded by $|\mathcal{C}_M| \cdot (2^{-\lambda} + 2^{-\kappa})$. □

Appendix B. Malicious Server

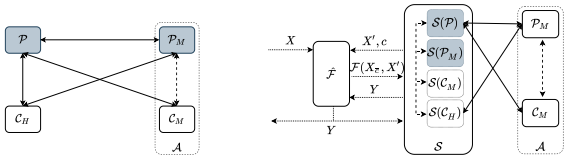
We now shift our focus to proving the guarantees of our protocol against malicious corruption of at most one server. We guarantee privacy of individual gradients in such a situation, but not correctness of the output. We first formally define a more general notion of malicious privacy than previously considered [14] followed by the ideal functionality. At a high-level, malicious privacy is defined similar to malicious security, but with a modified ideal functionality that is *corruptible*, meaning that the adversary can control what it outputs.

Definition 1. *Let* $f : (\{0,1\}^*)^p \to (\{0,1\}^*)^p$ *be a* p*-party functionality, and* $\hat{f}$ *be its* corruptible *counterpart which sends the evaluation of* f *on the inputs of parties to the adversary and, based on that, allows it to specify the final output. Let* Π *be an* n*-party protocol* ($n \geqslant p$) *with* p *primary parties (with inputs) and* $n - p$ *helpers that correctly computes* f*.* Π (t, s)*-privately realizes* f *in the presence of static malicious adversaries if for every non-uniform probabilistic polynomial-time (PPT) adversary* $\mathcal{A}$ *controlling at most* t *primary parties and* s *helpers in the real world, there exists a non-uniform PPT adversary* $\mathcal{S}$ *in the ideal world which only interacts with* $\hat{f}$*, such that the distributions*

$$\left\{\mathsf{Ideal}_{\hat{f},\mathcal{S}(z)}(\{X_i\}_{i=1}^p, \lambda, \kappa)\right\}_{\{X_i\}_{i=1}^p, z, \lambda, \kappa}$$

$$\left\{\mathsf{Real}_{\Pi,\mathcal{A}(z)}(\{X_i\}_{i=1}^p, \lambda, \kappa)\right\}_{\{X_i\}_{i=1}^p, z, \lambda, \kappa}$$

are computationally indistinguishable, where variables are defined as in Theorem A.1.



(a) Real world interaction. (b) Ideal world interaction.

Figure 6: Most of the details same as the caption of Fig. 5. $\mathcal{P}_M$ denotes malicious server component of the adversary $\mathcal{A}$. Dashed arrows denote transfer of information within $\mathcal{A}$ and $\mathcal{S}$. $\hat{\mathcal{F}}$ denotes corruptible ideal functionality corresponding to $\mathcal{F}$. c is the censor list, $X_{\bar{c}}$ are the inputs of clients outside the censor list, and Y is the final corrupted output of $\hat{\mathcal{F}}$.

Ideal Functionality $\mathcal{F}'$ **and corruptible** $\hat{\mathcal{F}}$**.** $\mathcal{F}'$ is similar to $\mathcal{F}$ from the proof of Theorem A.1 with a difference that $\mathcal{A}$ specifies a censor list which includes clients whose inputs won't be considered in the computation. $\mathcal{F}'$ receives the list and discards the inputs of clients present in it. The censor list captures the ability of a malicious server to falsely report the inputs of some clients (malicious and honest) as malformed or incomplete. $\hat{\mathcal{F}}$ is the corruptible counterpart of $\mathcal{F}'$ (Definition 1). For a pictorial representation of $\hat{\mathcal{F}}$, refer to Fig. 6b

Theorem B.1. *Our* $(|\mathcal{C}| + 2)$*-party protocol* $(|\mathcal{C}| - 1, 1)$*-privately computes the* $|\mathcal{C}|$*-party functionality* $\mathcal{F}'$ *in the* $(\mathcal{F}_{\mathsf{CoinFlip}}, \mathcal{F}_{\mathsf{RO}})$*-hybrid model in the presence of static malicious adversaries.*

Proof. Similar to the case of semi-honest server, we will use the simulation paradigm to prove this theorem. We assume that clients send entire transcript to the servers for simplicity. The optimized case of using transcript digests follows similarly, and we briefly discuss this at the end of the proof. Setup is presented in figures 6a and 6b.

Simulator $\mathcal{S}$**.** Given black-box access to $\mathcal{A}$, $\mathcal{S}$ does the following while providing access to the functionalities $\mathcal{F}_{\mathsf{CoinFlip}}, \mathcal{F}_{\mathsf{RO}}$ to $\mathcal{A}$:

Input Sharing, OT and Square Correlation Generation Phases

1) Assuming gradients of all honest clients (i.e., the set $\mathcal{C}_H$) are zero vectors, generate input shares, OT correlations, and square correlations for the first round of messages that honest clients send by following the relevant steps of our protocol, and simulate the messages to $\mathcal{P}_M$ as if it is talking to honest clients.
2) Facilitate the calls to $\mathcal{F}_{\mathsf{CoinFlip}}$ made by $\mathcal{P}_M$ and store the output of the calls, i.e., values $\{\chi_i\}_i, t$ used to verify OT and square correlations. $\mathcal{S}$ now stores all the random challenge values used in verification phases, and can detect if $\mathcal{A}$ causes $\mathcal{P}_M$ to send the wrong challenge values to honest clients before the transcripts (in second round) are submitted. If such a malicious behaviour is found directed towards certain clients, $\mathcal{S}$ stops simulating any more messages for those clients to $\mathcal{P}_M$, and adds them to a censor list c.

3) $\mathcal{S}$ now simulates the second round (sending transcripts) to $\mathcal{P}_M$ based on the previous simulation of the first round and the random values $\{\chi_i\}_i, t$ it received from $\mathcal{P}_M$.
4) $\mathcal{A}$ prompts $\mathcal{C}_M$ to send shares of input, OT and square correlations, and transcripts ($\mathcal{S}$ simulates random values through $\mathcal{F}_{\mathsf{CoinFlip}}$ calls to $\mathcal{C}_M$ between first and second round). $\mathcal{S}$ receives them.
5) If $\mathcal{A}$ aborts, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and sends $\perp$ to $\hat{\mathcal{F}}$ as the final output.
6) If $\mathcal{A}$ causes some clients in $\mathcal{C}_M$ to abort before completing the messages they are supposed to send to $\mathcal{P}$, then $\mathcal{S}$ adds those clients to the censor list c.

OT and Square Correlation Verification, Bit Composition, ℓ_2 Computation, and ℓ_2 Enforcement Phases

1) Follows our protocol description and simulates the view of $\mathcal{P}_M$. It also answers the calls to $\mathcal{F}_{\mathsf{CoinFlip}}$ and $\mathcal{F}_{\mathsf{RO}}$ when hash function is invoked or fresh common random values between servers need to be sampled.
2) If at any step, transcript from previous phase doesn't match the protocol messages that $\mathcal{S}$ receives from $\mathcal{P}_M$, then it stops sending any more messages to $\mathcal{P}_M$ concerning the honest client for whom mismatch occurs, and adds that clients to the list c. This also takes into account situations where $\mathcal{P}_M$ stops sending messages for processing select clients' data.
3) If $\mathcal{A}$ aborts, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and sends $\perp$ to $\hat{\mathcal{F}}$ as the final output.

Aggregation Phase

1) Sends zero vectors as gradients of all clients in $\mathcal{C}_M$ to $\hat{\mathcal{F}}$ along with the censor list c, and receives back the aggregate of honest clients gradients, a_H, who were not in c (because malicious clients' gradients were sent as zeros).
2) Follows our protocol description to generate the final message in the view of $\mathcal{P}_M$ but corrects for the initial assumption that gradients of all honest clients are zeros by adding a_H back into this message before sending.
3) Receives $\mathcal{P}_M$'s final message and adds it into the message sent in the previous step to reconstruct the final output Y that $\mathcal{A}$ wants from $\hat{\mathcal{F}}$.
4) If $\mathcal{A}$ aborts, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs, and sends $\perp$ to $\hat{\mathcal{F}}$ as the final output.
5) Sends Y to $\hat{\mathcal{F}}$.
6) Outputs whatever $\mathcal{A}$ outputs.

The plan for our proof will be to first construct a *simpler* adversary $\mathcal{A}'$ in the real world such that $\mathsf{Real}_{\Pi',\mathcal{A}'(z)}(\{X_i\}_{i=1}^p, \lambda, \kappa)$ is always identical to $\mathsf{Real}_{\Pi,\mathcal{A}(z)}(\{X_i\}_{i=1}^p, \lambda, \kappa)$. An important characteristic of $\mathcal{A}'$ is that it follows the protocol as dictated by the transcript, and in the end submits a list of clients to censor and outputs the same final message in the aggregation phase as $\mathcal{A}$. With this simpler adversary, we then use a similar sequence of hybrids as used in the proof of Theorem A.1 until we reach the final hybrid which represents the ideal world.

Constructing $\mathcal{A}'$. Our simpler adversary runs $\mathcal{A}$ inside it, and plays the role of the honest server to $\mathcal{A}$. Whenever $\mathcal{A}$ deviates from the transcript submitted by the clients, $\mathcal{A}'$ records the client id, adds it to the censor list, and proceeds interaction with $\mathcal{A}$ as an honest server would. On the other side, $\mathcal{A}'$ interacts in the real world with S_0 (honest server) by relaying the messages sent by $\mathcal{A}$ directly, with an exception that whenever $\mathcal{A}$ deviates from the transcript of an honest client, $\mathcal{A}'$ uses the transcript to replace that message with what the client expects. Therefore, $\mathcal{A}'$ never deviates from the protocol transcript submitted by honest clients. Before the aggregation phase, $\mathcal{A}'$ sends the censor list to $\mathcal{P}$ and asks it to remove the contribution of clients in the list from the aggregation being done (this is the only difference between Π and Π'). $\mathcal{A}'$ finally outputs whatever $\mathcal{A}$ outputs.

We first show that $\mathsf{Real}_{\Pi',\mathcal{A}'(z)}(\{X_i\}_{i=1}^p, \lambda, \kappa)$ is identical to $\mathsf{Real}_{\Pi,\mathcal{A}(z)}(\{X_i\}_{i=1}^p, \lambda, \kappa)$. Both the adversaries $\mathcal{A}$ and $\mathcal{A}'$ have the same behavior and send the same messages out except for when $\mathcal{A}$ deviates from the transcript. Whenever this happens, our protocol description says that the party interacting with $\mathcal{A}$ treats this as the concerned client's inputs being censored, and therefore, the rest of the interaction with $\mathcal{A}$ proceeds ignoring that client. Until the final message in the aggregation phase, all messages are sent independently to process each client's inputs, and therefore, after deviating, $\mathcal{A}$ doesn't receive any more messages relating to the censored client. $\mathcal{A}'$, on the other hand, keeps using the transcript submitted by the client to play these messages to S_0, but asks S_0 before the aggregation phase to remove the submissions of the clients that were censored by $\mathcal{A}$. As long as the final message sent by S_0 doesn't include the censored inputs, this message is exactly the same as to what S_0 interacting with $\mathcal{A}$ would send. In other words, $\mathcal{A}'$ can be thought of as a middle-man which plays some extra messages to S_0, but then asks S_0 to remove all contribution from these messages in the final aggregate. Hence, $\mathcal{A}$'s interaction with $\mathcal{A}'$ (in Π') and S_0 (in Π) is identical.

We now prove indistinguishability between real and ideal worlds using a sequence of hybrids.

Hybrid 0 $\mathcal{H}_0$. We start with the real world.
Hybrid 1 $\mathcal{H}_1$. In this hybrid, we introduce $\mathcal{A}'$ by wrapping $\mathcal{A}'$ around $\mathcal{A}$ and moving from Π to Π'. We therefore change $\mathcal{P}$ to accept a censor list from $\mathcal{A}'$, remove all shares related to the clients in the list from any local computation, and then send the final message in the aggregation phase. Indistinguishability follows from the construction of $\mathcal{A}'$.

Case 1: $\mathcal{P}_M$ is S_0, i.e., OTSn

Hybrids 2 and 3 $\mathcal{H}_2, \mathcal{H}_3$. We follow the same strategy as $\mathcal{H}_1, \mathcal{H}_2$ in the proof of semi-honest server case (Theorem A.1). More concretely, we construct $\mathcal{H}_2$ by having honest clients give $\mathcal{P}$ access to $\mathcal{P}_M$'s share ($x^{(0)}$) and Δ. $\mathcal{P}$ uses this to generate fresh shares of zero while keeping one share as $x^{(0)}$ and then plays the OT verification phase messages based on the new shares. In $\mathcal{H}_3$, we have $\mathcal{P}$ use shares of zeros for honest clients throughout the computation, until the final aggregation phase message where the correction is

added back. In both these hybrids, $\mathcal{P}$ communicates necessary changes in the transcript back to the honest clients, so they can send a consistent transcript to the adversary. For indistinguishability, note that the outer adversary $\mathcal{A}'$ doesn't deviate from an honest execution of the protocol when dealing with the inputs of honest clients until the aggregation phase where it specifies the censor list. Ignoring the aggregation phase for now, we proved in Theorem A.1 that the view of $\mathcal{A}'$ is computationally indistinguishable from previous hybrids when following the above mentioned strategy of using zero vectors as honest clients' inputs. Moreover, transcripts that are sent by the clients just contain subset of this view, so don't leak any additional information. The only message (tainted with honest clients' inputs) that deviates from an honest execution of the protocol is during aggregation phase; this stems from the role of the censor list sent by $\mathcal{A}'$. However, $\mathcal{A}'$ taking non-negligible advantage of this fact would imply that it could tell computationally indistinguishable views apart. Therefore, the distribution formed by the pair of the output of $\mathcal{A}'$ (same as $\mathcal{A}$'s output) and honest parties is indistinguishable from previous hybrids.

Hybrid 4 $\mathcal{H}_4$**.** This is the ideal world. Unlike the semi-honest server scenario, the output of the ideal functionality $\mathcal{F}'$ is dictated by the adversary and the malicious clients' inputs are dealt with entirely by $\mathcal{S}$ (not $\mathcal{F}'$). Therefore, indistinguishability from previous hybrids doesn't rely on the soundness of OT and square correlations (because similar to the honest server in the previous hybrids, $\mathcal{S}$ uses the correlations to process its part of the malicious clients' inputs, while in the semi-honest server scenario, $\mathcal{F}$ didn't use the correlations at all). In fact, the output of $\mathcal{A}$ and honest parties is identically distributed as the previous hybrid $\mathcal{H}_3$.

Case 2: $\mathcal{P}_M$ **is** S_1**, i.e.,** OTRc

Hybrid 2 $\mathcal{H}_2$**.** Follow the same strategy as the semi-honest server proof (Theorem A.1) where $\mathcal{P}$ uses shares of zero for honest clients' inputs and adds back a correction in the final message sent during aggregation phase. Indinstinguishability follows similar to the case 1 above.

Hybrid 3 $\mathcal{H}_3$**.** This is the ideal world.

Remark on Transcript Digests. When we use our optimization of using transcript digests from clients instead of entire transcripts to achieve malicious privacy, the proof above requires a few changes. In the construction of our simpler adversary $\mathcal{A}'$, we can no longer use transcripts to "correct" the malicious messages that $\mathcal{A}$ sends when processing honest clients' inputs. Relying on digests means that $\mathcal{A}'$ cannot even detect if $\mathcal{A}$ is sending malformed messages until the protocol is run to the point that transcript digest can be compared with the observed transcript; this point is right before the results of intermediate checks are opened by the servers as mentioned in the "Reducing transcript communication" paragraph under Section 3.3. Once $\mathcal{A}'$ has identified a digest mismatch, it can immediately ask $\mathcal{P}$ to add the corresponding client to the censor list and the rest of the proof can go as it is. Now the only thing left is to analyze the interaction between the servers until digest mismatch can be detected. The way our protocol is designed (all checks deferred to the end), the communication between the servers during this period is indistinguishable from uniform. Therefore, $\mathcal{A}$ doesn't gain any non-negligible advantage in cheating compared to the case of using full transcripts.

□

Appendix C. Supplementary Material for Section 3

Negative values. Till now we have assumed that w bits are enough to secret share all values with magnitude bounded by $2^w - 1$. However, this doesn't hold when negative values are also present. In that case, Boolean shares of an extra sign bit can be used by the servers to securely multiplex [91] between either x or $2^u - x$ after bit composition is performed, where x is the value being processed. Using the protocol from CrypTFlow2 [91], this requires only two calls to $\binom{2}{1}$-OT_u per component of the gradient vector.

Appendix D. Round Complexity

Each client runs in a single round. For the servers, the total rounds of server-server interaction depend on the 2PC sub-protocol used in ℓ_2 enforcement. Ignoring ℓ_2 enforcement for a moment, servers require 2 rounds for OT and square correlation verification (done in parallel), 1 round for bit composition, 1 round for ℓ_2 computation, and 1 round for aggregation phase. Coming back to ℓ_2 enforcement, in Algo. 5, we use GMW [58] sub-protocol with a ripple-carry adder which incurs $2u+1$ rounds (we set $u = 64$ in our evaluation). We can significantly reduce these rounds to just $2 \log u + 1$ for ℓ_2 enforcement using an optimized parallel prefix adder [85] at the cost of increasing the number of secure AND gates from u to $u \log u$. Moreover, if achieving constant rounds is the focus, then constant-round 2PC like garbled circuits [113] can be used for the adder.