LERNA: Secure Single-Server Aggregation via Key-Homomorphic Masking

Hanjun Li[1(✉)], Huijia Lin[1], Antigoni Polychroniadou[2], and Stefano Tessaro[1]

[1] University of Washington, Seattle, WA, USA
{hanjul,rachel,tessaro}@cs.washington.edu
[2] J.P. Morgan AI Research & AlgoCRYPT CoE, New York, NY, USA
antigoni.polychroniadou@jpmorgan.com

Abstract. This paper introduces LERNA, a new framework for single-server secure aggregation. Our protocols are tailored to the setting where multiple consecutive aggregation phases are performed with the same set of clients, a fraction of which can drop out in some of the phases. We rely on an initial secret sharing setup among the clients which is generated once-and-for-all, and *reused* in all following aggregation phases. Compared to prior works [Bonawitz et al. CCS'17, Bell et al. CCS'20], the reusable setup eliminates one round of communication between the server and clients per aggregation—i.e., we need two rounds for semi-honest security (instead of three), and three rounds (instead of four) in the malicious model. Our approach also significantly reduces the server's computational costs by only requiring the reconstruction of a *single* secret-shared value (per aggregation). Prior work required reconstructing a secret-shared value for each client involved in the computation.

We provide instantiations of LERNA based on both the Decisional Composite Residuosity (DCR) and (Ring) Learning with Rounding ((R)LWR) assumptions respectively and evaluate a version based on the latter assumption. In addition to savings in round-complexity (which result in reduced latency), our experiments show that the server computational costs are reduced by two orders of magnitude in comparison to the state-of-the-art. In settings with a large number of clients, we also reduce the computational costs up to twenty-fold for most clients, while a small set of "heavy clients" is subject to a workload that is still smaller than that of prior work.

Keywords: Secure Aggregation · Reusable Setup · Privacy Preserving Machine Learning

1 Introduction

A secure aggregation protocol allows a set of clients, each holding an input x_i, to interact with one or more servers, so that the latter learns the sum $\sum x_i$, but no additional information. The inputs x_i could be integers, often mod q, or vectors of integers. In contrast to the usual setting of multi-party computation, which

J. Guo and R. Steinfeld (Eds.): ASIACRYPT 2023, LNCS 14438, pp. 302–334, 2023.
https://doi.org/10.1007/978-981-99-8721-4_10

assumes point-to-point channels, here communication only occurs between each individual client and the server(s), i.e., there is no *direct* inter-client communication and clients can only communicate indirectly through the server(s).

Secure aggregation protocols are suitable for a broad range of applications, such as privacy-preserving telemetry in browsers [14], analytics in digital contact tracing [2], and Federated Machine Learning [9]. Practical *multi-server* protocols [15,18] are, in fact, already being considered for standardization by IETF [22]. In this paper, however, we target the *single-server* setting. This setting is preferable whenever distributing trust among multiple non-colluding entities is not easily feasible. However, it is also more challenging, as protocols require multiple rounds of interaction and need to accommodate for potential *client dropouts*, whilst ensuring the correctness of aggregation and the privacy of clients' inputs against other colluding clients and/or the server. These protocols have emerged primarily in the context of Federated Machine Learning, starting from Bonawitz et al. [10], which underlies Google's Federated ML system [9], and its recent optimizations and extensions [6,7].

This paper introduces a new general paradigm for single-server secure aggregation, which improves upon the state-of-the-art in terms of round and computational complexities. Our protocols are particularly advantageous in settings where *repeated aggregation* phases are performed with the same set of clients (some of which may drop out) as they only require two rounds per aggregation, in addition to an initial setup round, at the presence of semi-honest colluding clients and/or server. In comparison, prior protocols [7,10] require three rounds per aggregation (without initial setup). In the malicious security model, all protocols require one additional round, namely, three rounds in our protocols and four rounds in prior works. Moreover, our approach also significantly improves the server workload by reducing the number of secret-sharing reconstructions.

Repeated Aggregation. While existing single server aggregation protocols mainly focus on running a single aggregation, many scenarios require running repeated aggregation sessions throughout a period of time, with the same set of clients. A prototypical application involves a number of sensors or nodes in a network reporting telemetry data. For example, a company of Internet of Things (IoT) devices may want to aggregate operation data from a certain area periodically to help understand how the devices are used throughout the day. Other examples include wireless sensor networks (WSN) [21], smart meters [3], and medical devices [23].

Our protocol leverages the repeated aggregation setting by having an initial setup round that generates correlated states among clients to facilitate the many aggregation phases later, reducing both round and computational complexity. The protocol is robust to drop-outs, as long as the fraction of drop-out clients is bounded at any point in time. Our main protocol focuses on the setting with a large number of clients, e.g. $M \geq 20K$. To reduce communication costs, it selects a committee of fixed size $O(\kappa^2)$ in the initial setup round to hold the correlated states. And the committee stays unchanged through out many aggregation phases. The protocol guarantees the privacy of clients' inputs against

statically corrupted clients that may collude with the server, provided that the total number of corrupted clients in the setup and all aggregation phases are bounded. Compared to protocols designed for single aggregation, we rely on the more stringent condition that the total number of corrupted clients is bounded across many aggregation sessions. However, one can alleviate this assumption by periodically rerunning the setup phase, generating fresh correlated states among clients. Different applications may refresh at a different frequency, say, every day, every week, or even longer, depending on how likely clients are corrupted. Viewed this way, our protocol offers a new tradeoff between the rate of corruption and efficiency gain.

Alternatively, when the number of client is small, e.g. $M \leq 80$, our protocol can avoid the committee in the initial setup round to guarantee stronger privacy: in this setting, the clients may be *adaptively* corrupted instead of statically as assumed above. (See the full version for details on this variant.)

Existing Single-Server Secure Aggregation. It is helpful to first review the blueprint behind existing single-server aggregation protocols [7,10]. Here, we restrict ourselves to the semi-honest setting for simplicity, but these protocols (along with ours) can be modified to support malicious corruption of server and clients.

The initial idea is to have each client $i \in [M]$ send a *masked* input $z_i = x_i + c_i$ to the server. To generate these masks, every pair of clients i, j establishes a shared key $k_{ij} = k_{ji} = \mathrm{PRG}(g^{s_i s_j})$, where g^{s_i} is a group element which acts as an ephemeral *public key* associated with each client $i \in [M]$, and which is shared in an initial round (through the server) with all other clients. The value s_i is kept secret by client i. Then, each client $i \in [M]$ uses the mask

$$c_i = \sum_{j<i} k_{ij} - \sum_{j>i} k_{ij} \ .$$

These masks satisfy in particular the *cancellation property* $\sum_i c_i = 0$, and consequently the server can simply output $\sum_i z_i = \sum_i x_i$.

A first concern is that this only works if each client remains alive and indeed submits its own masked input—a term $k_{ij} = k_{ji}$ included in client j's mask c_j is not canceled out without client i's contribution. To handle a dropout, each client additionally secret shares their own secret s_i, which is reconstructed in case they drop out, to then, in turn, derive all k_{ji}'s for $j \neq i$.

A second concern is that a slow client i could be prematurely labeled as a dropout, and their secret s_i reconstructed *before* the masked value z_i reaches the server, thus revealing x_i. To prevent this, each client initially shares a second random mask b_i, along with s_i, and sends instead the masked input $z_i = x_i + b_i + c_i$ to the server. Then, after receiving the masked inputs $\{z_i\}_{i \in I}$ from a subset $I \subseteq [M]$ of the clients, for each $i \in I$, the server reconstructs b_i, thus allowing the inclusion of $(x_i + c_i)$ in the final sum. In contrast, it reconstructs s_i for all $i \notin I$, thus enabling the computation of $\sum_{i \in I} x_i$ as discussed above. For every client $i \notin I$, because b_i remains secret, the value x_i remains protected even if later z_i is obtained by the adversary.

Therefore, the overall protocol needs three rounds. An additional round is needed to tolerate a malicious server, and it forces the server to commit to a single set I of clients which are claimed not to have dropped out.

The Costs of Secret Sharing. The most expensive part in the above blueprint is the initial sharing of s_i and b_i, along with the later reconstruction of (one of) them for each client. This impacts both the round and computational complexity in several ways.

Foremost, secret sharing s_i and b_i takes one additional round of communication. While some initial setup round is somewhat inherent (e.g., to share keys to allow clients to communicate with each other via the server), this becomes a bigger concern in the repeated aggregation setting. Here, it is crucial that the values s_i and b_i are re-generated and re-shared *at each repeated session*, for otherwise dropping out at some later session may compromise the privacy of the inputs from prior sessions.

Moreover, the computation and communication costs due to secret sharing are high – $\Theta(M)$ for each client, and $\Theta(M^2)$ for the server. Crucially, the server needs to reconstruct one secret shared value—either s_i or b_i—for each client. In addition, for every client dropout, the server needs to perform $\Theta(M)$ exponentiations to recover the corresponding values k_{ij}. To reduce costs, Bell et. al. [7] proposed to have clients only secret share in a random neighborhood of size $\Theta(\log M + \kappa)$, where κ is the statistical security parameter. Though this idea reduces the client and server costs to $\Theta(\log M + \kappa)$ and $M(\Theta(\log M + \kappa))$, respectively, the improvement is at the cost of weakening the security guarantees at the presence of maliciously corrupted clients and/or server.[1]

Our Contributions. This paper proposes LERNA, a new lightweight approach to single-server secure aggregation which addresses the aforementioned issues. Foremost, it reduces the round complexity to two respectively three communication rounds for semi-honest and malicious security, respectively, in addition to an initial offline round which establishes a setup that can be re-used across multiple aggregations. Moreover, LERNA also features very small server costs, as the server only needs to perform a single reconstruction of a secret-shared value. We validate the performance of LERNA also by benchmarking a prototype implementation.

An important feature of our implementation is that it identifies a (random) subset of the clients as a *committee*. Our benchmarking shows that the computational costs of committee members are smaller than the client costs of prior solutions. However, LERNA is even more lightweight for clients outside of the committee. Indeed, in addition to participating in an initial setup stage, non-members only need to send a single message to the server to include their input in an aggregation session, and subsequent interaction within the same session only involves committee members. Our benchmarking demonstrates up to twenty-fold performance improvement for these non-committee clients.

[1] More specifically, using the protocol of Bell et. al., if the server is malicious, it may recover the sums of inputs of multiple subsets of clients.

A drawback of our solution, as shown in our benchmark, is a relatively heavy communication cost in the initial offline round. This requires participating devices to have sufficient storage and network bandwidth. To amortize this one-time cost, an ideal application for LERNA runs repeated aggregation for large numbers of iterations, T, before rerunning the setup. We envision running LERNA for machine learning from data collected from a large number of relatively powerful devices, e.g. the payment terminals Amazon One, medical imaging devices, weather stations, etc.

Our protocols are built on top of a new primitive, which we call a *key-homomorphic masking* scheme, which allows clients to initially secret share a *re-usable* secret value (i.e., which can be reused across multiple computations) to the committee as part of the initial offline round. We provide two instantiations from, respectively, the DCR assumption [16] and (Ring) LWR assumption [5], with the latter being our main result.

Related Work. The same reduction in round complexity was very recently achieved by Guo et al. [19], also relying on a re-usable secret shared value. However, their solution performs the aggregation in the exponent of a discrete-log hard group, resulting essentially in the sever obtaining the value $g^{\sum_i x_i}$, where g is a group generator. In other words, the actual result can only be extracted by computing the discrete logarithm, which is feasible only if $\sum_i x_i$ is sufficiently small. This forces the computation to be over small domains accommodating Federated learning of models with small weights, such as quantized or compressed models. In contrast, most Federated ML tasks typically involve large values. LERNA does not suffer from this drawback. Our approach differs from [19] in that it relies on different mathematical structures (underlying the LWR and DCR assumptions) to obtain the aggregated sum *in the clear*. This, in turn, requires overcoming a few challenges, in particular, designing special secret-sharing schemes tailored to our requirements – linear reconstruction via small coefficients (for LWR) and working over the integers (for DCR).

The work of [20] proposed a semi-honest protocol, SASH+, using a seed-homomorphic PRG based on LWR similar to our key-homomorphic masking scheme. However, SASH+ exploits the homomorphic property in a different way from LERNA. At high-level, assuming LWR with dimension n, SASH+ reduces the problem of aggregating ℓ-dimension inputs to aggregating n-dimensional homomorphic PRG seeds, which is done using the protocol of [7]. This reduction reduces the computation cost of the server and each client by roughly a factor of (ℓ/n), but at the cost of increasing the round complexity from 3 to 4 per iteration, and introducing an error to the aggregation result that scales linearly with M. In comparison, LERNA reduces the round complexity from 3 to 2, and improves the computation cost at the same time. LERNA also computes the aggregation results exactly without error. As we'll discuss in our benchmarks, LERNA server, and non-committee clients are significantly faster than SASH+'s, while LERNA committee clients become slower than SASH+ clients for very large M.

The work of [25] focuses on the specific application of repeated aggregation in federated machine learning (FL), where the server selects a random subset of

clients to aggregate at each iteration. It observes that the usual random client selection strategy in FL causes a leakage of client inputs when the model is close to converged. The paper proposes a new client selection algorithm to mitigate this leakage, assuming an honest server following this new algorithm. We note that LERNA can also be adapted to run repeated aggregation over a different subset of clients at each iteration. The mitigation strategy can then be orthogonally applied to the semi-honest version of LERNA. We stress that the client selection strategy is not to be confused with LERNA's committee selection. Client selection could be added on top of our protocol (but is not included explicitly), and would happen in every iteration, whereas committee selection is within our protocol, and happens only once during its setup phase.

A recent and concurrent work by Bell et al. [6] additionally considers the question of input validation. While this is extremely important, it is orthogonal to the issues studied by this paper. Their system also uses Ring-LWE for efficiency improvement, but still follows broadly the above blueprint without a re-usable setup.

1.1 Overview of LERNA

LERNA's approach differs from the existing protocols in [7,10] whose core idea is hiding each input x_i with a masks that, as described above, satisfies the cancellation property. Instead, LERNA starts with a conceptually simpler solution, where each client i hides its input x_i with a (random) mask c_i as $z_i = x_i + c_i$, and sends the masked value z_i to the server. With the help of the clients, the server first recovers $c_U = \sum_{i \in U} c_i$, for the set of online clients U, and hence the aggregation result $x_U = \sum_{i \in U} z_i - c_U$. The key question we answer is how the clients securely help the server to compute c_U.

Straw Man Solution. The first naïve idea is to let every client secret share its mask c_i with all other clients using a *linear* secret sharing scheme $(\mathsf{Share}, \mathsf{Recon})$, such as Shamir's secret-sharing scheme. In particular, the Recon algorithm involves evaluating a linear function on the shares. As in prior works [7,10] each client only has a private and authenticated channel with the server. They can also communicate with each other indirectly through the server. Assuming a PKI setup, such indirect communication can be private and authenticated.

In more detail, each client $i \in [M]$ sends (through the server) the j'th share c_j^i of c_i to each other client $j \in [M]$, before sending their masked input $z_i = c_i + x_i$. The server then finds the set of clients U who have completed both steps, and notifies them of the set U for aggregation. Each client j then locally aggregates the shares it has received from clients $i \in U$, obtaining $c_j^U = \sum_{i \in U} c_j^i$. By the linear homomorphism of the secret sharing, c_j^U is the j'th share of the aggregated mask c_U. As long as enough clients, say $j \in U' \subseteq U$, send their aggregated shares c_j^U to the server, the latter can reconstruct $c_U = \mathsf{Recon}(\{c_j^U\}_{j \in U'})$, and then recover the aggregated input x_U.

This simple solution is, however inefficient: The step where each client i shares its mask c_i with all other clients has overall $\Omega(M^2)$ communication complexity per aggregation. To aggregate T times, the cost grows as $\Omega(M^2 \times T)$.

Key-Homomorphic Masking Scheme. Somewhat informally a key-homomorphic masking scheme involves a pair of algorithms $\mathsf{Mask}, \mathsf{UnMask}$. The Mask algorithm takes an input x from some input space $\mathbb{Z}_p$, a masking key k from some key space $\mathcal{K}$, and a tag τ, and computes a masked message $z \leftarrow \mathsf{Mask}(k, \tau, x)$. The UnMask algorithm takes the above z and an "empty" mask $c \leftarrow \mathsf{Mask}(k, \tau, 0)$ under the same key k and tag τ, and recovers the message $x \leftarrow \mathsf{UnMask}(z, c)$.

Importantly, the scheme is *additively key-homomorphic* for masks with the *same* tag τ: $\mathsf{Mask}(k+k', \tau, x+x') \equiv \mathsf{Mask}(k, \tau, x) \boxplus \mathsf{Mask}(k', \tau, x')$, where $\boxplus$ represents homomorphic addition. We can generalize the additive homomorphism to evaluate any linear function L over masks $\{z_i \leftarrow \mathsf{Mask}(k_i, \tau, x_i)\}$:

$$\mathsf{Eval}(L, \{z_i\}) \equiv \mathsf{Mask}(L(\{k_i\}), \tau, L(\{x_i\})) \ ,$$

where the linear function L is evaluated respectively over k_i's in the key space $\mathcal{K}$ and over x_i's in the message space $\mathbb{Z}_p$.

Jumping ahead, our instantiation of the masking scheme under LWR will only achieve *approximate* key-homomorphism. We will explain below how we get around this limitation. For now, it is helpful to assume a perfect masking scheme to convey the main idea.

Sketch of the LERNA Protocol. We now describe the semi-honest protocol. Note that the following description depends on a commitment $Q \subseteq [M]$. One can easily think of this committee as containing all clients, although in our concrete instantiation below, we only include a (random) subset of the clients in Q

- Setup phase: The clients agree on a common committee $Q \subseteq [M]$ using public, common randomness. Every client i secret shares a fresh masking key k_i as $\{k_j^i\}_{j \in [Q]}$ and sends the j'th share k_j^i to committee member $j \in Q$.
- Online phase: In the t^{th} aggregation session,
  1. The clients sample a common tag $\tau \leftarrow \mathcal{H}(\mathsf{sid}, t)$ using a hash function $\mathcal{H}$, modeled as a random oracle. Every client P_i then computes a masked input $z_i \leftarrow \mathsf{Mask}(k_i, \tau, x_i)$ and sends z_i to the server.
     The server identifies the set U of online clients. It sends U to all committee members Q, indicating that it wants to aggregate the inputs in U.
  2. Upon receiving U, every committee member P_j aggregates the key shares k_j^i it received from clients $i \in U$, obtaining $k_j^U = \sum_{i \in U} k_j^i$, which by linear homomorphism, equals the j'th share of $k_U = \sum_{i \in U} k_i$. (Therefore, given enough shares $\{k_j^U\}_{j \in U'}$, for a large enough subset U', one can recover k_U.) Then, P_j computes an empty mask $c_j^U \leftarrow \mathsf{Mask}(k_j^U, \tau, 0)$, and sends it back to the server.

Upon receiving enough shares $\{c_j^U\}_{j\in U'}$ from a subset $U' \subseteq U$, the server homomorphically computes the aggregated mask

$$\begin{aligned} c_U &= \mathsf{Eval}(\mathsf{Recon}, \{c_j^U\}_{j\in U'}) \\ &\equiv \mathsf{Mask}(\mathsf{Recon}(\{k_j^U\}_{j\in U'}), \tau, 0) \equiv \mathsf{Mask}(k_U, \tau, 0) \end{aligned}$$

where the first equivalence uses the fact that the Recon algorithm is linear. Similarly,

$$\begin{aligned} z_U &= \sum_{i\in U} z_i = \sum_{i\in U} \mathsf{Mask}(k_i, \tau, x_i) \\ &\equiv \mathsf{Mask}(\sum_{i\in U} k_i, \tau, \sum_{i\in U} x_i) = \mathsf{Mask}(k_U, \tau, x_U) \end{aligned}$$

The server can now recover $x_U = \mathsf{UnMask}(z_U, c_U)$.

LWR-Based Instantiation. Our main instantiation of the masking scheme is inspired by the simple seed-homomorphic PRG of [11]. The LWR assumption [5] is associated with two moduli $q > p$, where p is the modulus of the message space. A tag τ is an LWR public vector $\mathbf{a} \in \mathbb{Z}_q^n$, and the masking key k is an LWR secret $\mathbf{s} \in \mathbb{Z}_q^n$. A masked input z is simply an LWR sample rounded to p added with the message x, i.e.,

$$\text{LWR: } \tau = \mathbf{a} \in \mathbb{Z}_q^n, \qquad k = \mathbf{s} \in \mathbb{Z}_q^n, \qquad z = \lfloor \langle \mathbf{s}, \mathbf{a} \rangle \rceil_p + x \in \mathbb{Z}_p \ .$$

The linear structure of LWR implies the key homomorphism property. However, it only holds *approximately* due to rounding errors. More specifically: *i)* additive key-homomorphism holds approximately with bounded error, and *ii)* linear key-homomorphism holds with bounded error if the linear function L evaluated has *small coefficients*. To see *i)*, consider two masks with keys $k_1 = \mathbf{s}_1, k_2 = \mathbf{s}_2$, inputs x_1, x_2, and a common tag $\tau = \mathbf{a}$. We have

$$\begin{aligned} z_1 + z_2 &= \lfloor \langle \mathbf{s}_1, \mathbf{a} \rangle \rceil_p + x_1 + \lfloor \langle \mathbf{s}_2, \mathbf{a} \rangle \rceil_p + x_2 \\ &= \lfloor \langle \mathbf{s}_1 + \mathbf{s}_2, \mathbf{a} \rangle \rceil_p + x_1 + x_2 + \epsilon \ , \end{aligned}$$

where ϵ is the rounding difference between $\lfloor \langle \mathbf{s}_1, \mathbf{a} \rangle \rceil_p + \lfloor \langle \mathbf{s}_2, \mathbf{a} \rangle \rceil_p$ and $\lfloor \langle \mathbf{s}_1 + \mathbf{s}_2, \mathbf{a} \rangle \rceil_p$, which is bounded by 1. With regard to *ii)*, when evaluating a linear function L over the masks using the above approximate additive homomorphism, the error is scaled by the coefficients of L.

The approximate key homomorphism creates a technical issue in the protocol: when the server evaluates Recon homomorphically, it introduces an additive error in the aggregation result. To remove the error, our solution is to multiply the inputs with a scaling factor Δ, set to be larger than the noise.

If the coefficients of Recon are large – e.g., as in Shamir's secret sharing – then the error induced by homomorphic evaluation, and hence the scaling factor Δ becomes large, causing a significant overhead in the protocol. To minimize

this overhead, we will use a linear secret-sharing scheme whose reconstruction function has only $-1, 0, 1$ coefficients – referred to as the *flatness* property. An additional benefit of the flatness property is that Recon becomes computationally cheaper, involving only simple additions and subtractions.

Committee Based Flat Secret Sharing Scheme. As motivated above, we need a secret sharing scheme with small reconstruction coefficients. One solution appears to come from the work of [17], which transforms any monotone Boolean formula for the threshold function into a linear secret-sharing scheme with small coefficients, satisfying flatness. Unfortunately, however, known constructions of Boolean formulae for the threshold function with M inputs has a size $\Omega(M^{5.3})$ [26], which by the transformation of [17] gives a secret sharing consisting of $\Omega(M^{5.3})$ elements in total. This is prohibitively expensive and recent work [4] indicates several challenges in improving this.

Our committee-based construction follows the blueprint of [17], but drastically reduces the total share size from $\Omega(M^{5.3})$ to $\Theta(\kappa^2)$ where κ is the security parameter. Our key observation is that in the setting of secure aggregation, a much weaker secret sharing scheme (than that of [17]) suffices:

1. Instead of using a monotone Boolean formula for threshold functions, it suffices to consider *gap threshold* functions. Such a function outputs 1 if more than ρ fraction of the inputs are 1 and outputs 0 if less than $\gamma < \rho$ fraction of the inputs are 1 (and has no guarantees for inputs in between). The values of ρ and γ correspond to the reconstruction and privacy thresholds in the context of secret sharing.
2. Instead of using a single formula, we use a *distribution* $\mathcal{F}$ of formulae. Our secret-sharing scheme has a setup phase where a formula is sampled $f \leftarrow \mathcal{F}$. As such, the security and correctness of secret sharing only need to hold with overwhelming probability over the random choice of f.
   Sampling $f \leftarrow \mathcal{F}$ directly translates to sampling a committee of share holders in the secret sharing scheme, corresponding to the committee Q chosen in the setup phase of our protocol above.
3. In fact, we do not even need formulae that compute exactly the gap threshold function. Instead, it suffices if for every "promised" input x, a random formula $f \leftarrow \mathcal{F}$ computes the correct output with overwhelming probability. That is,

$$\forall x \text{ with hamming weight } < \gamma M \text{ or } > \rho M,$$
$$\Pr\left[f(x) \text{ correct } | f \leftarrow \mathcal{F}\right] > 1 - \text{negl}(\kappa).$$

These relaxations allow us to modify the randomized construction of formulae for threshold function in [26] to obtain a distribution of formulae with sizes $\Theta(\kappa^2)$ satisfying the above. The transformation of [17] then gives a committee based secret sharing consisting of only $\Theta(\kappa^2)$ elements in total, with a $\Theta(\kappa^2)$ size committee.

Server Efficiency. LERNA admits very efficient server computation. Upon collecting all the masked inputs z_i and all the mask shares c_i^U, the server simply

computes a sum $\sum_{i \in U} z_i$, reconstruction over shares c_i^U, and finally unmasks. Since our secret sharing has 0/1 coefficients, reconstruction is also computing a sum. LERNA server is 100× faster than that of prior work [7], where the server needs to perform $\Theta(M)$ reconstruction of Shamir's secret sharing, and $\Theta(M(\log M + \kappa))$ group exponentiations. See Sect. 5 experimental data, and the full version for asymptotic comparisons.

Static vs. Adaptive Corruptions. One consequence of the above approach is that the random choices involved in sampling the formula (i.e., the committee of share holders) need to be independent of corruptions and dropouts in an execution of the protocol, which we expect to be chosen non-adaptively (for dropouts, in fact, we only require an overall set of potential dropouts to be fixed non-adaptively, but when individual parties drop out can be chosen adaptively). We stress that this assumption already inherently underlies the optimized aggregation protocol from [7], which relies on choosing a random graph independently of corruption and dropout patterns.

For the setting where the number of clients is small, e.g. $M \leq 80$, we show an alternative instantiation of LERNA that doesn't involve sampling a committee of share holders in the full version. In this variant, LERNA tolerates adaptive corruption.

2 Preliminaries

In this section, we explain the system and failure models of LERNA, and give an overview of LERNA's security requirements. We provide a formal security definition in the UC framework in the full version.

System Model. LERNA is a framework for secure aggregation involving M clients and a single server. Different from the systems in [7,10], LERNA has a one-time setup phase followed by many, T, online phases (also referred to as aggregation sessions). The setup phase creates correlated secrets $s_1, \ldots, s_M$ among the M clients, which are re-used in all following online phases. During each online phase, the server computes the aggregation over fresh inputs $\mathbf{x}_1, \ldots, \mathbf{x}_M$ from the same set of clients. The inputs to the clients $\mathbf{x}_i \in \mathbb{Z}^\ell$ are large integer vectors from a bounded (but potentially exponentially large) range, and the aggregation results are computed coordinate-wise over the integers.

Communication Model. Similar to prior work, LERNA has a simple communication pattern. During the online phases, each client communicates only with the server through private and authenticated channels. During the setup phase, the clients communicate indirectly with each other through the server, also in a private and authenticated way. This can be achieved by assuming a PKI setup, or, to avoid the PKI setup, the clients can run pairwise key-agreement through the server at the beginning of the setup phase. We need to assume (similarly to [7,10]) the server behave honestly in the key-agreement round.

The LERNA protocol proceeds in rounds. In each round, each client may send one message to the server, and may receive a reply message from the server. For simplicity, we assume synchronized communication channels.

Failure Model. LERNA is designed to be robust against two types of failures, corruption and dropout. For the first type of failure, a subset of the parties, may or may not include the server, collude to try to learn the individual input of the other clients. We further differentiate static and adaptive corruptions. In static corruption, the adversary selects a subset of corrupted parties at the beginning of the protocol execution. In adaptive corruption, the adversary is free to choose which party to corrupt at any stage of the protocol execution. Our main protocol in Sect. 4, suitable for running with large number of clients, tolerates static corruption. The variant described in the full version for running with small number of clients tolerates adaptive corruption. In the semi-honest setting, the adversary learns the inputs and the internal states of the corrupted parties, throughout the setup phase and all online phases. In the malicious setting, the adversary controls the actions of the corrupted parties entirely.

For the second type of failure, a potentially different subset of clients drop out from each online phase (and may come back in the future). We model no clients dropout during the setup phase. This is equivalent to saying only the set of clients who complete the setup phase is considered during the following online phases. More precisely, we model the dropout failure by allowing the adversary to choose a set of potential dropout clients D_t for each online phase t, all at the beginning of the protocol. The adversary is allowed to adaptively decide whether and when each client $P_i \in D_t$ (from the potential set) actually drops out during the online phase t.

Security Definition. The security of LERNA has two aspects: correctness and privacy. They are parameterized by a constant fraction δ, which represents the fraction of dropout clients tolerated by LERNA.

Correctness guarantees that in the semi-honest setting, the server computes the correct result in a session, as long as less than δM clients drop out in that session. In contrast, in the malicious setting, a corrupted client may arbitrarily "pollute" the aggregation result or cause it to be $\perp$, indicating an error.

For privacy, we consider an adversary that *statically* corrupts at most a γ fraction of the clients, before the aggregation protocol begins. We tolerate any fraction $0 \leq \gamma < 1 - \delta$. The adversary may additionally corrupt the server. The following privacy guarantee applies to both the semi-honest and the malicious settings.

In the simpler case, where only clients but not the server are corrupted, the adversary learns only the corrupted clients' inputs in each aggregation session and nothing else. In the case where the server is also corrupted, the adversary learns the corrupted clients' inputs, as well as a single sum of the honest clients' inputs in a sufficiently large set $U \subseteq [M]$, where $|U| > (1 - \delta)M$.

For comparison, the privacy guarantee of [7] is weaker. In the case where both the server and a subset of the clients are corrupted, an adversary may learn *multiple* non-overlapping sums of the honest inputs in each aggregation session. Their security guarantees that each such sum contains at least $\Omega(\log M)$ inputs, which provides a weaker degree of anonymity.

Formally, we define the security of LERNA in the UC framework [13]. Details of the UC framework and our formal security definition are deferred to the full version.

3 Technical Tools

In this section, we construct two technical tools, a key-homomorphic masking scheme and a flat secret sharing scheme. As outlined in the technical overview (Sect. 1.1), the masking scheme is used for hiding clients' input vectors, and the secret sharing scheme is used for sharing each client's secret masking key.

3.1 Key-Homomorphic Masking

We first introduce the syntax of a key-homomorphic masking scheme.

- $\mathsf{Setup}(1^\lambda, \ell, B_{\mathrm{msg}})$: takes as inputs the security parameter λ, a message dimension ℓ, and a lower bound B_{msg} on the message modulus. It outputs public parameters pp, which defines a key space $\mathcal{K}$, a message space $\mathbb{Z}_{p_m}^{\ell}$ with some modulus $p_m \geq B_{\mathrm{msg}}$, and a mask space $\mathbb{Z}_q^{\ell}$ with some modulus q.

In our framework, we assume every client enters the setup phase (Fig. 1) with common correctly generated public parameters pp. If the Setup algorithm is deterministic, or public-coin, then this assumption is simply a notational convenience, since each client can compute the common pp on its own, using a random oracle to derive common public randomness if necessary.

- $\mathsf{KeyGen}(\mathsf{pp})$: outputs a masking key $k \in \mathcal{K}$.
- $\mathsf{TagGen}(\mathsf{pp})$: outputs a tag τ.

In our framework, each client P_i derives its secret masking key k_i in the setup phase, and re-uses it during all online phases. In contrast, it derives a fresh tag τ for each online phase, using common public randomness. We only require the key-homomorphic property to hold for masks under the same tag τ. While the tag is public to all clients, the masking keys must remain secret.

- $\mathsf{Mask}(\mathsf{pp}, k, \tau, \mathbf{m})$: takes as inputs a masking key $k \in \mathcal{K}$, a tag τ, and a message $\mathbf{m} \in \mathbb{Z}_{p_m}^{\ell}$, and outputs a masked message $\mathbf{c}_m$.
- $\mathsf{UnMask}(\mathsf{pp}, \mathbf{c}_m, \mathbf{c}_0)$: takes as inputs a masked message $\mathbf{c}_m$, and an "empty" mask $\mathbf{c}_0$ (of message $\mathbf{0}$) under the same key and tag. It recovers a message $\mathbf{m}^*$ or $\perp$.

The UnMask algorithm is a bit unusual, as it doesn't take the masking key k or the tag τ to recover the message. Instead, it asks the caller to first compute an empty mask $\mathbf{c}_0$ using the key k and tag τ, and then feed $\mathbf{c}_0$ to the algorithm. We define such a syntax because in our framework, the caller of UnMask is the server. The clients jointly help the server compute the empty mask $\mathbf{c}_0$, instead of revealing their masking keys, so that the keys remain secret during each online phase.

– $\mathsf{Eval}(\mathsf{pp}, L, \{\mathbf{c}_i\})$: takes as inputs a linear function L with d integer coefficients and d masks $\{\mathbf{c}_i\}_{i\in[d]}$. It homomorphically evaluates L on the masks and outputs the result $\mathbf{c}_L$.

As mentioned earlier, the input masks $\{\mathbf{c}_i\}$ to the Eval algorithm should be masked under a common tag τ. Evaluating L on the masks roughly translates to evaluating L on both the masking keys, over the key space $\mathcal{K}$, and over the messages, over the message space $\mathbb{Z}_{p_m}^{\ell}$. We define this property below as key-homomorphism.

Correctness. Formally, we define the correctness and the key-homomorphism requirement as follows.

Definition 1 (correctness)**.** *For all public parameters* pp*, tags* τ*, and keys* k *output by* Setup*,* TagGen *and* KeyGen*, and for all messages* $\mathbf{m} \in \mathbb{Z}_{p_m}^{\ell}$*, the following holds.*

$$\Pr\left[\mathsf{UnMask}(\mathsf{pp}, \mathbf{c}_m, \mathbf{c}_0) = \mathbf{m} \;\middle|\; \begin{array}{l} \mathbf{c}_m \leftarrow \mathsf{Mask}(\mathsf{pp}, k, \tau, \mathbf{m}), \\ \mathbf{c}_0 \leftarrow \mathsf{Mask}(\mathsf{pp}, k, \tau, \mathbf{0}). \end{array}\right] = 1.$$

Definition 2 (key-homomorphism)**.** *Consider any linear function* L*, represented by* d *integer coefficients. For all public parameters* pp*, tag* τ*, and keys* $k_1, .., k_\ell$ *output by* Setup*,* TagGen*, and* KeyGen*, and all messages* $\mathbf{m}_1, ..., \mathbf{m}_d \in \mathbb{Z}_{p_m}^{\ell}$*, the following holds.*

$$\begin{aligned} &\big\{\tilde{\mathbf{c}}_L \leftarrow \mathsf{Mask}\big(\mathsf{pp}, L(\{k_i\}), \tau, L(\{\mathbf{m}_i\})\big)\big\} \\ \equiv &\big\{\mathbf{c}_L \leftarrow \mathsf{Eval}(\mathsf{pp}, L, \{\mathbf{c}_i\}) \mid \mathbf{c}_i \leftarrow \mathsf{Mask}(\mathsf{pp}, k_i, \tau, \mathbf{m}_i)\big\}, \end{aligned}$$

where $L(\{\mathbf{m}_i\})$ *is evaluated over* $\mathbb{Z}_{p_m}^{\ell}$ *and* $L(\{k_i\})$*, over the key space* $\mathcal{K}$*.*

The key-homomorphism definition above requires the evaluated mask $\mathbf{c}_L$ to have the same distribution as the "target" mask $\tilde{\mathbf{c}}_L$. We next introduce a relaxation to this rather strong property. Roughly, the evaluated mask $\mathbf{c}_L$ should be distributed close to the target mask $\tilde{\mathbf{c}}_L$. In other words, through homomorphic evaluation we obtain the target mask with some bounded additive noise.

Our framework requires two additional properties from an approximate key-homomorphic scheme. First, when computing UnMask on a masked input $\mathbf{c}_m$ and an empty mask $\mathbf{c}_0$, any additive noises in them translate to additive noises in the recovered message. Second, when computing Eval on noisy masks, the additive noises translates to an additive noise in the evaluated mask, with bounded magnitude. We formalize the above requirements as follows.

Definition 3 (ϵ*-approximate* key-homomorphism)**.** *Consider any linear function* L*, with* d *integer coefficients whose absolute values are bounded by some* $B_L \in \mathbb{N}$*.*

– *Let* $\tilde{\mathbf{c}}_L$*,* $\mathbf{c}_L$ *be the evaluated and the "target" masks as defined in Definition 2. We require*

$$\|\tilde{\mathbf{c}}_L - \mathbf{c}_L\|_\infty \leq \epsilon d B_L.$$

– *Let* $\mathbf{c}_m, \mathbf{c}_0$ *be the masked input and the empty mask as defined in Definition 1. For all integer noise vectors* $\mathbf{e}_1, \mathbf{e}_2 \in \mathbb{Z}^\ell$*, we require*

$$\mathsf{UnMask}(\mathsf{pp}, \mathbf{c}_m + \mathbf{e}_1, \mathbf{c}_0 + \mathbf{e}_2) = \mathbf{m} + \mathbf{e}_1 + \mathbf{e}_2 \ \textit{mod}\ p_m.$$

– *Let* pp, $\{\mathbf{c}_i\}$ *be the public parameters and the masks as defined in Definition 2. For all integer noise vectors* $\{\mathbf{e}_i\}$*, whose values are bounded by some* $B_e \in \mathbb{N}$ *we require*

$$\|\mathsf{Eval}(\mathsf{pp}, L, \{\mathbf{c}_i\}) - \mathsf{Eval}(\mathsf{pp}, L, \{\mathbf{c}_i + \mathbf{e}_i\})\|_\infty \leq B_e d B_L.$$

Security. For security, we require a mask under a randomly chosen key hides its message. We further require this holds for a polynomial number of adaptive sessions, each with a fresh tag sampled with public randomness, reusing the same key.

Definition 4 (security)**.** *Let* λ *be the security parameter. The masking scheme is* secure *if for all input dimension* $\ell = \ell(\lambda) \leq \mathrm{poly}(\lambda)$ *and message modulus lower bound* $B_{msg} = B_{msg}(\lambda) \leq 2^{\mathrm{poly}(\lambda)}$*, any efficient adversary* $\mathcal{A}$ *has negligible advantage in distinguishing the experiments* $\mathsf{Exp}^{\mathcal{A},b}_{\mathsf{Mask}}(1^\lambda)$ *defined as follows:*

– *The challenger computes* $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda, \ell, B_{msg})$*, and samples a masking key* $k \leftarrow \mathsf{KeyGen}(\mathsf{pp})$*. It launches* $\mathcal{A}(1^\lambda)$*, sends* pp *to* $\mathcal{A}$*, and repeats the following steps until* $\mathcal{A}$ *outputs a bit* b'*.*
  1. *Run* $\tau \leftarrow \mathsf{TagGen}(\mathsf{pp}; r)$ *using fresh randomness* r*, and send* (τ, r) *to* $\mathcal{A}$*.* $\mathcal{A}$ *replies with a message* $\mathbf{m} \in \mathbb{Z}^\ell_{p_m}$*.*
  2. *If* $b = 1$*, compute* $\mathbf{c}_1 \leftarrow \mathsf{Mask}(\mathsf{pp}, k, \tau, \mathbf{m})$*. Otherwise, compute* $\mathbf{c}_0 \leftarrow \mathsf{Mask}(\mathsf{pp}, k, \tau, \mathbf{0})$*. Send* c_b *to* $\mathcal{A}$*.*

Construction Based on LWR. We construct a 1-approximate key-homomorphic masking scheme based on the learning with rounding (LWR) assumption [5]. The construction is a slight modification to the almost seed homomorphic PRG based on LWR in [11].

Definition 5 (LWR [5])**.** *let* λ *be the security parameter,* $n = n(\lambda)$*,* $q = q(\lambda)$*,* $p = p(\lambda)$ *be integers. The* $LWR_{n,q,p}$ *assumption states that for any* $m = \mathrm{poly}(n)$ $A \leftarrow \mathbb{Z}_q^{m \times n}$*,* $\mathbf{s} \leftarrow \mathbb{Z}_q^n$*,* $\mathbf{u} \leftarrow \mathbb{Z}_q^m$*, the following indistinguishability holds:*

$$(A, \lfloor A \cdot \mathbf{s} \rfloor_p) \approx^c (A, \lfloor \mathbf{u} \rfloor_p),$$

where $\lfloor \cdot \rfloor_p$ *is the rounding function defined as* $\lfloor \cdot \rfloor_p : \mathbb{Z}_q \to \mathbb{Z}_p : x \mapsto \lfloor (p/q) \cdot x \rfloor$*.*

Construction 1 (key-homomorphic masking by LWR)**.**

– $\mathsf{Setup}(1^\lambda, \ell, p_m)$: *deterministically* choose a modulus q and dimension n such that LWR_{n,q,p_m} is assumed to be hard. Output $\mathsf{pp} = (\ell, p_m, q, n)$. The key space is $\mathcal{K} = \mathbb{Z}_q^n$, the message space, $\mathbb{Z}^\ell_{p_m}$, which is the same as the mask space.

- $\mathsf{KeyGen}(\mathsf{pp})$: sample a vector $\mathbf{s} \leftarrow \mathbb{Z}_q^n$, and output $k = \mathbf{s}$.
- $\mathsf{TagGen}(\mathsf{pp})$: sample a matrix $A \leftarrow \mathbb{Z}_q^{n \times \ell}$, and output $\tau = A$.
- $\mathsf{Mask}(\mathsf{pp}, k, \tau, \mathbf{m})$: parse the key and tag as $k, \tau = \mathbf{s}, A$. Output the masked message $\mathbf{c}_m = \lfloor A \cdot \mathbf{s} \rceil_{p_m} + \mathbf{m} \in \mathbb{Z}_{p_m}^{\ell}$.
- $\mathsf{UnMask}(\mathsf{pp}, \mathbf{c}_m, \mathbf{c}_0)$: output the message $\mathbf{m}^* = \mathbf{c}_m - \mathbf{c}_0 \in \mathbb{Z}_{p_m}^{\ell}$.
- $\mathsf{Eval}(\mathsf{pp}, L, \{\mathbf{c}_i\})$: parse L as d integer coefficients $u_1, \ldots, u_d$. Output the evaluated mask $\mathbf{c}_L = \sum_{i \in [d]} u_i \mathbf{c}_i \in \mathbb{Z}_{p_m}^{\ell}$.

The idea of the construction is simple. A masking key is an LWR secret $k = \mathbf{s}$, and a tag is a random LWR public matrix $\tau = A$. Given a masking key $\mathbf{s}$, a tag A, and a message $\mathbf{m}$ as inputs, the Mask algorithm hides the message $\mathbf{m}$ with a fresh LWR sample $\lfloor A \cdot \mathbf{s} \rceil_{p_m}$. We defer the proof of Lemma 1 to the full version.

Lemma 1. *Construction 1 is a* 1*-approximate key-homomorphic masking scheme under the* LWR_{n,q,p_m} *assumption.*

Choosing Parameters q, n**.** It is proved in [5] that under the Learning With Error (LWE) assumption with dimension n, modulus q, and any noise distribution bounded by B, the LWR assumption also holds with dimension n and moduli q, p_m such that $q \geq B p_m n^{\omega(1)}$. It's commonly believed that the LWE assumption holds for sufficiently large $B = \mathrm{poly}(n)$, and sub-exponential modulus-to-noise ratio $\alpha = q/B \leq 2^{\sqrt{n}}$. Therefore, given a message modulus $p_m \in \mathbb{N}$, it suffices to set $n = (\log p_m + \Omega(\lambda))^2$, and $q = B p_m n^{\log \lambda}$.

Extension to Ring LWR. The above scheme can also be instantiated using the Ring LWR assumption introduced together with LWR in [5]. We implement the more computationally efficient version with Ring LWR and present experiment data in Sect. 5.

Construction Based on DCR. Due to limited space, we defer our construction of an exact key-homomorphic masking scheme under the decisional composite residuosity (DCR) assumption to the full version.

3.2 Flat Secret Sharing

A threshold secret-sharing scheme with M parties normally has two algorithms $\mathsf{Share}, \mathsf{Recon}$, and is parameterized by privacy and reconstruction thresholds γ, ρ, where $0 < \gamma < \rho < 1$. Running Share on a secret value creates M shares. Running Recon on any subset of more than ρM shares recovers the secret. Any subset of less than γM shares contains no information about the secret.

Secret Sharing in Our Framework. Our framework uses the scheme in an unusual way. In the setup phase, the clients run Share to create shares of their masking keys. In the online phase, the server runs Recon *not* over the key shares, but homomorphically over empty masks created under the key shares. As long as Recon is a linear function, the key-homomorphism property (Definition 2) ensures that running Recon over the masks translates to over the underlying key

shares. The two thresholds γ, ρ guarantees the masking keys are hidden when at most γM clients are corrupted, and Recon succeeds when at least ρM clients are online.

This approach creates a technical challenge when the masking scheme has only approximate key-homomorphism (Definition 3). Namely, evaluating Recon homomorphically creates an additive noise, which grows with the magnitude of the coefficients of Recon. The noise then propagates into the aggregation result.

To help remove the noise, each client input is multiplied with a scaling factor Δ, set larger than the noise. To accommodate the factor Δ in the clients inputs, the message modulus of the masking scheme is in turn increased by $\log \Delta$ bits. This overhead motivates us to construct a secret-sharing scheme with small coefficients in Recon, which we call a *flat* secret sharing scheme.

Overview of Our Scheme. Our starting point is the linear secret sharing scheme [17] that has $0, 1$ coefficients. However, using the scheme has a prohibitive overhead: the *total* share size scales polynomially in the population M, namely $\Omega(M^{5.3})$.

A first attempt at reducing the share size is to run the scheme in a small committee, sampled during the setup phase. If client corruption and dropout happen independently to the committee sampling, then the fractions of corruption and dropout in the committee roughly equal the true fractions in the population. This is true in our framework, where the set of corrupted clients, and potential dropout clients are decided statically at the beginning.

That is, we add a Setup algorithm to the scheme, which samples a committee $Q \subseteq [M]$ at random. It can be shown that when the fractions $0 < \gamma < \rho < 1$ has a constant gap, a committee of size $O(\kappa)$ suffices, with a $O(2^{-\kappa})$ statistical error.

Running [17] as a blackbox with a committee of size $O(\kappa)$ reduces the total share size from $O(M^{5.3})$ to $O(\kappa^{5.3})$. But we are able to further improve it to $O(\kappa^2)$, with a $O(\kappa^2)$-size committee, by re-visiting the analysis of [26], and constructing a committee version of [17] in a non-blackbox way. We summarize the syntax of our committee-based scheme for some secret space $\mathcal{M}$ below.

- $\mathsf{Setup}(1^\kappa, M)$ takes as inputs the statistical security parameter κ, and the population size M. It outputs a committee Q of share holders, and public parameters pp.
- $\mathsf{Share}(\mathsf{pp}, s)$ outputs shares $\{s_j\}_{j \in Q}$ computed from $s \in \mathcal{M}$.
- $\mathsf{Recon}(\mathsf{pp}, W, \{s_j\}_{j \in W})$ takes as inputs a set W indicating which shares are received, and the set of shares $\{s_j\}_{j \in W}$. It outputs a recovered secret s^* or $\perp$.

Correctness and Security. Formally, we define the correctness requirements as follows.

Definition 6 (ρ-reconstruction)**.** *Let κ be the statistical security parameter. For all population size $M \in \mathbb{N}$, secret $s \in \mathcal{M}$, and subset $T \subseteq [M]$ with size $|T| > \rho M$*

the following holds.

$$\Pr\left[\begin{array}{l|l} \mathsf{Recon}(\mathsf{pp}, W, \{s_j\}_W) & (Q, \mathsf{pp}) \leftarrow \mathsf{Setup}(1^\kappa, M), \\ \quad = s & W = T \cap Q, \\ & \{s_j\}_Q \leftarrow \mathsf{Share}(\mathsf{pp}, s) \end{array}\right] \geq 1 - \mathrm{negl}(\kappa).$$

The usual security requires that, for any corruption set $C \subseteq [M]$ below the threshold, i.e. $|C| \leq \gamma M$, corrupted shares $\{s_j\}_{Q \cap C}$ contain no information about the secret s.

We need a stronger property (which implies the usual one) to prove security of our framework: given corrupted shares $\{s_j\}_{Q \cap C}$ of 0, there is algorithm Ext that "extents" them to a full set of shares $\{s_j\}_Q$ for any secret s. The shares $\{s_j\}_Q$ distribute statistically close to shares of s. This is analogous to the property that, given a corrupted subset of Shamir's shares, one can interpolate the rest of the shares to any secret s. We formalize this requirement as follows.

Definition 7 (γ-simulation-privacy)**.** *Let κ be the statistical security parameter. There exists an efficient deterministic algorithm Ext such that for all population size $M \in \mathbb{N}$, secret $s \in \mathcal{M}$, and subset $C \subseteq [M]$ with size $|C| < \gamma M$ the following two distributions are statistically close.*
They share the same public parameters $(Q, \mathsf{pp}) \leftarrow \mathsf{Setup}(1^\kappa, M)$.

1. *$\{s_j\}_Q$ is computed normally as $\{s_j\}_Q \leftarrow \mathsf{Share}(\mathsf{pp}, s)$.*
2. *$\{\tilde{s}_j\}_Q = \{s'_j\}_{Q \cap C} \cup \{\tilde{s}_j\}_{Q \cap \overline{C}}$ is computed by $\{s'_j\}_Q \leftarrow \mathsf{Share}(\mathsf{pp}, 0)$ and $\{\tilde{s}_j\}_{Q \cap \overline{C}} = \mathsf{Ext}(\mathsf{pp}, C, \{s'_j\}_{Q \cap C}, s)$.*

Flatness. As explained in "Secret Sharing in Our Framework", we require the Recon algorithm to have small coefficients as a linear function over the input shares. This minimizes the noise introduced by evaluating Recon homomorphically over empty masks. A similar situation arises in the security proof of our framework, where the simulator needs to evaluate Ext (Definition 7) homomorphically over noisy masks. We therefore additionally require Ext to have small coefficients as a linear function over the input shares and the secret. We summarize the above requirements as "flatness".

Definition 8 (flatness)**.** *Let κ be the statistical security parameter. A flat secret sharing scheme satisfies the following.*

- *The Recon algorithm, when not outputting $\perp$, can be written as a linear function over the input shares, with integer coefficients bounded by $O(1)$.*
- *The Ext algorithm can be written as a linear function over the input shares and the secret, with integer coefficients bounded by $O(\log \kappa)$.*

Construction Details. We start by recalling the result of [8] and [17], summarized in the following theorem.

Theorem 1 (formula to secret sharing [8,17]). *For secrets over $\mathcal{M} = \mathbb{Z}_q$ for any modulus q or $\mathcal{M} = \mathbb{Z}$, there exists an efficient algorithm that translates any monotone Boolean formula $f : \{0,1\}^M \to \{0,1\}$, over variables $x_1, \ldots, x_M$, of size $d = |f|$, to a pair of secret sharing algorithms $\mathsf{Share}_f, \mathsf{Recon}_f$ satisfy the following:*

- $\mathsf{Share}_f(s)$ *computes d share units, each corresponding to a literal in f. For each share holder $i \in [M]$, its share s_i consists of all units corresponding to x_i. $\mathsf{Share}_f(s)$ outputs the shares $\{s_i\}$.*
  If $\mathcal{M} = \mathbb{Z}_q$, each share unit is an element in $\mathbb{Z}_q$. If $\mathcal{M} = \mathbb{Z}$, with secrets bounded by B, each unit is an integer bounded by $B2^\kappa$.
- *For any subset $T \subseteq [M]$, let $\mathbf{a}_T \in \{0,1\}^M$ denote the assignment where $a_i = 1$ iff $i \in T$. For every subset of the shares $\{s_j\}_T$, reconstruction $\mathsf{Recon}(T, \{s_j\}_T)$ succeeds iff $f(\mathbf{a}_T) = 1$.*
  For any subset $\{s_j\}_C$ that fails to reconstruct, there exists a simulation algorithm Ext defined analogously to Definition 7.
- *The algorithms $\mathsf{Share}_f, \mathsf{Recon}_f$ satisfy "flatness" per Definition 8.*

With Theorem 1, constructing a flat secret sharing scheme for any access structure reduces to finding a corresponding formula f:

- Setup constructs a formula f as pp, and defines the committee Q as the set of distinct literals in f.
- $\mathsf{Share}, \mathsf{Recon}$ simply run $\mathsf{Share}_f, \mathsf{Recon}_f$ given by Theorem 1.

Below we first describe the result of [26], which shows the existence of a formula f_t, of size $O(M^{5.3})$, for any t-threshold function. (Note that for any $\gamma M < t < \rho M$, f_t satisfies our requirement.)

Construction 2 (t-threshold monotone Boolean formula [26]).
In [26], f_t (over M variables) is implicitly constructed through a formulae distribution F_t satisfying the following:

$$\forall \mathbf{a} \in \{0,1\}^M, \quad \Pr\left[f(\mathbf{a}) = \mathsf{Thresh}_t(\mathbf{a}) \mid f \leftarrow F_t\right] > 1 - 2^M, \tag{1}$$

where Thresh_t denotes the t-threshold function. Applying the union bound over all 2^M values for $\mathbf{a}$, we have

$$\Pr\left[\forall \mathbf{a} \in \{0,1\}^M,\ f(\mathbf{a}) = \mathsf{Thresh}_t(\mathbf{a}) \mid f \leftarrow F_t\right] > 0.$$

Hence, there exists a formula f_t in F_t that computes Thresh_t exactly.

Further, note that for any threshold $0 < t < M$, the function Thresh_t over M inputs is equivalent to $\mathsf{Thresh}_{M'/2}$ over $M' = M + D \leq 2M$ inputs, with $D \leq M$ dummy variables always set to 1 or 0, respectively for the case of $t < M/2$ or $t \geq M/2$. For technical reasons, we always choose M' to be odd. Therefore, it remains to construct a formulae distribution $F_{M/2}$, for any *odd* M.

The construction is recursive. In the base case, $F^{(0)}$ is defined as

$$F^{(0)} := \begin{cases} x_j \text{ for a uniform } j \xleftarrow{\$} [M] & \text{w/ prob. } p = 3 - \sqrt{5} \\ 0 & \text{w/ prob. } (1-p). \end{cases}$$

For $i \geq 1$, the formulae distribution F^i is defined inductively

$$F^{(i)} := (F_1^{(i-1)} \vee F_2^{(i-1)}) \wedge (F_3^{(i-1)} \vee F_4^{(i-1)}),$$

where $F_1^{(i-1)}, F_2^{(i-1)}, F_3^{(i-1)}, F_4^{(i-1)}$ are distributions independent and identical to $F^{(i-1)}$. It's shown that after $k = O(1) + 2.65 \log M$ recursion steps, the distribution $F_{M/2} = F^{(k)}$ satisfies Eq. 1.

Correctness and Efficiency of Construction 2. According to Eq. 1, we examine the probability that, for any assignment $\mathbf{a} \in \{0,1\}^M$, a sample $f^{(i)} \leftarrow F^{(i)}$ computes the *incorrect* result.

- When $\mathbf{a}$ has less than $M/2$ ones, $f^{(i)}(\mathbf{a})$ is supposed to output 0, but instead (incorrectly) outputs 1. Let $p_s^{(i)}$ denote this probability, i.e., $f^{(i)}(\mathbf{a}) = 1$. By construction, we have

$$p_s^{(i)} = \left(1 - (1 - p_s^{(i-1)})^2\right)^2. \tag{2}$$

- When $\mathbf{a}$ has at least $M/2$ ones, let $p_c^{(i)}$ denote the probability that $f^{(i)}(\mathbf{a})$ (incorrectly) outputs 0. Similarly, we have

$$p_c^{(i)} = 1 - (1 - \left(p_c^{(i-1)}\right)^2)^2. \tag{3}$$

By construction of $F^{(0)}$, and that M is odd, we also have

$$p_s^{(0)} < p(\frac{1}{2} - \frac{1}{2M}), \quad p_c^{(0)} \leq (1-p) + p(\frac{1}{2} - \frac{1}{2M}).$$

It remains to show that $p_s^{(k)}, p_c^{(k)} < 2^M$ for $k = O(1) + 2.65 \log M$, which follows from the technical claims below, which are taken directly from [26].

Claim 1 (phase 1)**.** *For the recurrence relations specified by Eq. 2, 3 with any initial values satisfying* $p_s^{(0)} < p/2 - p/(2M)$, $p_c^{(0)} < 1 - p/2 - p/(2M)$, *it holds that* $p_s^{(k_1)} \leq p/2 - \Omega(1)$, *and* $p_c^{(k_1)} \leq 1 - p/2 - \Omega(1)$ *for* $k_1 = 1.65 \log M$.

Claim 2 (phase 2)**.** *For the recurrence relations specified by Eq. 2, 3 with any initial values satisfying* $p_s^{(0)} < p/2 - \Omega(1)$, *and* $p_c^{(0)} < 1 - p/2 - \Omega(1)$, *it holds that* $p_s^{(k_2)}, p_c^{(k_2)} < 2^M$ *for* $k_2 = O(1) + \log M$.

Intuitively, a formula sampled from $F^{(0)}$ fails with probability close to (but less than) $p/2$ and $1 - p/2$ respectively in the two cases. Each recursive step "shifts" them further away from the starting points towards 0. Claim 1 shows that it takes $k_1 = O(\log M)$ steps to start at $\Theta(1/M)$-away and shift to $\Omega(1)$-away from the starting points. Claim 2 shows that it takes additional $k_2 = O(\log M)$ steps to shift exponentially close to 0.

Since each recursive step multiplies the formula size by 4, after $k = k_1 + k_2 = O(1) + 2.65 \log M$ steps, the formulas in $F^{(k)}$ has size $4^{O(1)+2.65 \log M} = O(M^{5.3})$.

Reducing the Size of Construction 2. Our first observation is instead of the formula f_t, we only need a formula $f_{\rho,\gamma}$ that 1) computes 1 if the inputs have $> \rho M$ ones, 2) computes 0 if the inputs have $< \gamma M$ ones, and 3) may otherwise compute either. We denote this (ρ, γ)-threshold function $\mathsf{Thresh}_{\rho,\gamma}$. A similar trick reduces computing $\mathsf{Thresh}_{\rho,\gamma}$ over M variables to $\mathsf{Thresh}_{1/2+\delta,1/2-\delta}$ over $M' \leq 2M$ variables for some constant fraction $\delta = (\rho - \gamma)/4$.

This observation allows us to calculate the initial failure probability for $f^{(0)} \leftarrow F^{(0)}$ differently from above.

- When $\mathbf{a}$ has less than $M(1/2 - \delta)$ ones, $f^{(0)}$ fails (i.e., computes 1) with probability $p_s^{(0)} < p(1/2 - \delta) < p/2 - \Omega(1)$.
- When $\mathbf{a}$ has more than $M(1/2 + \delta)$ ones, $f^{(0)}$ fails with probability $p_c^{(0)} < (1 - p) + p(1/2 + \delta) < 1 - p/2 - \Omega(1)$.

Since the initial values of $p_s^{(0)}, p_c^{(0)}$ already satisfies the condition for Claim 2, we indeed only need $k_2 = O(1) + \log M$ recursive steps! This observation already let us reduce the size of the formula from $4^{O(1)+2.56 \log M} = O(M^{5.3})$ to $4^{O(1)+\log M} = O(M^2)$.

Our second observation is that in the static corruption model, the set of corrupted and the reconstructing share holders C, T_i at each iteration i is fixed before the secret sharing Setup algorithm. Therefore, instead of finding an exact formula $f_{\rho,\gamma}$ that's correct on all assignments, it suffices to sample $f \leftarrow F_{\rho,\gamma}$ during Setup that's correct on the (poly(κ) many) fixed assignments $\mathbf{a}_C$ and $\mathbf{a}_{T_i}$.

In particular, we can avoid taking the union bound over 2^M values for $\mathbf{a}$, and only construct a distribution $F_{\rho,\gamma}$ (equivalently, $F_{1/2+\delta,1/2-\delta}$) such that

$$\forall \mathbf{a} \in \{0,1\}^M, \quad \Pr\left[f(\mathbf{a}) = \mathsf{Thresh}_{\rho,\gamma}(\mathbf{a}) \mid f \leftarrow F_{\rho,\gamma}\right] > 1 - 2^{\kappa}.$$

By Claim 2, we now only need $k_2' = O(1) + \log \kappa$ recursive steps, which further reduces the formula size to $O(\kappa^2)$!

To summarize, we obtain the following lemma.

Lemma 2 (flat secret sharing)**.** *For any population size $M \in \mathbb{N}$, constant fractions $0 < \gamma < \rho < 1$, integer modulus q and dimension ℓ, there exists a flat secret-sharing scheme $\mathsf{Setup}, \mathsf{Share}, \mathsf{Recon}$ for secrets space $\mathcal{M} = \mathbb{Z}_q^\ell$ or $\mathcal{M} = \mathbb{Z}^\ell$, with privacy and reconstruction thresholds γ, ρ. Furthermore,*

- *It has committee size $|Q| = O(\kappa^2)$, where the constant depends on the thresholds γ, ρ.*
- *The Recon algorithm, when written as a linear function, has $O(\kappa)$ non-zero coefficients, which are 1 or -1.*

Concrete Algorithm for Theorem 1. When sharing a secret according to a formula f, Share_f views f as a tree with AND, OR on the intermediate nodes, and literals x_i on the leaf nodes. It assigns a share to each node of this tree: i) Upon reaching an AND node, split the current share s into two additive shares of s, and assign them to the children. ii) Upon reaching an OR node, duplicate s and assign them to the children. iii) Upon reaching a literal x_i, assign s to share holder i. Reconstruction according to f follows a similar recursive algorithm.

4 The LERNA Framework

In this section, we describe our abstract secure aggregation protocol assuming the existence of the two technical tools introduced in Sect. 3:

- An ϵ-approximate key-homomorphic masking scheme $\mathsf{HM} = (\mathsf{HM.Setup}, \mathsf{KeyGen}, \mathsf{TagGen}, \mathsf{Mask}, \mathsf{UnMask}, \mathsf{Eval})$ setup properly with $\mathsf{HM.pp}$, specifying a message space $\mathbb{Z}_{p_m}^{\ell}$, mask space $\mathbb{Z}_q^{\ell}$ and key space $\mathcal{K}$.
- A flat secret sharing scheme $\mathsf{SS} = (\mathsf{SS.Setup}, \mathsf{Share}, \mathsf{Recon})$ for sharing the masking keys in the above key space $\mathcal{K}$.

The protocol additionally assumes a public key encryption scheme and two hash functions $\mathcal{H}_1, \mathcal{H}_2$ modeled as random oracles. We assume the hash functions $\mathcal{H}_1, \mathcal{H}_2$ output exactly the numbers of random bits required by the algorithms $\mathsf{SS.Setup}$, and TagGen.

The protocol runs with M clients $\{P_i\}$ and a single server S for T iterations. During each iteration $t \in [T]$, every client P_i obtains a fresh integer vector $\mathbf{x} \in \mathbb{Z}^{\ell}$ from a bounded range $[0, B_x]$. To avoid wrap-around in the aggregation results, we setup the masking scheme with a modulus lower bound $B_{\text{msg}} = \Delta M B_x$, where Δ is a message scaling factor introduced in the protocol.

The protocol is further parameterized by two thresholds $\gamma, \delta \in (0, 1)$, specifying the maximum fractions of corrupted clients and dropout clients, respectively, under the restriction that $\gamma + \delta < 1$. We set the privacy threshold of the secret sharing scheme to γ, and the reconstruction threshold to $\rho = 1 - \delta$.

In the online phase, the protocol uses a noise bound B_e and a message scaling factor Δ, which we specify in Sect. 4.4 for concrete instantiations under LWR.

4.1 The Semi-honest Protocol

We start with the simpler, semi-honest variant of the protocol, given in Fig. 1, and Fig. 2. We describe the additional steps to obtain the malicious variant next, and defer the more formal (in the UC framework) security proof for the malicious protocol to the full version.

Setup Phase. During the setup phase, the clients first agree on a small committee Q, computed using public common randomness $\mathbf{r}_1$. They each sample a secret masking key k_i, and secret share it to the committee Q, using the server

Setup Phase

Inputs to P_i: The session id sid, public keys of other clients, and public parameters of the masking scheme $\mathsf{HM.pp}$.

1. Each client P_i obtains common randomness $\mathbf{r}_1 = \mathcal{H}_1(\mathsf{sid})$ for sampling the committee $(Q, \mathsf{SS.pp}) \leftarrow \mathsf{SS.Setup}(; \mathbf{r}_1)$.
   Next, P_i samples a masking key $k_i \leftarrow \mathsf{KeyGen}(\mathsf{HM.pp})$ and secret shares it to the committee:
   $$\{k_j^i\}_{j \in Q} \leftarrow \mathsf{Share}(\mathsf{SS.pp}, k_i).$$
   P_i encrypts each share k_j^i with the public key of its target P_j as $\tilde{k}_j^i$, and sends $(\mathsf{sid}, i, j, \{\tilde{k}_j^i\}_{j \in Q})$ to the server S.
2. The server S receives the above encrypted shares from all M clients, and distributes them through messages $(\mathsf{sid}, \{\tilde{k}_j^i\}^{i \in [M]})$ to every committee member $P_j \in Q$.
3. Each committee member $P_j \in Q$ receives encrypted shares, decrypts them, and stores the plain shares $\{k_j^i\}^{i \in [M]}$.

Fig. 1. LERNA protocol for the setup phase.

to distribute those shares. To keep the shares secret from the server, the clients encrypt each share using the public key of its target share-holder.

Note that the clients only run the setup phase once, followed by T online phases. In each online phase, each client P_i uses the same masking key k_i to mask its fresh input vector $\mathbf{x}_i$. Reusing the masking key may seem like a privacy concern. To address this, we ensure that in each online phase, the clients sample a *fresh tag* τ used for computing the mask. The randomness of the tag τ protects the input vector $\mathbf{x}_i$, as long as the masking key remains secret.

Online Phase.

Step 1: Every client runs the key-homomorphic masking scheme $\mathsf{HM.Mask}$ to obtain a masked input vector $\mathbf{z}_i$, and sends it to the server S. It's important to note that key-homomorphism only holds for masks computed using the same tag τ. Therefore, the clients sample the tag using public common randomness $\mathbf{r}_2$.

Step 2: The server receives the masked input vectors $\{\mathbf{z}_i\}$ from the online clients, and replies the online set U to each committee member. Note that non-committee member clients don't need to send anything in the rest of the online phase.

Step 3: Every committee member P_j aggregates locally its shares of masking keys from the online set U to obtain an aggregated key share k_j^U, uses it to compute an "empty mask" as its reconstruction vector $\mathbf{w}_j$, and sends it to the server S.

Step 4: The server S receives reconstruction vectors $\{\mathbf{w}_j\}$ from the online committee members. It proceeds to locally recover the aggregation result.

First, it homomorphically aggregates the masked input vectors $\mathbf{z}_i$ to obtain $\mathbf{c}_{\text{sum}}$. By key-homomorphism, the vector $\mathbf{c}_{\text{sum}}$ approximately equals running

Online Phase: iteration $t = 1, ..., T$

Inputs to P_i: The session id sid, and an integer vector $\mathbf{x}_i \in \mathbb{Z}^\ell$.

1. Each client P_i obtains common randomness $\mathbf{r}_2 = \mathcal{H}_2(\mathsf{sid}, t)$ for sampling a tag $\tau = \mathsf{TagGen}(\mathsf{HM.pp}; \mathbf{r}_2)$, computes
$$\mathbf{z}_i \leftarrow \mathsf{Mask}(\mathsf{HM.pp}, k_i, \tau, \Delta \cdot \mathbf{x}_i),$$
and sends a message $(\mathsf{sid}, i, \mathbf{z}_i, t)$ to the server S.
2. The server S receives a masked input vector $\mathbf{z}_i$ from each online client, and records the set of dropout clients D. It computes the "online set" $U = [M] \setminus D$, and sends a message (sid, U, t) to every online committee member $P_j \in (Q \cap U)$.
3. Each online committee member P_j checks that $|U| > (1 - \delta)M$, and computes a reconstruction vector
$$\mathbf{w}_j = \mathsf{Mask}(\mathsf{HM.pp}, \textstyle\sum_{i \in U} k_j^i, \tau, \mathbf{0}) + \mathbf{e}_j,$$
where $\mathbf{e}_j \leftarrow [B_e]^\ell$ is a uniformly sampled noise from range B_e. If $|U|$ is too small, P_j sets $\mathbf{w}_j = \perp$. P_j sends a message $(\mathsf{sid}, j, \mathbf{w}_j, t)$ to the server S.
4. The server S receives a reconstruction vector $\mathbf{w}_j$ from every online committee member, ignoring $\perp$. It records the set of valid vectors W. S homomorphically sums over the masked inputs as $\mathbf{c}_{\text{sum}} = \mathsf{Eval}(\mathsf{HM.pp}, +, \{\mathbf{z}_i\}_{i \in U})$, and then homomorphically runs the Recon algorithm over the vectors $\{\mathbf{w}_i\}$
$$\mathbf{c}_0 = \mathsf{Eval}\big(\mathsf{HM.pp}, \mathsf{Recon}(\mathsf{SS.pp}, W, \cdot), \{\mathbf{w}_i\}\big).$$
If Recon aborts on the set W, then S outputs a message $(\mathtt{sid}, \perp, D, t)$. Otherwise, it uses $\mathbf{c}_0$ as the "empty mask" to recover $\mathbf{x}'_U \leftarrow \mathsf{UnMask}(\mathsf{HM.pp}, \mathbf{c}_{\text{sum}}, \mathbf{c}_0)$, and rounds $\mathbf{x}'_U$ by Δ to obtain $\mathbf{x}_U$. It outputs a message $(\mathsf{sid}, \mathbf{x}_U, D, t)$.

Fig. 2. LERNA protocol for the online phase (semi-honest).

$\mathsf{HM.Mask}$ on the scaled aggregation result $\mathbf{x}'_U = \Delta \cdot \sum_U \mathbf{x}_i$ under the key $k_U = \sum_U k_i$. It remains to obtain an "empty mask" $\mathbf{c}_0$ under the same key k_U, with which the server can recover the scaled aggregation result $\mathbf{x}'_U$, and then the actual aggregation result $\mathbf{x}_U = \lfloor \mathbf{x}'_U / \Delta \rceil$ through rounding.

To obtain the empty mask $\mathbf{c}_0$ under the key k_U, the server homomorphically runs the algorithm $\mathsf{SS.Recon}$ over the reconstruction vectors $\mathbf{w}_j$. By key-homomorphism, the result indeed approximately equals $\mathbf{c}_0$. Note that approximate key-homomorphism causes some errors in the recovered result $\mathbf{x}'_U$. But we set the scaling factor Δ sufficiently large to make sure such errors are removed by the rounding step.

Alternative to the PKI Setup. The setup phase of our protocol requires the clients to encrypt their secret shares under the public keys of the target shareholders. For simplicity, our protocol assumes a public key infrastructure (PKI), and that each client enters the setup phase knowing every other client's public key.

An alternative approach is to let the clients run pairwise key agreement at the beginning of the setup phase, as described in the "Communication Model" paragraph (Sect. 2).

Committee Members and Non-members. Note that in each online phase, non-member clients only have one task: send masked input vectors to the server. The rest of the reconstruction steps are handled by committee member clients.

This separation of responsibility suggests an alternative aggregation model, where during each phase, only a small, potentially random, subset among the non-member clients is required to provide inputs. Our protocol can be adapted straightforwardly to guarantee: as long as not too many committee members drop out during the session, the server can securely compute the aggregation result. This scenario can be useful for stochastic federated learning algorithms that benefit from a large input population, but only learns from a random subset at each iteration.

4.2 Correctness of LERNA

Below, we illustrate correctness by proving Lemma 3. A formal functionality definition and security proof in the UC framework is in the full version.

Lemma 3 (correctness)**.** *If less than δM clients dropout in an online session t, then the server outputs the correct aggregation result with overwhelming probability in the semi-honest setting.*

Proof (sketch). Looking at the reconstruction step (online step 4), we first argue that the aggregated mask $\mathbf{c}_{\text{sum}}$ is distributed close to a mask over the aggregation result. By ϵ-approximate key homomorphism (Definition 3), we have

$$\|\mathbf{c}_{sum} - \mathsf{Mask}(\mathsf{HM.pp}, \sum_{i\in U} k_i, \tau, \Delta \sum_{i\in U} \mathbf{x}_i)\|_\infty \leq \epsilon M.$$

For the UnMask algorithm to work correctly, we need to argue the reconstructed mask $\mathbf{c}_0$ is distributed close to an empty mask under the key $\sum_{i\in U} k_i$. To this end, we first argue that the Recon algorithm succeeds over the shares from the set W with overwhelming probability. By assumption, online set U computed by the server at the online step 2 has size $|U| > (1-\delta)M$. Therefore, all online committee members send reconstruction vectors $\mathbf{w}_j$ at online step 3. Let the online set at online step 3 be $U' \subseteq U$. The set of valid reconstruction vectors W equals $W = U' \cap Q$. By assumption, we have $|U'| > (1-\delta)M$. Therefore, by $(1-\delta)$-reconstruction, the algorithm Recon indeed succeeds with overwhelming probability.

By flatness (Definition 8), the function $\mathsf{Recon}(\mathsf{SS.pp}, W, \cdot)$ is linear with $O(1)$ coefficients. Therefore, by ϵ-approximate key homomorphism, we have

$$\begin{aligned}&\|\mathbf{c}_0 - \mathsf{Mask}(\mathsf{HM.pp}, \sum_{i\in U} \underbrace{\mathsf{Recon}(\mathsf{SS.pp}, W, \{k_j^i\}_{j\in W})}_{k_i}, \tau, \mathbf{0})\|_\infty \\ &\leq O(\epsilon B_e |Q|),\end{aligned}$$

where B_e is the bound on the noises $\mathbf{e}_j$ in the vectors $\mathbf{w}_j$.

Finally, we conclude that the UnMask algorithm on masks $\mathbf{c}_{\text{sum}}$ and $\mathbf{c}_0$ returns a noisy result $\mathbf{x}'_U = \Delta \sum_U \mathbf{x}_i + \mathbf{e}$, where the noise has entries bounded by $\|\mathbf{e}\|_\infty = O(\epsilon(M + B_e|Q|))$. As long as the message scaling factor Δ is sufficiently large $\Delta \geq 2\|\mathbf{e}\|_\infty$, the server indeed recovers the correct result through rounding by Δ. □

4.3 Achieving Malicious Security

To achieve malicious security, we keep the setup phase (Fig. 1) unchanged, and only modify the online phase (Fig. 2) starting from step 2. The modifications follow similar ideas to prior work [7,10]. The modified online phase is given in Fig. 3, where the changes are highlighted in blue.

Online Phase: iteration $t = 1, ..., T$

Inputs to P_i: The session id sid, public keys of other clients, and an integer vector $\mathbf{x}_i \in \mathbb{Z}^\ell$.

2. The server S records the dropout set D and online set $U = [M] \setminus D$ as in Figure 2, and sends a message (sid, U, t) to every online committee member $P_j \in (Q \cap U)$. Additionally, it sends a short hash of U, h_U to every online client.
3. Each client P_i receives a hash h_U from the server S, and sends its signature $\sigma_i(h_U)$ to S.
4. The server S receives a signature from every online client, and sends the set of valid signatures $\{\sigma_i(U)\}$ to every online committee member P_j.
5. Each online committee member P_j checks that at least $(1+\gamma)M/2$ signatures over the hash h_U are valid. If there are not enough valid signatures, it sets the reconstruction vector $\mathbf{w}_j = \perp$. Otherwise, it proceeds as in Figure 2 step 3 to compute the vector $\mathbf{w}_j$, and sends it to the server S.
6. The server S receives a reconstruction vector $\mathbf{w}_j$ from every online committee member, ignoring invalid vectors like $\perp$. It proceeds as in Figure 2 step 4 to recover the result $\mathbf{x}_U$. In case of any failed step, it outputs a message $(\mathsf{sid}, \perp, D, t)$.

Fig. 3. LERNA protocol for the online phase (malicious). (Color figure online)

To see why we need the additional steps in the malicious setting, consider the following corrupted server. Recall that in the semi-honest online protocol, the server sends an online set U to online committee members to recover an aggregation result $\mathbf{x}_U = \sum_U \mathbf{x}_i$. A corrupted server instead sends different online sets, $U \neq U'$, to two subsets of online committee members. As long as both subsets are large enough, the correctness of the semi-honest protocol guarantees the successful recovery of both results $\mathbf{x}_U$ and $\mathbf{x}_{U'}$ by the server. This obviously violates our security definition, which requires only a single sum of honest inputs is leaked in each online phase.

The additional steps 3–4 in Fig. 3 roughly ask each client, including corrupted ones, to "vote" on an online set U by signing a hash h_U. The server collects those signatures as unforgeable votes and sends them to the committee members. The threshold in step 5 is set such that at most one online set U^* can have enough votes. Therefore, the above attack is prevented.

Preventing Abort Attacks. While setting the threshold for valid signatures in Step 5 to $(1+\gamma)M/2$ guarantees that at most one online set U^* has enough votes, it creates an opportunity for malicious clients to abort the protocol, even when the server is honest, by not sending enough valid signatures. To avoid this issue, we need enough honest clients so that their signatures alone are enough for the threshold. Restricting the corruption and dropout threshold γ, δ such that $(3\gamma + 2\delta) < 1$ suffices.

Claim 3. *Assuming* $(3\gamma + 2\delta) < 1$*, and the server is honest, then every honest committee member always collects at least* $(1+\gamma)M/2$ *valid signatures in Step 5.*

Proof. By the assumption, there are at least $(1-\gamma-\delta)M$ honest clients in each iteration that remain online, and will send a valid signature in Step 3 on the hash h_U received from an honest server. Calculation shows $(1-\gamma-\delta)M \geq (1-\gamma)M/2$ iff $1 \geq (3\gamma + 2\delta)$. □

By the above claim, an honest server is guaranteed to receive non-$\perp$ reconstruction messages from all honest online committee members in Step 6. By ρ-reconstruction ($\rho = 1-\delta$) of the secret sharing, the server succeeds in computing the empty mask $\mathbf{c}_0$.

Finally, the server may still abort if $\mathsf{UnMask}(\mathsf{HM.pp}, \mathbf{c}_{\mathrm{sum}}, \mathbf{c}_0)$ fails. However, in our LWR masking scheme (Construction 1), the UnMask algorithm simply computes a subtraction modulo p_m, which always succeeds.

Overhead of the Malicious Protocol. As highlighted in Fig. 3, the communication and computation overhead of the malicious variant consists of the server sending valid signatures $\{\sigma_i(U)\}$ in step 4, and each committee member verifying those signatures in step 5, respectively.

For ease of presentation, the variant shown in Fig. 3 requires every client to send a signature in step 3. However, it can be shown that at the cost of a $O(2^{-\kappa})$ statistical error in privacy, only committee members need to send signatures. Note that the number of signatures is at most the committee size $|Q| = O(\kappa^2)$, which is independent of the number of clients M, or the input dimension ℓ. Therefore, when M or ℓ is large, sending and checking those signatures incur only negligible communication and computation overheads over the semi-honest variant.

4.4 Instantiation Under LWR

Concretely, we instantiate the LERNA protocol with the 1-approximate homomorphic masking scheme based on LWR in Construction 1.

We set the noise bound $B_e = O(\log(\kappa)M2^\kappa)$, which is required for security. (See the full version for security proofs.) where κ is the statistical security parameter. We set the message scaling factor $\Delta = O(M + B_e|Q|)$ as required by the correctness proof of Lemma 3, where $|Q| = O(\kappa^2)$ is the committee size of the flat secret sharing scheme, as described in Sect. 3.2. Under these settings, our protocol sets up the LWR-based masking scheme with message modulus (which is the same as the mask modulus) $p_m = \Delta M B_x$, which has bit length $\log p_m < O(1) + 3\log\kappa + \kappa + 2\log M + \log B_x$.

The LWR-based masking scheme has keyspace $\mathcal{K} = \mathbb{Z}_q^n$, where the dimension n and modulus q is chosen such that LWR_{n,q,p_m} is assumed to be hard. We therefore instantiate a flat secret-sharing scheme with secret space $\mathcal{M} = \mathcal{K} = \mathbb{Z}_q^n$.

We present communication and computation efficiency analysis for the LWR instantiation in the full version.

5 Experimental Evaluation

We benchmark the concrete efficiency of the LERNA framework by implementing the semi-honest protocol instantiated under the (Ring) LWR assumption (cf. Sect. 4.4 for a description).

As our baseline, we compare our protocol design with the semi-honest protocol from [7], adapted naturally to the multi-session setting. In particular, the baseline server uses the setup phase to randomly sample a communication graph, and inform each client of its set of neighbors. Baseline clients re-use the same communication graph throughout the following online phases.

Our benchmarks clearly highlight the lightweight server computation during each online phase.

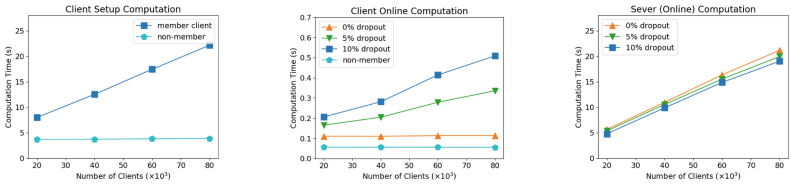
5.1 Implementation Details

Our prototypes are implemented in Python. The protocol simulations are run locally, using the ABIDES simulation framework [12]. Our implementations use the following libraries for heavy computations:

- SEAL [24] and PySEAL [2] for polynomial arithmetics required by Ring LWR.
- Gmpy2[3] for large integer arithmetics.
- M2Crypto[4] as an interface to AES for implementing a PRG and a random oracle.
- PyNaCl[5] for public key encryption and key-agreement.

Setting Parameters. In the LERNA framework, we need to set two security parameters, $\lambda = 128, \kappa = 40$. Computationally secure primitives (e.g., encryption

[2] https://github.com/Lab41/PySEAL.
[3] https://gmpy2.readthedocs.io/.
[4] https://m2crypto.readthedocs.io/.
[5] https://pynacl.readthedocs.io/.



(a) LERNA client setup computation.

(b) LERNA client online computation.

(c) LERNA server online computation.

Fig. 4. LERNA computation time vs. number of clients (M), with fixed input dimension $\ell = 10K$.

and the masking scheme) are set to have $\lambda = 128$ bits of security, and statistically secure primitives (e.g., the flat secret sharing scheme) are set to have $\kappa = 40$ bits of security. The concrete committee size equals $|Q| = 2^{14} = 16384$ for $\kappa = 40$.

In our prototype, the message modulus p_m for the (Ring) LWR based key-homomorphic masking scheme is set as described in Sect. 4.4, which ranges from 142 to 145 bits in our benchmark settings. We set the RLWR dimension to be 2^{11}, and the modulus to $q = p_m \cdot 2^{254}$ to guarantee at least 128 bits of security, according to the hardness estimator[6] of [1].

In the baseline prototype, we set the field size for Shamir's secret sharing to be a 257 bit prime, because the secrets are 256 bit curves used in key-agreement. To set the neighborhood size k and privacy threshold t of Shamir's secret sharing, we follow Theorem 3.10 in [7] (section 3.5). In our settings where the number of parties ranges from $M = 400, \ldots, 80K$, the neighborhood size ranges from $k = 109, \ldots, 126$, and the privacy threshold ranges from $t = 55, \ldots, 63$ to achieve $2^{-\kappa} = 2^{-40}$ statistical error.

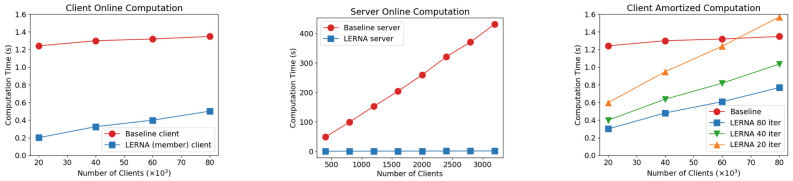
5.2 Benchmarks

Our benchmarks are run on a desktop machine with 32 Gigabyte of memory and with a single core CPU speed 3.9 GHz. Our prototype implementations do not take advantage of multiple cores. For computation time measurements, we report an average over 10 experiment runs.

Computation Efficiency. We first benchmark the computation time of our LERNA prototype with increasing numbers of clients $M = 20K \ldots, 80K$. We run the prototype with $\ell = 10K$ dimension inputs vector with random entries from $[0, 2^{64}]$, and fix the corruption threshold at $\gamma = 10\%$. In Fig. 4a, 4b, and 4c, we respectively plot our client runtime during the setup and the online phases, and our server runtime during the online phase. Comparing Fig. 4a and 4b, we observe that the setup phase is much heavier compared to the online phases.

In Fig. 4b, and 4c, we observe that the dropout rate affects the computation time of both committee member clients and the server. This is because our

[6] Running code provided at https://lwe-estimator.readthedocs.io.

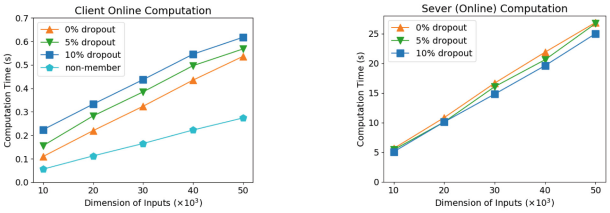


(a) Client online computation.

(b) Server online computation.

(c) Client amortized computation.

Fig. 5. Computation time comparison between LERNA and [7], with fixed input dimension $\ell = 10K$, and dropout rate $\gamma = 10\%$. (b) compares the server computation at a smaller number of clients $M = 400, \ldots, 3200$ due to the high cost of the baseline server. (c) compares the amortized computation time of a single setup phase plus 20/40/80 online phases. Since the baseline client has negligible computation during setup, its amortized time equals that shown in (a).



(a) LERNA client online computation.

(b) LERNA server online computation.

Fig. 6. LERNA computation time vs. input dimension (ℓ), with fixed number of clients $M = 20K$. The plot for client setup computation is omitted, as it doesn't depend on ℓ.

committee member needs to aggregate masking key shares over the dropout set, which becomes larger both under higher dropout rates and with a larger number of clients. Our server similarly aggregates masked input vectors over the online set, which becomes smaller under higher dropout rates.

In the full version, we give more detailed numbers about the running time of different components of our protocols.

We next benchmark the computation time of our protocol with increasing input dimensions $\ell = 10K \ldots, 50K$. We run the prototype with $M = 20K$ clients, and fix the corruption threshold again at $\gamma = 10\%$. In Fig. 6a, and Fig. 6b, we respectively plot our client and server during the online phase. Since the clients and the server during the setup phase are independent of input dimensions, we omit their plots.

Communication Efficiency. In Table 1 we report the communication sizes of our client with increasing input dimensions $\ell = 10K \ldots, 50K$. The server communication can be deduced as the sum of all clients. Hence we omit its

table. We run the prototype with $M = 20K$ clients, and input entries from $[0, 2^{64}]$. We fix the corruption threshold and the dropout rate both at 10%.

Table 1. Client communication sizes.

Phase	$\ell = 10K$	$\ell = 30K$	$\ell = 50K$
Non-member setup	2.00 (GB)	2.00 (GB)	2.00 (GB)
Member setup	4.44 (GB)	4.44 (GB)	4.44 (GB)
Non-member online	0.18 (MB)	0.54 (MB)	0.91 (MB)
Member online	0.37 (MB)	1.09 (MB)	1.82 (MB)

The total offline communication of our clients is indeed heavy, as reported in Table 1. Each client sends encrypted shares of its masking key to the server. Due to the large Ring LWR dimension (2048) and modulus (∼400 bits), this phase requires large communication (2 GB) from each client. Each committee member additionally receives the encrypted shares from the clients.

Thankfully, the entire offline phase doesn't need to be synchronized, which eases the bandwidth requirement. If needed, each client can send a share of its masking key to a committee member one at a time.

Comparing with the Baseline. To compare with the baseline, we run both prototypes with $10K$ dimension inputs vectors with random entries from $[0, 2^{64}]$. We fix the corruption rate and the dropout rate at $\gamma = 10\%$.

As discussed in the introduction, we assume a statically corrupted set of clients throughout the repeated T sessions. A larger T, means a stronger assumption on the staticness and the fraction of corruption. On the flip side, since our protocol enjoys a re-usable setup across T sessions, a larger T gives better efficiency. In comparing with the baseline, we not only compare the computation time of each online iteration (Fig. 5a, 5b), but also the amortized time over different settings of T (Fig. 5c). The client computation and communication cost of running our setup phase (, where a fresh committee is formed and secret masking keys are shared,) are shown in Fig. 4a and Table 1. The server costs of setup for our server and for the baseline solution are negligible. Hence we omit reporting them here.

From Fig. 5a, we observe that even our slower committee member client runs faster than the baseline during each online iteration for $M = 20K$ to $M = 80K$. As expected, its running time grows faster with M than the baseline because our committee member needs to aggregate masking key shares over the dropout set. If the dropout rate is a non-zero constant, as set in our experiment, then the committee client's work grows linearly in M. In comparison, the computation of the baseline depends linearly in its neighborhood size in the communication graph, which is $O(\log M)$.

In Fig. 5c, we compare the clients' amortized running time (showing the heavy member clients for LERNA) of a single setup phase followed by $T = 20/40/80$

online iterations. Since the baseline client has negligible computation during setup, its amortized time equals its online computation time, which doesn't change with T. We observe an advantage, even for the member clients, over the baseline when amortized over more than $T = 40$ online sessions. For example, at $M = 80K$, the total client computation time of 40 LERNA iterations equals $22 + 0.5 \cdot 40 = 42(s)$. The total time of 40 baseline iterations is at least $1.2 \cdot 40 = 48(s)$, according to the plot.

In Fig. 5b we are only able to compare the server's performance at moderate numbers of clients $M = 400 \ldots 3200$, because the baseline server runs too long when M reaches $10K$. But this is enough to illustrate LERNA's advantage (concretely, more than $100\times$) in server computation times.

Comparing with SASH+ [20]. As mentioned in "Related Work", the protocol SASH+ from [20] reduces aggregating ℓ-dimension inputs to aggregating n-dimensional homomorphic PRG seeds, where n is the LWR dimension. SASH+ then runs [7] for the latter. Asymptotically, SASH+ reduces the computation cost of [20] from $\widetilde{O}(\kappa^2 + \kappa\ell)$ to $\tilde{O}(\kappa^2 + \kappa n + \ell)$ for the clients, and from $\widetilde{O}(\kappa M\ell + \kappa^2 M)$ to $\widetilde{O}(\kappa Mn + \kappa^2 M + \ell M)$ for the server. We optimistically estimate that SASH+ reduces the computation cost of [7] by a factor of (ℓ/n).

In our benchmarks, $\ell = 10K$, and the LWR dimension $n = 2048$. We estimate the server and client computational costs of SASH+ to be 5x smaller than [7] (in reality, the improvement is smaller due to other computation steps that remain constant). Under this estimation, we observe that the LERNA server (Fig. 5b) and non-committee member clients (Fig. 4b) still significantly outperforms the SASH+ server and SASH+ clients. However, the cost of a LERNA committee member (Fig. 5a) becomes comparable to (when M is relatively small e.g. $20K$) or slower than (when M is larger) a SASH+ client.

Acknowledgement. This paper was prepared in part for information purposes by the Artificial Intelligence Research group and AlgoCRYPT CoE of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

Hanjun Li was supported by a NSF grant CNS-2026774 and a Cisco Research Award.

Huijia Lin was supported by NSF grants CNS-1936825 (CAREER), CNS-2026774, a JP Morgan AI Research Award, a Cisco Research Award, and a Simons Collaboration on the Theory of Algorithmic Fairness.

Stefano Tessaro was supported in part by NSF grants CNS-2026774, CNS-2154174, a JP Morgan Faculty Award, a CISCO Faculty Award, and a gift from Microsoft.

References

1. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046 (2015). https://eprint.iacr.org/2015/046
2. Apple, Google: Exposure notification privacy-preserving analytics (ENPA) (2021). https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf
3. Asghar, M.R., Dán, G., Miorandi, D., Chlamtac, I.: Smart meter data privacy: a survey. IEEE Commun. Surv. Tutorials **19**(4), 2820–2835 (2017). https://doi.org/10.1109/COMST.2017.2720195
4. Ball, M., Çakan, A., Malkin, T.: Linear threshold secret-sharing with binary reconstruction. In: Tessaro, S. (ed.) 2nd Conference on Information-Theoretic Cryptography, ITC 2021, 23–26 July 2021, Virtual Conference. LIPIcs, vol. 199, pp. 12:1–12:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITC.2021.12
5. Banerjee, A., Peikert, C., Rosen, A.: Pseudorandom functions and lattices. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 719–737. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_42
6. Bell, J., et al.: Acorn: input validation for secure aggregation. Cryptology ePrint Archive, Paper 2022/1461 (2022). https://eprint.iacr.org/2022/1461
7. Bell, J.H., Bonawitz, K.A., Gascón, A., Lepoint, T., Raykova, M.: Secure single-server aggregation with (poly)logarithmic overhead. In: Ligatti, J., Ou, X., Katz, J., Vigna, G. (eds.) ACM CCS 2020, pp. 1253–1269. ACM Press (2020). https://doi.org/10.1145/3372297.3417885
8. Benaloh, J., Leichter, J.: Generalized secret sharing and monotone functions. In: Goldwasser, S. (ed.) CRYPTO 1988. LNCS, vol. 403, pp. 27–35. Springer, New York (1990). https://doi.org/10.1007/0-387-34799-2_3
9. Bonawitz, K.A., et al.: Towards federated learning at scale: system design. In: Talwalkar, A., Smith, V., Zaharia, M. (eds.) Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, 31 March– 2 April 2019. mlsys.org (2019). https://proceedings.mlsys.org/book/271.pdf
10. Bonawitz, K., et al.: Practical secure aggregation for privacy-preserving machine learning. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017, pp. 1175–1191. ACM Press (2017). https://doi.org/10.1145/3133956.3133982
11. Boneh, D., Lewi, K., Montgomery, H., Raghunathan, A.: Key homomorphic PRFs and their applications. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 410–428. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_23
12. Byrd, D., Hybinette, M., Balch, T.H.: ABIDES: towards high-fidelity multi-agent market simulation. In: Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2020, Miami, FL, USA, 15–17 June 2020, pp. 11–22 (2020). https://doi.org/10.1145/3384441.3395986
13. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: 42nd FOCS, pp. 136–145. IEEE Computer Society Press (2001). https://doi.org/10.1109/SFCS.2001.959888
14. Corrigan-Gibbs, H.: Privacy-preserving firefox telemetry with prio (2020). https://rwc.iacr.org/2020/slides/Gibbs.pdf

15. Corrigan-Gibbs, H., Boneh, D.: Prio: private, robust, and scalable computation of aggregate statistics. In: Akella, A., Howell, J. (eds.) 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, 27–29 March 2017, pp. 259–282. USENIX Association (2017). https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs
16. Damgård, I., Jurik, M.: A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In: Kim, K. (ed.) PKC 2001. LNCS, vol. 1992, pp. 119–136. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44586-2_9
17. Damgård, I., Thorbek, R.: Linear integer secret sharing and distributed exponentiation. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 75–90. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_6
18. Gilboa, N., Ishai, Y.: Distributed point functions and their applications. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 640–658. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_35
19. Guo, Y., Polychroniadou, A., Shi, E., Byrd, D., Balch, T.: Microfedml: privacy preserving federated learning for small weights. Cryptology ePrint Archive, Paper 2022/714 (2022). https://eprint.iacr.org/2022/714
20. Liu, Z., Chen, S., Ye, J., Fan, J., Li, H., Li, X.: SASH: efficient secure aggregation based on SHPRG for federated learning. In: Cussens, J., Zhang, K. (eds.) Uncertainty in Artificial Intelligence, Proceedings of the Thirty-Eighth Conference on Uncertainty in Artificial Intelligence, UAI 2022, 1–5 August 2022, Eindhoven, The Netherlands. Proceedings of Machine Learning Research, vol. 180, pp. 1243–1252. PMLR (2022). https://proceedings.mlr.press/v180/liu22c.html
21. Özdemir, S., Xiao, Y.: Secure data aggregation in wireless sensor networks: a comprehensive overview. Comput. Netw. **53**(12), 2022–2037 (2009). https://doi.org/10.1016/j.comnet.2009.02.023
22. Patton, C., Barnes, R., Schoppmann, P.: Verifiable Distributed Aggregation Functions. Internet-Draft draft-patton-cfrg-vdaf-01, Internet Engineering Task Force (2022). https://datatracker.ietf.org/doc/html/draft-patton-cfrg-vdaf-01
23. Rieke, N., et al.: The future of digital health with federated learning. CoRR abs/2003.08119 (2020). https://arxiv.org/abs/2003.08119
24. Microsoft SEAL (release 4.0). Microsoft Research, Redmond, WA (2022). https://github.com/Microsoft/SEAL
25. So, J., Ali, R.E., Guler, B., Jiao, J., Avestimehr, S.: Securing secure aggregation: mitigating multi-round privacy leakage in federated learning. arXiv preprint arXiv:2106.03328 (2021)
26. Valiant, L.G.: Short monotone formulae for the majority function. J. Algorithms **5**(3), 363–366 (1984). https://doi.org/10.1016/0196-6774(84)90016-6