Dordis: Efficient Federated Learning with Dropout-Resilient Differential Privacy

Zhifeng Jiang
HKUST
zjiangaj@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

Ruichuan Chen
Nokia Bell Labs
ruichuan.chen@gmail.com

Abstract

Federated learning (FL) is increasingly deployed among multiple clients to train a shared model over decentralized data. To address privacy concerns, FL systems need to safeguard the clients' data from disclosure during training and control data leakage through trained models when exposed to untrusted domains. Distributed differential privacy (DP) offers an appealing solution in this regard as it achieves a balanced tradeoff between privacy and utility without a trusted server. However, existing distributed DP mechanisms are impractical in the presence of *client dropout*, resulting in poor privacy guarantees or degraded training accuracy. In addition, these mechanisms suffer from severe efficiency issues.

We present Dordis, a distributed differentially private FL framework that is highly efficient and resilient to client dropout. Specifically, we develop a novel 'add-then-remove' scheme that enforces a required noise level precisely in each training round, even if some sampled clients drop out. This ensures that the privacy budget is utilized prudently, despite unpredictable client dynamics. To boost performance, Dordis operates as a distributed parallel architecture via encapsulating the communication and computation operations into stages. It automatically divides the global model aggregation into several chunk-aggregation tasks and pipelines them for optimal speedup. Large-scale deployment evaluations demonstrate that Dordis efficiently handles client dropout in various realistic FL scenarios, achieving the optimal privacy-utility tradeoff and accelerating training by up to 2.4× compared to existing solutions.

CCS Concepts: • **Computing methodologies** → **Supervised learning**; • **Security and privacy** → *Database and storage security*.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629559

Keywords: Federated Learning, Distributed Differential Privacy, Client Dropout, Secure Aggregation, Pipeline

ACM Reference Format:
Zhifeng Jiang, Wei Wang, and Ruichuan Chen. 2024. Dordis: Efficient Federated Learning with Dropout-Resilient Differential Privacy. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3629559

1 Introduction

Federated learning (FL) [42, 53] enables collaborative training of a shared model among multiple clients (e.g., mobile and edge devices) under the orchestration of a central server. In scenarios where the number of clients is large (e.g., millions of mobile devices), the server dynamically *samples* a small subset of clients to participate in each training round [14]. These clients download the global model from the server, compute local updates using private data, and upload these updates to the server for global aggregation. Throughout the training, no client's data is exposed directly. FL has been deployed in various domains, enabling a multitude of privacy-sensitive AI applications [28, 47, 51, 59, 62, 82].

However, solely keeping client data on the device is inadequate for preserving data privacy. Recent work has shown that sensitive data can still be exposed through message exchanges in FL training [27, 81, 85, 86]. It is also possible to infer a client's data from the trained models [17, 57, 70, 74] by exploiting their ability to memorize information [17, 74].

Current FL systems often use *differential privacy* (DP) [24] to perturb the aggregate model update in each round, limiting the disclosure of individual clients' data throughout training. Among the three typical DP models (i.e., central [54, 65], local [63], and distributed DP [5, 40, 75]), *distributed DP* is the most appealing for FL as it: 1) assumes *no trusted server* (in contrast to central DP), and 2) imposes *minimum noise* given a privacy budget (unlike local DP), causing little utility loss to the trained models. Specifically, given a global privacy budget that must not be exceeded, the system first calculates the minimum random noise required in each round. Then, in each round, every sampled client adds a small portion of the required noise to its local update. The aggregate update at the server is thus perturbed by *exactly* the minimum required noise. In distributed DP, local updates are aggregated using secure aggregation, which ensures that the (untrusted) server learns only the aggregate result, not individual updates.

Existing distributed DP mechanisms, however, face two practical challenges when deployed in real-world FL systems. First, the privacy guarantee of distributed DP can be compromised in the presence of *client dropout*, which can occur at any time due to network errors, low battery, or changes in eligibility, as frequently observed in production [14, 44, 83]. For dropped clients, their noise contributions are missing in the aggregate update, leading to insufficient DP protection and consuming more privacy budget than initially allocated in each round [40]. This exhausts the privacy budget quickly, resulting in early termination of training and significant model utility loss (§2.3.1). As a quick solution, proactively increasing the amount of noise used may not achieve the optimal privacy-utility tradeoff without strong expertise in client dynamics and learning tasks.

Second, aside from the privacy issue caused by client dropout, the secure aggregation protocols (e.g., SecAgg [15]) employed in distributed DP present severe efficiency challenges. This is due to the multi-round communications and heavy cryptographic computations involved, which can consume up to 97% of the time of each training round, as observed in real deployments (§2.3.2). Recent attempts [12, 39, 72, 73] to improve the asymptotic complexity sacrifice desired properties (e.g., malicious security or dropout tolerance), and/or have limited practical gains.

In this paper, we present Dordis, an efficient FL framework that enables **d**rop**o**ut-**r**esilient **dis**tributed DP in FL. Dordis addresses both the privacy and efficiency issues of distributed DP in FL systems with two key contributions. First, to ensure that the aggregate model update is perturbed with exactly the minimum required noise regardless of client dropout, we devise a novel *add-then-remove* scheme named `XNoise` (§3). `XNoise` initially lets each selected client to add excessive noise to its local update. After aggregation, the server removes part of the excessive noise that exceeds the minimum noise requirement, based on the actual client participation. To cope with the potential failures in executing `XNoise`, we consolidate its security using efficient cryptographic primitives. `XNoise` is proven to preserve the privacy of honest clients, even in the presence of a *malicious* server colluding with a small subset of other clients.

Second, to expedite the execution of distributed DP, we run Dordis as a *distributed parallel architecture* to overlap computation- and communication-intensive operations (e.g., data encoding and transmission) (§4). To enable a generic design, Dordis first abstracts the distributed DP workflow into a sequence of *stages* with different dominant system resources. By dividing the global aggregation task into several *chunk*-aggregation task, Dordis allows pipeline parallelism by scheduling them to run different stages concurrently. With a realistic performance model and profiling technique, Dordis can identify the optimal pipeline configuration by solving an optimization problem for *maximum speedup*.

We implemented Dordis as an end-to-end system with generic designs that support a wide range of distributed DP protocols (§5).[1] We deployed Dordis in a real distributed environment that features data and hardware heterogeneity of client devices (§6). Our evaluations across various FL training tasks show that the necessary noise for aggregated updates in Dordis can be enforced without impairing model utility in the presence of client dropout. Moreover, Dordis's pipeline execution speeds up the baseline systems with different secure aggregation protocols by up to 2.4×, without reducing their security properties.

2 Background and Motivation

2.1 Scenario

Federated Learning. Federated learning (FL) enables a large number of clients (e.g., millions of mobile and edge devices) to collaboratively build a global model without sharing local data [42, 53]. In FL, a (logically) centralized server maintains the global model and orchestrates the iterative training. At the beginning of each training round, the server randomly samples a subset of available clients as participants [14]. The sampled clients perform local training to the downloaded global model using their private data and report only the model updates to the server. The server collects the updates from participants until a certain deadline, and aggregates these updates. It then uses the aggregate update (e.g., FedAvg [53]) to refine the global model.

Threat Model. Although client data is not directly exposed in the FL process, a large body of research has shown that it is still possible to reveal sensitive client data from individual updates or trained models via *data reconstruction* or *membership inference* attacks. For example, an adversary can accurately reconstruct a client's training data from its gradient updates [27, 81, 85, 86]; an adversary can also infer from a trained language model whether a client's private text is in the training set [17, 57, 70, 74].

We aim to control the exposure of honest clients' data against the above-mentioned attacks. Following the Secure Multi-Party Computation (SMPC) literature [26], we target both the *semi-honest* setting (where all parties faithfully follow the protocol) and the *malicious* setting (where the adversary can deviate arbitrarily from the protocol). In both settings, the adversary is eager to pry on honest clients' data, and may collude with the server and a fraction of sampled clients to boost its advantages. We assume mild client collusion because, in real deployments, the number of clients is usually large, making it hard for an adversary to corrupt a large fraction of them. For example, at the scale of the Apple ecosystem (over 2 billion active devices [8]), even compromising 1% would mean about 20 million nodes. Consequently, the chance of having many colluded clients sampled by the server is tiny, if it follows the agreed-upon sampling

[1] Dordis is available at https://github.com/SamuelGong/Dordis.

algorithm. Even if the server behaves maliciously in client sampling, it can still be refrained from impersonating or simulating arbitrarily many clients given the use of a public-key infrastructure and a signature scheme, with which honest clients can verify the source of their received messages and detect such behaviors (§3.3). In addition, we can also prevent the server from cherry-picking colluded clients when verifiable random functions (VRFs) [22, 56] are deployed to ensure the correctness of random client sampling (§7).

2.2 Differential Privacy

The reconstruction and membership attacks work by finding clients' data that make the observed messages (e.g., individual updates or trained models) more likely. Differential privacy (DP) [21, 23, 24] effectively prevents these attacks by ensuring that no specific client participation can noticeably increase the likelihood of such observed messages. This guarantee is captured by two parameters, ϵ and δ [24]. Given any neighboring training sets D and $D^{'}$ that differ only in the inclusion of a single client's data, the aggregation procedure f is (ϵ, δ)-differentially private if, for any set of output R, we have $\Pr[f(D) \in R] \leq e^{\epsilon} \cdot \Pr[f(D') \in R] + \delta$.

In other words, a change in a client's participation yields at most a multiplicative change of e^{ϵ} in the probability of any output, except with a probability δ. Intuitively, smaller ϵ and δ indicate a stronger privacy guarantee. A key property of DP is composition which states that the process of running (ϵ_1, δ_1)-DP and (ϵ_2, δ_2)-DP computations on the same dataset is $(\epsilon_1+\epsilon_2, \delta_1+\delta_2)$-DP. This allows one to account for the privacy loss resulting from a sequence of DP-computed outputs, such as the release of multiple aggregate updates in FL.

Central DP and Local DP. One way to apply DP in FL is to let the server add DP-compliant noise to the aggregate update, i.e., the *central DP* scheme [42]. However, the server must be trusted as it has access to the (unprotected) aggregate update. While the server may establish a trusted execution environment (TEE) [9, 31] with hardware support, it is still vulnerable to various attacks, e.g., side-channel attacks [18, 78]. An alternative DP scheme is *local DP*, in which each sampled client adds DP noise to perturb its local update. As long as the noise added by a client is sufficient for a DP guarantee on its own, its privacy is preserved regardless of the behavior of other clients or the server. This, however, results in excessive accumulated noise in the aggregate update, significantly harming the model utility [40].

Distributed DP. Compared to central and local DP, *distributed DP* offers an appealing solution in FL scenarios as it: 1) requires no trusted server, and 2) imposes minimum noise given a privacy budget. In distributed DP, a privacy goal is specified as a global privacy budget (ϵ_G, δ_G), which can be viewed as a non-replenishable resource that is consumed by each release of an aggregate update. Ideally, by the time when the training completes, the remaining privacy budget should be zero, so as to meet the privacy goal at the expense of minimum DP noise and model utility loss.

This requires the system to perform *offline noise planning* ahead of time to determine the minimum required noise that should be added to the aggregate update in each training round to control the privacy loss. The system then proceeds to *online noise enforcement*. In each training round, it evenly splits the noise adding task to all sampled clients. Each of them slightly perturbs its update by adding an even share of the minimum required noise. The clients then mask their updates and send them to the server using the secure aggregation (SecAgg) protocol [15], which ensures that the server learns nothing but the aggregate update that is perturbed with exactly the minimum required noise. Note that, besides the commonly-used SecAgg, distributed DP can also be implemented using alternative approaches such as secure shuffling [13, 19, 25]. In this paper, we focus on the approaches using SecAgg, given their popularity in FL.

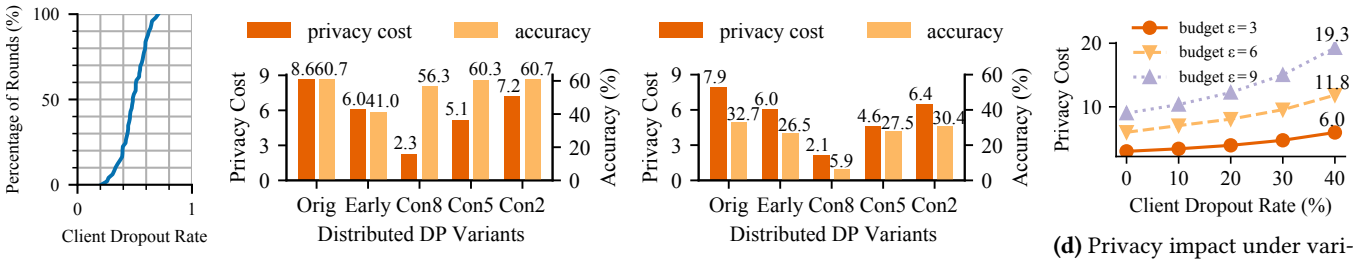
2.3 Practical Issues of Distributed DP

While distributed DP can achieve an appealing privacy-utility tradeoff, its deployment in real world has significant issues.

2.3.1 Privacy Issue Caused by Client Dropout. In FL training, *client dropout* can occur anytime, e.g., due to low battery, poor connection, or switching to a metered network. The prevalence of client dropout, which has been widely observed in real-world systems [14, 44, 83], raises a severe privacy issue in distributed DP. Specifically, if clients drop out after being sampled, without their noise contributions, the total noise added to the aggregate update falls below the minimum required level. This leads to increased data exposure that forces the system to consume more privacy budget than planned for each round. Without the ability to deterministically enforce the consumption of privacy budget, the system may fail to incentivize clients to join or comply with privacy regulations [60, 64, 79].

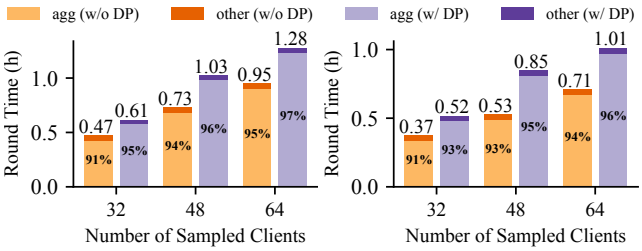
To illustrate this issue, we analyze a realistic FL task. We run two FL testbeds in which 100 clients jointly train a ResNet-18 [3] model over the CIFAR-10 dataset and CIFAR-100 dataset [43] for 150 and 300 rounds, respectively. To emulate the dynamics of clients, we use a large-scale user behavior dataset spanning 136k mobile devices [83] and extract 100 volatile users. We sample 16 clients to train in each round and observe great dynamics in their availability, as shown in Figure 1a. In this case, the original distributed DP training over CIFAR10 (resp. CIFAR100) with a global privacy budget $\epsilon_G = 6$ ends up consuming an ϵ of 8.6 (resp. 7.9) at the 150th (resp. 300th) round due to the missing noise from the dropped clients (see `Orig` in Figure 1b and 1c).

Naive Solutions and Limitations. One solution to this issue is to stop the training early when the privacy budget runs out (`Early`). However, this inevitably harms the model utility. As shown in Figure 1b and 1c, `Early` reduces the model utility



(a) Client dynamics. **(b)** Privacy v.s. utility in CIFAR-10. **(c)** Privacy v.s. utility in CIFAR-100. **(d)** Privacy impact under various client dropout rates.

Figure 1. Privacy impact of client dropout.



(a) SecAgg with per-round dropout rate being 10%. **(b)** SecAgg+ with per-round dropout rate being 10%.

Figure 2. Impact of secure aggregation on training efficiency.

by 19-29% compared to non-private training. Another solution is to make a conservative estimation on the per-round dropout rate during offline noise planning. However, a good estimation that balances the privacy-utility tradeoff requires accurate information on client dynamics and learning tasks. For instance, without accurate information on client dynamics (e.g., Figure 1a), one common practice is to overestimate the dropout severity (e.g., 80% as in Con8) which leads to suboptimal model utility, while underestimating it (e.g., 20% as in Con2) results in excessive privacy budget consumption. Even given a priori knowledge of client dynamics, the trade-off is still hard to navigate without trial-and-error experiments due to its task-specific nature. For example, while guessing 50% as the per-round dropout rate (Con5) yields a near-optimal privacy-utility tradeoff for CIFAR-10, it causes noticeable utility degradation (16%) for CIFAR-100.

Impact of Client Dropout Rate. To further relate privacy violation to dropout severity, we let the clients randomly drop with a configurable rate after being sampled. Figure 1d shows that, for the CIFAR-10 testbed, as the dropout rate increases, more clients' data gets exposed during training, leading to a larger privacy deficit regardless of the budget.

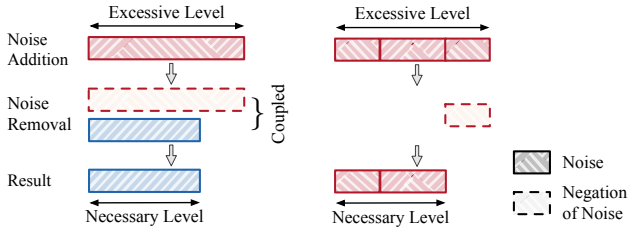
2.3.2 Performance Issue Caused by Secure Aggregation. Besides the privacy issue caused by client dropout, the SecAgg algorithm [15] used in distributed DP creates a severe performance issue. Specifically, to ensure the server learns no individual update from any client but the aggregate update only, SecAgg lets clients synchronize secret keys and use them to generate zero-sum masks (a detailed description of SecAgg is embedded in Figure 5). This involves extensive use of pairwise masking and secret sharing, incurring high complexity in computation and communication.

To quantify the performance impact of SecAgg, we refer to Figure 2a which shows the breakdown of the average runtime of one training round in the previous experiments with varying numbers of sampled clients. For comparison, we also run the experiment with SecAgg but add no DP noise to the aggregate update. In all experiments, the cost of SecAgg dominates, accounting for 86-91% of the training time, while SecAgg with DP features a slightly more serious bottleneck than that without DP. Furthermore, the dominance of SecAgg is accentuated at scale.

Existing Solutions and Limitations. There have been active studies on improving the performance of secure aggregation, but they all have significant limitations. For example, one guarantee provided by SecAgg is input privacy against malicious adversaries, while TurboAgg [72], FastSecAgg [39], and LightSecAgg [73] only handle a semi-honest adversary. Moreover, their improved complexity comes at the cost of degraded dropout tolerance [32, 49], with their communication cost still being high in FL practice [52]. Among the follow-up works of SecAgg, SecAgg+ [12] is the state-of-the-art which improves the asymptotic complexity with a slight compromise on security and robustness. Yet, as Figure 2b implies, a further improvement is still desired given the consistent dominance of SecAgg+ on the training time.

3 Dropout-Resilient Noise Enforcement

To tackle the privacy issue mentioned in §2.3.1, we first formalize the noise enforcement problem under client dropout, and present the technical intuition to address this problem (§3.1). We then describe a novel 'add-then-remove' noise enforcement approach that realizes this intuition (§3.2) with security consolidation in real deployments (§3.3), followed by a security analysis (§3.4). Without loss of generality, we assume that the random noise distribution $\chi(\sigma^2)$ used in DP is *closed under summation* w.r.t. the variance σ^2. That is, given two independent noises $X_1 \sim \chi(\sigma_1^2)$ and $X_2 \sim \chi(\sigma_2^2)$,



(a) Rebasing. **(b)** Decomposition.

Figure 3. Two 'add-then-remove' approaches.

we have $X_1 \pm X_2 \sim \chi(\sigma_1^2 + \sigma_2^2)$. For example, both Gaussian and Skellam [5] distributions exhibit this property.

3.1 Technical Intuition

We start with a formal description of the original noise addition process used in distributed DP, denoted as `Orig`.

Definition 1 (`Orig`). *Given the set of sampled clients U and the target noise level σ_*^2 in a certain round,* `Orig` *lets each client $c_j \in U$ perturb its update Δ_j by adding noise $\boldsymbol{n}_j \sim \chi(\sigma_*^2/|U|)$ and upload the result $\tilde{\Delta}_j = \Delta_j + \boldsymbol{n}_j$ to the server for aggregation.*

As described in §2.3.1, the problem with `Orig` is that when some clients drop after being sampled, their noise contributions are missing; thus, the eventual noise aggregated at the server will be insufficient. One potential fix is to let the server add back the missing noise contributed by the dropped clients [40]. However, this is not viable under our threat model (§2.1) in which the server can be part of the adversary who can infer clients' data from the insufficiently perturbed aggregate result with unbounded advantages (semi-honest) and/or even omit the noise addition task (malicious).

Add-Then-Remove Noise Enforcement. To ensure that the aggregate noise never goes inefficient and always lands at the minimum required level, we design an 'add-then-remove' noise enforcement scheme: 1) each sampled client first adds a higher-than-required amount of noise to its model update, rendering an overly perturbed aggregate update despite client dropout, and 2) the server removes the excessive part of the aggregate noise based on the actual dropout outcome. This scheme can be realized by two possible approaches:

- *Rebasing*: During noise addition, each sampled client adds its noise share $\boldsymbol{n}_o$ to the local update, and sends the noisy update as a whole to the server. To facilitate noise removal, each surviving client computes the newly-required noise $\boldsymbol{n}_u$ based on the actual client dropout outcome, and subtracts the original noise share $\boldsymbol{n}_o$. To ensure that the noise removal is privacy-preserving, only '$\boldsymbol{n}_u - \boldsymbol{n}_o$' is transmitted to the server and added to the aggregate update (Figure 3a). This approach was adopted by [10].
- *Decomposition:* Instead of adding noise as a whole, each sampled client decomposes its noise share into multiple additive components that can be added separately to the client's local update. For privacy-preserving noise removal, each surviving client only sends the noise components that are over the newly-required amount to the server for subtracting them from the aggregate (Figure 3b).

Comparison. One difference between the two approaches lies in their *communication efficiency*. In FL, a DP noise is a sequence of pseudo-random numbers (PRNs) of the same length (e.g., millions to billions) as the model, and can be uniquely generated via feeding a seed (e.g., 20 bytes) into a PRN generator. During noise removal, 'decomposition' allows each surviving client to send the relevant seeds to the server for it to generate each noise component that needs to be removed. However, 'rebasing' requires each surviving client to generate and send the updated noise $\boldsymbol{n}_u - \boldsymbol{n}_o$ as a whole to conceal the two individual noises. Otherwise, the server can use them to reconstruct the noise-free aggregated update. This results in poor efficiency as the communication cost of noise removal increases prohibitively with the ever-growing model size. In Section 6.3, we compare the scalability of the two approaches in communication.

Another difference lies in their *robustness*. In reality, surviving clients can also drop out in the middle of noise removal. Missing their noise seeds, the server cannot fully remove the excessive noise added to the aggregate update. To tackle this issue, 'decomposition' can efficiently back up each noise component before model aggregation by secret-sharing its seed across clients (e.g., via the Shamir's scheme [69]). Such a scheme, however, does not apply to 'rebasing' as the updated noise to transmit can neither be generated with a seed nor determined before aggregation.

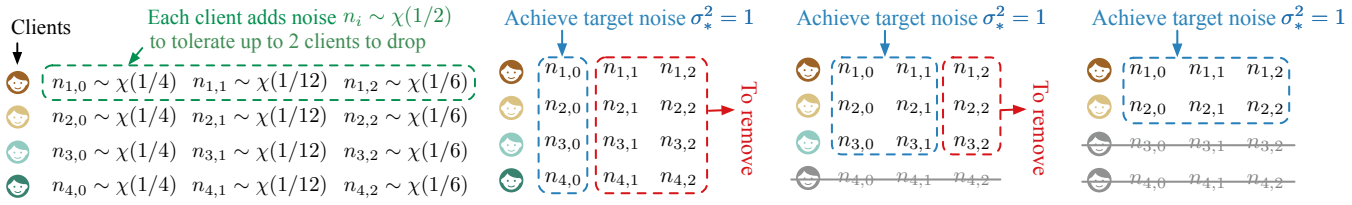
Challenges. Due to the poor efficiency and robustness of the 'rebasing' approach, we opt to instantiate the 'add-then-remove' noise enforcement scheme with 'decomposition' and tackle two technical challenges in its design:

- Given various dropout outcomes of a training round, how to decompose a client's noise share to accommodate every possible requirement during noise removal (§3.2)?
- In real deployments, how to make the noise enforcement approach secure, preferably in an efficient way (§3.3)?

3.2 Add-Then-Remove with Noise Decomposition

We start with how much noise a sampled client should add. Without loss of generality, we assume that the system's *tolerance* to client dropout is a configurable parameter. Let U be the set of sampled clients in a certain training round, among which the system can tolerate up to T dropouts.

Noise Addition and Removal. Let σ_*^2 be the target noise level in each round. To meet this noise level even in the worst case, each client in Dordis adds an excessive noise at



(a) Noise decomposition. **(b)** No client drops. **(c)** One client drops. **(d)** Two clients drop.

Figure 4. An illustration of how the 'add-then-remove' approach with noise decomposition deals with client dropout precisely.

the level of $\frac{\sigma_*^2}{|U|-T}$. In doing so, even if there are T clients dropped after being sampled, the total noise contributed by the surviving $|U| - T$ clients is still sufficient at the target level.

On the other hand, when fewer than T clients dropped after being sampled, the aggregate noise exceeds the target level and part of it needs to be removed for model utility. Let $D \subset U$ denote the set of clients dropped after being sampled, where $|D| \leq T$. The amount of excessive noise that should be removed by the server is:

$$l_{\text{ex}} = \underbrace{(|U| - |D|)\frac{\sigma_*^2}{|U| - T}}_{\text{Actual noise level}} - \sigma_*^2 = \frac{T - |D|}{|U| - T}\sigma_*^2. \tag{1}$$

Dordis evenly distributes the noise removal task across surviving clients, therefore, each of them needs to help the server remove the noise of level:

$$l'_{\text{ex}} = \frac{l_{\text{ex}}}{|U| - |D|} = \sigma_*^2\Big(\frac{1}{|U| - T} - \frac{1}{|U| - |D|}\Big). \tag{2}$$

Noise Decomposition for Precise Control. Equation (2) indicates that the noise to be removed by a surviving client decreases when the number of dropped clients increases. Given this monotonicity, each client in Dordis can carefully decompose its added noise share into multiple additive components, and remove some of such noise components when needed for the precise control of noise level.

For example, consider a scenario where the number of sampled clients $|U| = 4$, the dropout tolerance $T = 2$, and the target noise level $\sigma_*^2 = 1$. To enforce this target even if 2 clients drop out, the noise added to each client's update should be of the level 1/2. Moreover, as shown in Figure 4a, such a noise can be added as 3 separate components of level 1/4, 1/12, and 1/6, respectively, then one can accommodate all possible dropout outcomes within the tolerance by subtracting a subset of the added components to precisely remove the excessive noise. To be exact, if no client drops, i.e., $|D| = 0$, each surviving client removes $l'_{ex} = 1/12 + 1/6$ (Figure 4b); if one client drops, i.e., $|D| = 1$, each surviving client removes $l'_{ex} = 1/6$ (Figure 4c); if two clients drop, i.e., $|D| = 2$, each surviving client removes $l'_{ex} = 0$ (Figure 4d).

To be general, Dordis lets each client $c_i \in U$ decompose the added noise $\boldsymbol{n}_i \sim \chi(\frac{\sigma_*^2}{|U|-T})$ into $T+1$ components, i.e., $\boldsymbol{n}_i = \sum_{k=0}^{T} \boldsymbol{n}_{i,k}$ where $\boldsymbol{n}_{i,0} \sim \chi(\frac{\sigma_*^2}{|U|})$ and $\boldsymbol{n}_{i,k} \sim \chi(\frac{\sigma_*^2}{(|U|-k+1)(|U|-k)})$ for $k = 1, 2, \cdots, T$. These noise components are constructed in a way that when there are $|D|$ clients dropped after being sampled, the noise components $\boldsymbol{n}_{i,k}$ contributed by the surviving clients $c_i \in U \setminus D$ with the index $k > |D|$ become excessive and should be removed. One can verify that the aggregate of these removed components is exactly l_{ex}, i.e., $\sum_{c_i \in U \setminus D} \sum_{k=|D|+1}^{T} \boldsymbol{n}_{i,k} \sim \chi(l_{\text{ex}})$. We formalize the noise enforcement process for precise noise control as `XNoise`.

Definition 2 (`XNoise`). *In each training round, a sampled client $c_i \in U$ adds the intended excessive noise to its update Δ_i and sends the perturbed result $\tilde{\Delta}_i = \Delta_i + \sum_{k=0}^{T} \boldsymbol{n}_{i,k}$ to the server. Among these sampled clients, a subset D has dropped where $|D| \leq T$. The server calculates the aggregate update $\tilde{\Delta} = \sum_{c_i \in U \setminus D} \tilde{\Delta}_i$, and then removes some excessive noise components contributed by the surviving clients (known as* survivals*) to precisely enforce the target noise level, i.e., $\tilde{\Delta} - \sum_{c_i \in U \setminus D} \sum_{k=|D|+1}^{T} \boldsymbol{n}_{i,k}$.*

Dropout-Resilient Noise Removal with Secret Sharing. As described in §3.1, our noise decomposition design allows clients to transmit the seeds that are used to generate the requested noise components instead of those components in noise removal, greatly reducing the communication overhead. To further make this process dropout-resilient, we use Shamir's secret sharing scheme [69] for seed bookkeeping: each sampled client secretly shares with others the seeds it uses to generate local noise components before the secure aggregation takes place.

As such, to recover a local noise component during noise removal, the server first directly consults the related client on the corresponding seed. If the client drops out before reporting the seed, the server then initiates one additional communication round to collect the secret shares of the seed from all available clients. The server can recover the seed provided that the number of responding clients in this communication round exceeds a certain threshold τ specified by the secret sharing scheme [69].

Given all the above, a faithful execution of XNoise strictly enforces the target noise level, as established by Theorem 1. The proof is given in the full version of this paper [37].

Theorem 1 (Correctness). XNoise *ensures the noise level in the aggregate update is exactly σ_*^2, regardless of the dropout outcome as long as $|D| \leq T$, i.e., the number of dropped clients does not exceed the dropout tolerance.*

3.3 Security Consolidation with Optimized Practice

We next describe how we deploy XNoise in unsecure environments with optimized implementation.

Establishment of Secure Channels across Clients. For a client to be able to share a secret with another client as desired in §3.2, they first establish a secure channel. To achieve this over a server-mediated network, Dordis has them conduct key agreement via the Diffie-Hellman protocol [55] to establish the shared secret key for encrypting the subsequent communication. Furthermore, when a channel needs to be authenticated (under the malicious threat model), either end of the channel has to sign its messages for the other end to verify its identity with a public key infrastructure (PKI).

Prevention from Understating Dropout. As characterized by Equation (1), the less severe the client dropout is, the more noise the server removes. While a semi-honest server faithfully runs the protocol and never reports a smaller number of dropped clients than the actual one, a malicious server may *understate* the dropout severity for removing more noise than needed and obtaining an insufficiently perturbed aggregate. In the worst case, the malicious server can reduce the aggregate noise level to $(1 - T/|U|)\sigma^*$, e.g., only 40% of the target noise remains given the dropout tolerance set as 60% of the sampled clients. To detect whether a malicious server understates client dropout, Dordis lets clients verify the broadcasted dropout outcome with the use of a PKI and a standard UF-CMA[2] signature scheme **SIG**:

- Before uploading its perturbed local update $\tilde{\Delta}_i$, each client i signs the current round number R with its signing key d_i^{SK} and produces a signature $\omega_i' \leftarrow \textbf{SIG.sign}(d_i^{SK}, R)$. ω_i' is sent along with $\tilde{\Delta}_i$ to the server.
- When broadcasting the dropout outcome D, the server also broadcasts the set $\{j, \omega_j'\}_{j \in P}$, which contains all the signatures it has received (P denoted as the related clients).
- After receiving D and $\{j, \omega_j'\}_{j \in P}$, each client i verifies that: 1) all signatures are correct, i.e., $\textbf{SIG.ver}(d_j^{PK}, R, \omega_j') = 1$ for all $j \in P$, and 2) they agree with the broadcasted dropout outcome, i.e., $P = U \setminus D$ (otherwise aborts).

Intuitively, for the server to pretend that a client j survives, it has to forge that client's signature on the current round number, which is computationally infeasible if the client in fact dropped out, given the security of the signature scheme.

Optimization via Integration with Secure Aggregation. The aforementioned security-related secure channel establishment and the dropout outcome verification both induce $O(|U|)$ cost to each sampled client in computation and communication, and $O(|U|^2)$ cost to the server in communication. On the other hand, secure aggregation, the other indispensable component in the distributed DP workflow, often has instantiated similar primitives for correctness and security of its execution [12, 15]. We thus repurpose the existing security infrastructure to reduce the implementation complexity and improve the runtime efficiency.

To exemplify, Figure 5 details how we integrate XNoise with SecAgg [15] for reusing the secure channels across clients and correct broadcast on dropout outcome. SecAgg is instantiated with a public key infrastructure (PKI), the Diffie-Hellman key agreement [55] **KA** protocol composed with a secure hash function, the Shamir's t-out-of-n secret sharing scheme [69] **SS**, an IND-CPA (Indistinguishability against Chosen-Plaintext Attacks) and INT-CTXT (Integrity of Ciphertext) authenticated encryption scheme **AE**, a UF-CMA signature scheme **SIG**, and a secure pseudorandom generator **PRG**. For a detailed explanation of the cryptographic primitives employed by SecAgg, we refer the reader to the original paper [15]. While we opted to implement XNoise by resuing SecAgg's infrastructure for improved complexity and efficiency, we emphasize that XNoise is self-contained and complementary to secure aggregation protocols.

Handling Mild Collusion. The server can collude with a subset of clients under semi-honest and malicious threat models. In its strongest form, they collude from the very beginning of the protocol and pool their views all the time. As the collusion scale is presumably mild (justified in §2.1), honest clients can use a slightly increased amount of local noise to handle such collusion without utility loss. For example, given a collusion tolerance $T_C \approx 0.01|U|$, instead of adding noise of level $\frac{\sigma_*^2}{|U|-T}$, each honest client adds noise of level $\frac{\sigma_*^2}{|U|-T} \cdot \frac{t}{t-T_C}$ where t is the threshold as used in SecAgg. In doing so, a collusion within the tolerance T_C will not yield an insufficiently perturbed aggregate (§3.4).

It is important to mention that in the malicious setting with mild collusion, Dordis no longer enforces the minimum necessary noise but instead introduces a noise inflation factor of $\frac{t}{t-T_C}$. Fortunately, given that t is intentionally much larger than T_C,[3], the inflation factor is only slightly greater than 1. It should be noted, however, that this approach alone is insufficient to address the privacy leakage caused by dropout without any loss in utility. We anticipate that dropout could be on a much larger scale than collusion, potentially reaching the same magnitude as t (§2.3.1).

[2]Unforgability against Chosen-Message Attacks.

[3]The feasible range of t is $(0.5|U|, |U|]$ in the malicious setting [15].

The SecAgg Protocol integrated with XNoise

- **Setup**:
  - All parties are given the current round index r, the security parameter η, the number of users $|U|$ and a threshold for SecAgg t, honestly generated $pp \leftarrow \mathbf{KA.gen}(\eta)$, parameters m and R such that $\mathbb{Z}_R^m$ is the space from which inputs are sampled, and a field $\mathbb{F}$ to be used for secret sharing and noise samping, a noise distribution χ, the target central noise level σ_*^2, a dropout tolerance T and a collusion tolerance T_C for XNoise. All users also have a private authenticated channel with the server.
  - *User* u:
    - Have an input vector Δ_u, noises $\boldsymbol{n}_{u,0} \overset{g_{u,0}}{\sim} \chi\left(\frac{\sigma_*^2}{|U|} \cdot \frac{t}{t-T_C}\right)$, $\boldsymbol{n}_{u,k} \overset{g_{u,k}}{\sim} \chi\left(\frac{\sigma_*^2}{(|U|-k+1)(|U|-k)} \cdot \frac{t}{t-T_C}\right)$ $(k \in [1,T])$ where $g_{u,k} \leftarrow \mathbb{F}$ for all k.
    - Add to Δ_u a series of noises and produce $\tilde{\Delta}_u$: $\tilde{\Delta}_u = \Delta_u + \sum_{0 \le k \le T} \boldsymbol{n}_{u,k}$.
    - *[Receive signing key d_u^{SK} from the trusted third party, together with verification keys d_v^{PK} bound to each identity v.]*
- **Stage 0 (AdvertiseKeys)**:
  - *User* u:
    - Generate key pairs $(c_u^{PK}, c_u^{SK}) \leftarrow \mathbf{KA.gen}(pp)$, $(s_u^{PK}, s_u^{SK}) \leftarrow \mathbf{KA.gen}(pp)$, *[and generate $\omega_u \leftarrow \boldsymbol{SIG.sign}(d_u^{SK}, c_u^{PK} \| s_u^{PK}) = 1$]*.
    - Send $(c_u^{PK} \| s_u^{PK} [\| \omega_u])$ to the server (through the private authenticated channel) and move to next round.
  - *Server*:
    - Collect at least t messages from individual users in the previous round (denote with U_1 this set of users). Otherwise, abort.
    - Broadcast to all users in U_1 the list $\{(v, c_v^{PK}, s_v^{PK} [, \omega_v])\}_{v \in U_1}$.
- **Stage 1 (ShareKeys)**:
  - *User* u:
    - Receive the list $\{(v, c_v^{PK}, s_v^{PK} [, \omega_v])\}_{v \in U_1}$ broadcasted by the server. Assert that $|U_1| > t$, that all the public keys pairs are different, *[and that $\forall v \in U_1$, $\boldsymbol{SIG.ver}(d_v^{PK}, c_v^{PK} \| s_v^{PK}, \omega_v) = 1$]*.
    - Sample a random element $b_u \leftarrow \mathbb{F}$ (to be used as a seed for a **PRG**).
    - Generate t-out-of-$|U_1|$ shares of s_u^{SK}: $\{(v, s_{u,v}^{SK})\}_{v \in U_1} \leftarrow \mathbf{SS.share}(s_u^{SK}, t, U_1)$ and of b_u: $\{(v, b_{u,v})\}_{v \in U_1} \leftarrow \mathbf{SS.share}(b_u, t, U_1)$.
    - Generate t-out-of-$|U_1|$ shares for each of $g_{u,k}$ where $k > 0$: $\{(v, g_{u,k,v})\}_{v \in U_1} \leftarrow \mathbf{SS.share}(g_{u,k}, t, U_1)$ for $k = [1,T]$.
    - For each other user $v \in U_1 \setminus \{u\}$, compute $e_{u,v} \leftarrow \mathbf{AE.enc}(\mathbf{KA.agree}(c_u^{SK}, c_v^{PK}), u \| v \| s_{u,v}^{SK} \| b_{u,v} \| \{g_{u,k,v}\}_{1 \le k \le T})$.
    - If any of the above operations (assertion, signature verification, key agreement, encryption) fails, abort.
    - Send all the ciphertexts $e_{u,v}$ to the server (each implicitly containing addressing information, u, v as metadata).
  - *Server*:
    - Collect lists of ciphertexts from at least t users (denote with $U_2 \subseteq U_1$ this set of users).
    - Sends to each user $u \in U_2$ all ciphertexts encrypted for it: $\{e_{u,v}\}_{v \in U_2}$.
- **Stage 2 (MaskedInputCollection)**:
  - *User* u:
    - Receive (and store) from the server the list of ciphertexts $\{e_{u,v}\}_{v \in U_2}$ (and infer the set U_2). If the list is of size $< t$, abort.
    - For each other user $v \in U_2 \setminus \{u\}$, compute $s_{u,v} \leftarrow \mathbf{KA.agree}(s_u^{SK}, s_v^{PK})$ and expand this value using a **PRG** into a random vector $\boldsymbol{p}_{u,v} = \gamma_{u,v} \cdot \mathbf{PRG}(s_{u,v})$, where $\gamma_{u,v} = 1$ when $u > v$, and $\gamma_{u,v} = -1$ when $u < v$ (note that $\boldsymbol{p}_{u,v} + \boldsymbol{p}_{v,u} = 0 \forall u \neq v$). Define $\boldsymbol{p}_{u,u} = 0$.
    - Compute the user's own private mask vector $\boldsymbol{p}_u = \mathbf{PRG}(b_u)$ and the masked perturbed input $\boldsymbol{y}_u \leftarrow \tilde{\Delta}_u + \boldsymbol{p}_u + \sum_{v \in U_2} \boldsymbol{p}_{u,v} \pmod R$.
    - If any of the above operations (key agreement, PRG) fails, abort. Otherwise, send $\boldsymbol{y}_u$ to the server and move to the next round.
  - *Server*: Collect $\boldsymbol{y}_u$ from at least t users (denote with $U_3 \subseteq U_2$ this set of users). Send to each user in U_3 the list U_3.
- ***[Stage 3 (ConsistencyCheck)]***:
  - *User u: [Receive from the server a list $U_3 \subseteq U_2$ containing at least t users (u included). Abort if $|U_3| < t$. Send to the server $\omega_u' \leftarrow \boldsymbol{SIG.sign}(d_u^{SK}, r \| U_3)$.]*
  - *Server: [Collect ω_u' from at least t users (denote with $U_4 \subseteq U_3$ this set of users). Send to each user in U_4 the set $\{v, \omega_v'\}_{v \in U_4}$.]*
- **Stage 4 (Unmasking)**:
  - *User* u:
    - Receive from the server a list $\{v [, \omega_v']\}_{v \in U_4}$. Verify that $U_4 \subseteq U_3$, that $|U_4| \ge t$ *[, that $\boldsymbol{SIG.ver}(d_v^{PK}, r \| U_3, \omega_v') = 1$ for all $v \in U_4$, (otherwise abort).]*
    - For each other user $v \in U_2 \subseteq \{u\}$, decrypt the ciphertext $v' \| u' \| s_{v,u}^{SK} \| b_{v,u} \| \{g_{v,k,u}\}_{1 \le k \le T} \leftarrow \mathbf{AE.dec}(\mathbf{KA.agree}(c_u^{SK}, c_v^{PK}), e_{v,u})$ received in the **MaskedInputCollection** round and assert that $u = u' \wedge v = v'$.
    - If any of the decryption operations fail (in particular, the ciphertext does not correctly authenticate), abort.
    - Send a list of shares to the server: $s_{v,u}^{SK}$ for users $v \in U_2 \setminus U_3$ and $b_{v,u}$ in $v \in U_3$; and a list of seeds $g_{u,k}$ for $|U \setminus U_3| + 1 \le k \le T$.
  - *Server*:
    - Collect responses from at least t users (denote with $U_5 \subseteq U_4$ this set of users).
    - For each user $u \in U_2 \setminus U_3$, reconstruct $s_u^{SK} \leftarrow \mathbf{SS.recon}(\{s_{u,v}^{SK}\}_{v \in U_5}, t)$ and use it (with the public keys received in the **AdvertiseKeys** round) to recompute $\boldsymbol{p}_{u,v}$ for all $v \in U_3$ using the PRG. For each user $u \in U_3$, reconstruct $b_u \leftarrow \mathbf{SS.recon}(\{b_{u,v}\}_{v \in U_5}, t)$ to recompute $\boldsymbol{p}_u$ using the PRG.
    - Compute $\boldsymbol{z} = \sum_{u \in U_3} \tilde{\Delta}_u$ as $\sum_{u \in U_3} \tilde{\Delta}_u = \sum_{u \in U_3} \boldsymbol{y}_u - \sum_{u \in U_3} \boldsymbol{p}_u + \sum_{u \in U_3, v \in U_2 \setminus U_3} \boldsymbol{p}_{v,u}$.
    - Sends to each user $u \in U_5$ the set U_5, if $U_3 \setminus U_5 \neq \emptyset$.
- **Stage 5 (ExcessiveNoiseRemoval)**:
  - *User* u:
    - Receive from the server the set U_5. Verify that $U_5 \subseteq U_4$ and that $U_5 \ge t$ (otherwise abort).
    - Send a list of shares to the server, which consists of $g_{v,k,u}$ for users $v \in U_3 \setminus U_5$ and $|U \setminus U_3| + 1 \le k \le T$.
  - *Server*:
    - Collect responses from at least t users (denote with $U_6 \subseteq U_5$ this set of users).
    - For each user $u \in U_3 \setminus U_5$, reconstruct $g_{u,k} \leftarrow \mathbf{SS.recon}(\{g_{u,k,v}\}_{v \in U_6}, t)$ for $|U \setminus U_3| + 1 \le k \le T$.
    - Generate random noises $\boldsymbol{n}_{u,k} \overset{g_{u,k}}{\sim} \chi\left(\frac{\sigma_*^2}{(|U|-k+1)(|U|-k)} \cdot \frac{t}{t-T_C}\right)$ for $u \in U_3$ and $|U \setminus U_3| + 1 \le k \le T$ and subtracts them from $\boldsymbol{z}$.

Figure 5. Detailed description of the SecAgg protocol [15] integrated with XNoise (§3.2). *[Italicized parts inside square brackets are required to guarantee security in the malicious threat model.]* Red, underlined parts are specific for XNoise. Green, highlighted parts are secure results of SecAgg reused by XNoise. The symbol $\|$ denotes concatenation.

3.4 Security Analysis

We consider the strongest adversary in our threat model (§2.1), i.e., a malicious server colluding with a subset of sampled clients, as it subsumes weaker adversaries. By being malicious, we mean arbitrary deviation from the protocol, e.g., sending incorrect and/or chosen messages to honest clients, aborting, or omitting messages. In this case, we aim to provide the target level of differential privacy for honest clients (i.e., the adversary never sees an insufficiently perturbed update). We regard the protocol illustrated in Figure 5 (denoted by π) as the target implementation of `XNoise` without loss of generality.

Let η be security parameter; t be the threshold of SecAgg; T and T_C be the tolerated number of clients dropping out and colluding in `XNoise`, respectively; U be the set of sampled clients; $C \subset U \cup \{S\}$ be the set of colluding parties (where S is the server); $\mathbf{\Delta}_{U'} = \{\mathbf{\Delta}_u\}_{u \in U'}$ and $g_{U'} = \{g_{u,k}\}_{u \in U', k \in [0,T]}$ be the input vectors and sampling seeds used in noise addition of any subset of users $U' \subseteq U$, respectively; σ_*^2 be the target level of aggregate noise for each round. Theorem 2 shows that a computationally bounded adversary cannot recover an insufficiently perturbed aggregate during the execution of π. We give the proof in the full version of this paper [37].

Theorem 2 (Privacy against Malicious Adversaries)**.** *For all η, t, U, T, T_C, $C \subseteq U \cup \{S\}$, $\mathbf{\Delta}_{U \setminus C}$ and $g_{U \setminus C}$. If $2t > |U| + |C \cap U|$ and $T_C \geq |C \cap U|$, probabilistic polynomial-time (PPT) adversary, given its view of an execution of π, cannot recover an aggregate update perturbed with noise less than the target level σ_*^2 with non-negligible probability.*

4 Optimal Pipeline Acceleration

As described in §2.3.2, secure aggregation protocols used for distributed DP in FL create a severe performance bottleneck in the round latency. Additionally, integrating our dropout-resilient noise enforcement scheme may further exacerbate its inefficiency. To address this performance issue, we target system-level solutions that preserve all the existing merits of a specific secure aggregation protocol.

Technical Intuition. We first identified three types of operations in distributed DP that use different system resources: 1) `s-comp` that uses the compute resources (e.g., CPU and memory) of the server, 2) `c-comp` that uses the compute resources of clients, and 3) `comm` that relies on server-client communications. As observed in FL practice, plain execution of distributed DP leads to low utilization of these resources over time: `s-comp`, `c-comp`, and `comm` can be idle for up to 53%, 63%, and 93% of the round time, respectively. This indicates that pipelining, which enables overlapping resource usage, is viable for improving the utilization. However, designing and automating pipelined execution for distributed DP in FL leads to two novel system challenges:

Table 1. Abstracting the workflow of dropout-resilient distributed DP into multiple stages for pipelined execution.

Step	Operation	Stage (Resource)
1	Clients encode updates.	1 (`c-comp`)
2	Clients generate security keys.	
3	Clients establish shared secrets.	
4	Clients mask encoded updates.	
5	Clients upload masked updates.	2 (`comm`)
6	Server deals with dropout.	3 (`s-comp`)
7	Server computes aggregate update.	
8	Server updates the global model.	
9	Server dispatches the aggregate.	4 (`comm`)
10	Clients decode the aggregate.	5 (`c-comp`)
11	Clients use the aggregate.	

- Given the large variety in secure aggregation protocols, how to represent their workflows to facilitate generic solutions to the pipelined execution problem (§4.1)?
- Given the complexity of FL environments, how to correctly model them for generating an optimal pipelining plan for maximum acceleration (§4.2)?

4.1 Staging Workflow for Pipelined Execution

To allow for a generic solution, we first abstract away the specifics of secure aggregation protocols.

Abstracting Workflow for Generality. Unlike traditional ML workflows, which involve a computation graph of neural network with general consensus established on their representation [2, 33], secure aggregation protocols combine both computation and communication, with no standardized approach. Nonetheless, we notice that secure aggregation protocols are often designed as multi-round server-client interactions. We thus propose to represent the workflow of a secure aggregation protocol as a sequence of 'round-trip *steps*', each of which starts with the server's request for some data and ends with the related clients' responses. Furthermore, we associate each step with its dominant system resource and group consecutive steps that use the same resource into a *stage* which corresponds to the minimum scheduling unit in pipelining. For example, Table 1 illustrates our representation of a distributed DP protocol using SecAgg [15], where all 11 steps are grouped into 5 stages.

Pipelining via Task Partitioning. By construction, any two adjacent stages consume different system resources, enabling overlapped execution among independent aggregation workflows. Leveraging the coordinate-wise nature of aggregation, Dordis partitions each client's update Δ_i into m chunks $\Delta_{i,1}, \cdots, \Delta_{i,m}$. This divides the original aggregation task into m independent sub-tasks, where the j-th sub-task aggregates the j-th chunks of all clients, i.e.,

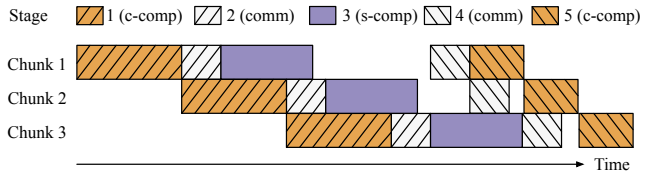


Figure 6. Pipeline scheduling of 3 chunk-aggregation tasks for distributed DP in 5 stages, as specified in Table 1.

$\sum_i \Delta_i = (\sum_i \Delta_{i,1}) \| \cdots \| (\sum_i \Delta_{i,m})$, where $\|$ denotes concatenation. As such, Dordis can enable pipeline parallelism by scheduling the processing stages of the m sub-tasks.

Reducing Design Space. Note that each partition configuration is associated with a completion time that is achieved by executing that configuration. Hence, deriving an optimal pipelined execution plan essentially requires searching for the configuration with the shortest completion time. To avoid a complex combinatorial problem, we focus on evenly partitioning model updates, which reduces the problem to deciding only m, i.e., the number of chunks. Figure 6 illustrates a pipeline execution result with 3 equally-sized aggregation sub-tasks. In Section 6.4, we show that such a reduced consideration suffices to gain a remarkable speedup in practice.

4.2 Determining Optimal Number of Chunks

While a formal description of the optimization problem for pipelined execution is deferred to the full version of this paper [37], it is clear that the key to solving an optimal pipelining plan lies in accurately determining the completion time for each possible choice of m. This requires a performance model and a profiling approach that both fit the FL practice.

Performance Model with Intervention Accounted. To compute the completion time associated with a specific m, Dordis relies on the following performance model that empirically characterizes how the processing time for a sub-task at a stage s, denoted by τ_s, is related to m:

$$\tau_s = \beta_{s,1}\frac{d}{m} + \beta_{s,2}m + \beta_{s,3}, \tag{3}$$

where d is the update size; $\beta_{s,1}$, $\beta_{s,2}$ and $\beta_{s,3}$ are the profiled parameters used to weigh the impact of partition size, inter-task intervention, and constant cost, respectively. The first and third terms are intuitive, while the second term is specifically designed for FL. Unlike traditional ML where each node can be dedicated to one task, resources in FL are scarce, less capable, and do not provide strong isolation across tasks. As the only device contributed by the owner client, a mobile device is seldom fully committed to pipeline stages that use `c-comp`. Instead, some of its CPU cycles will be spent on network IO to facilitate stages that use `comm`. Such distraction can be accentuated as the pipeline goes deeper. Dordis thus respects this intervention effect across pipelined tasks.

Parameter Profiling. β_1, β_2, and β_3 in Equation 3 depend only on the hardware capabilities of the server and the participating clients, especially the slowest one. As the sampled clients' tail latency does not vary vastly across rounds in practice, we let Dordis profile these constants by linear regression with offline micro-benchmarking, which executes the protocol with small-scale proxy data for certain rounds. Note that such lightweight profiling can also be conducted online by interleaving it with the training workflow if needed.

5 Implementation

We have implemented Dordis with 10.3k lines of Python code. It leverages `PyTorch` [61] to instantiate FL applications and employs the distributed DP protocol with DSkellam [5].

System Architecture. Figure 7 shows how Dordis fits in the existing FL workflow, with yellow boxes being Dordis-introduced components. ① *Client sampling and training*: at the beginning of each round, the server randomly samples a subset of available clients as participants. The sampled clients then fetch the global model from the server and compute local updates using private data. ② *Pipeline preparation*: based on the optimal pipeline execution plan provided by the server, each client chunks the local update for pipelined aggregation. ③ *Client processing*: for each update chunk, the client perturbs it with DSkellam's encoding scheme and our `XNoise` noise enforcement approach. The perturbed chunk is further masked following the secure aggregation protocol. ④ *Server aggregation*: the server aggregates and unmasks the received update chunks, removes the excessive parts of their DP noises to ensure that the residual noise remains at the minimum required level, and uses each aggregated update chunk to refine the respective part of the global model.

Programming Interface. Despite our prototype choice, Dordis is proactively designed to be complementary to existing differentially private FL (DPFL) frameworks. In particular, Dordis offers a user-friendly programming interface for developers to implement a variety of privacy and security building blocks. Further details are provided in the full version of this paper [37]. To the best of our knowledge, Dordis is the first generic and end-to-end implementation of distributed DP with pipeline acceleration.

6 Evaluation

We evaluate Dordis's effectiveness on three CV and NLP FL tasks in the semi-honest setting. The highlights of our evaluation are listed below.

1. Our noise enforcement scheme, `XNoise`, ensures that the target privacy level is consistently attained, even when client dropout occurs, without impairing model utility (§6.2). The runtime overhead is deemed acceptable even without pipeline acceleration, and the network overhead remains constant despite the model's expanding size (§6.3).

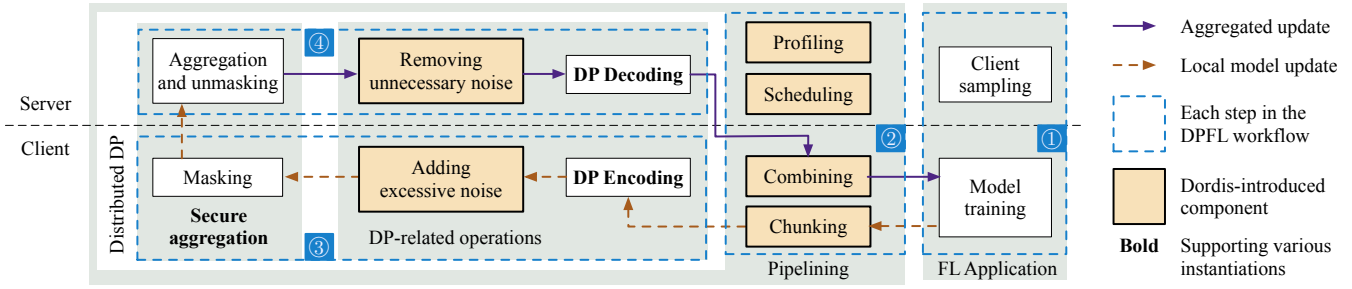


Figure 7. An overview of Dordis' system architecture and how it fits in the existing FL workflow.

2. The pipeline-parallel aggregation design in Dordis significantly enhances the training speed, resulting in up to 2.4× improvement in the round time (§6.4).

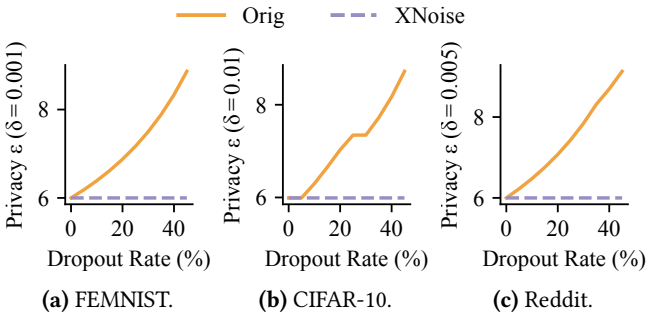
6.1 Methodology

Datasets and Models. We run two categories of applications with three real-world datasets of different scales.

- *Image Classification*: the first dataset, CIFAR-10 [43], consists of 60k colored images categorized into 10 classes. We train a ResNet-18 [29] model with 11M parameters over 100 clients using a non-IID data distribution, generated by applying latent Dirichlet allocation (LDA) [4, 6, 30, 67] with concentration parameters set to 1.0 (i.e., label distributions highly skewed across clients). The second dataset, FEMNIST [16], consists of 805k greyscale images classified into 62 classes. The dataset was partitioned by the original data owners, and we merge every three owners' data to form a client's dataset. We train a CNN model [5, 40] with 1M parameters over 1000 clients.
- *Language Modeling*: the large-scale Reddit dataset [1]. We train an Albert [45] model with 15M parameters over 200 clients for next-word prediction.

Experiment Setup. We launch an AWS EC2 `r5.4xlarge` instance (16 vCPUs and 128 GB memory) for the server and one `c5.xlarge` (4 vCPUs and 8 GB memory) instance for each client, aiming to match the computing power of mobile devices. To emulate hardware heterogeneity, we set the response latencies of clients to follow the Zipf distribution [34–36, 46, 76] with $a = 1.2$ (moderately skewed) such that the end-to-end latency of the i-th slowest client is proportional to i^{-a}. We also emulate network heterogeneity by throttling clients' bandwidth to fall within the range [21Mbps, 210Mbps] to match the typical mobile bandwidth [20] and meanwhile follow another independent Zipf distribution with $a = 1.2$.

Hyperparameters. For FL training, we use the mini-batch SGD for FEMNIST and CNN and AdamW [50] for Reddit, all with momentum set to 0.9. The number of training rounds and local epochs are 50 and 2 for both FEMNIST and Reddit, and 150 and 1 for CIFAR-10, respectively. The batch size and learning rate are 20 and 0.01 for FEMNIST, 128 and 0.005 for CIFAR-10, and 20 and 8e-5 for Reddit, respectively.



(a) FEMNIST. **(b)** CIFAR-10. **(c)** Reddit.

Figure 8. Privacy budget consumption. A larger ϵ corresponds to worse privacy preservation.

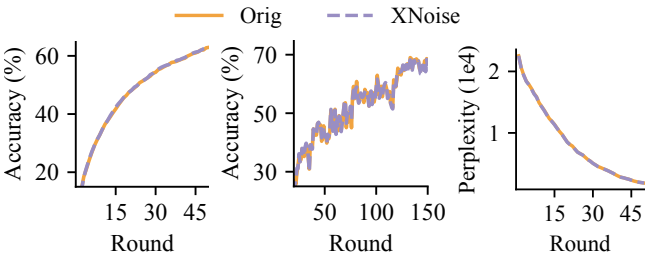
For distributed DP, we set the privacy budget ϵ to 6 and δ the reciprocal of the total number of clients as they represent standard privacy in the DPFL literature [5, 40, 75]. We fix the signal bound multiplier $k = 3$, bias $\beta = e^{-0.5}$, and bit-width $b = 20$ for the configuration of DSkellam [5] as specified in the original paper. The L2-norm clipping bounds [5] for FEMNIST and CIFAR-10 is set to 1 and 3, respectively. Also, there are 100 and 16 clients being sampled in each round for FEMNIST and CIFAR-10, respectively. While we base the implementation of secure aggregation on SecAgg [15] when evaluating our noise enforcement approach, `XNoise`, we also implement SecAgg+ [12] in Section 6.4 to highlight the generality of our distributed pipeline architecture.

Dropout Model. We assume that when clients drop out of the protocol, they drop out after being sampled but before sending their masked and perturbed update to the server. To study the impact of various severities, we let clients randomly drop with a configurable rate in each training round. The dropout rate is consistent within a training process, while varying from 0 to 40% across different processes.

Baseline. We compare Dordis against `Orig`, the original, commonly-used distributed DP protocol that lacks the ability to enforce the target noise level (Definition 1 in §3) and support pipeline execution.

Table 2. Final testing accuracy (for **F**EMNIST and **C**IFAR-10) or perplexity (for **R**eddit, the lower, the better) of **`Orig`** and **`XNoise`** across various dropout rates d.

d	0		10%		20%		30%		40%	
	Ori	XNo	Ori	XNo	Ori	XNo	Ori	XNo	Ori	XNo
F	61.3	61.4	61.4	61.4	61.2	61.4	61.2	61.2	61.4	61.5
C	66.5	66.3	66.7	66.9	66.6	65.7	64.3	65.7	63.8	64.2
R	2169	2142	2158	2179	2286	2285	2294	2317	2299	2329



(a) FEMNIST. **(b)** CIFAR-10. **(c)** Reddit.

Figure 9. Round-to-accuracy performance (20% dropout).

6.2 Effectiveness of Noise Enforcement

`XNoise` Ensures Privacy without Sacrificing Utility. Figure 8 displays the end-to-end privacy budget consumption. As expected, `XNoise` achieves the target privacy ($\epsilon = 6$) in all cases by accurately enforcing the target noise (Theorem 1). On the other hand, due to the missing noise contributions from dropped clients, the overall privacy budget consumed by `Orig` dramatically grows as the severity of client dropout increases. For example, when the dropout rate is 40%, training FEMNIST, CIFAR-10, and Reddit to the preset number of rounds ends up consuming an ϵ of 8.3, 8.2, and 8.7, respectively.

We also report the final model accuracy in Table 2. Compared to `Orig`, which fails to achieve the target privacy in the presence of client dropout, our `XNoise` converges at the same speed (as exemplified by the learning curves in Figure 9 where the per-round dropout rate is 20%) and induces no more than 0.9% accuracy loss as it uses the minimum noise required to maintain the desired level of privacy. It should be noted that the accuracies obtained with Dordis are similar to those in other FL studies that use distributed DP [5, 40, 75]. This is because FL clients have limited and non-IID data, and model updates are discretized for secure aggregation. Additionally, in some cases, `XNoise` achieves higher accuracy than `Orig`, even though the latter uses less noise. This is because the stochasticity introduced by slight additional random noise can act as a regularizer to reduce overfitting [48, 58].

6.3 Efficiency of Noise Enforcement

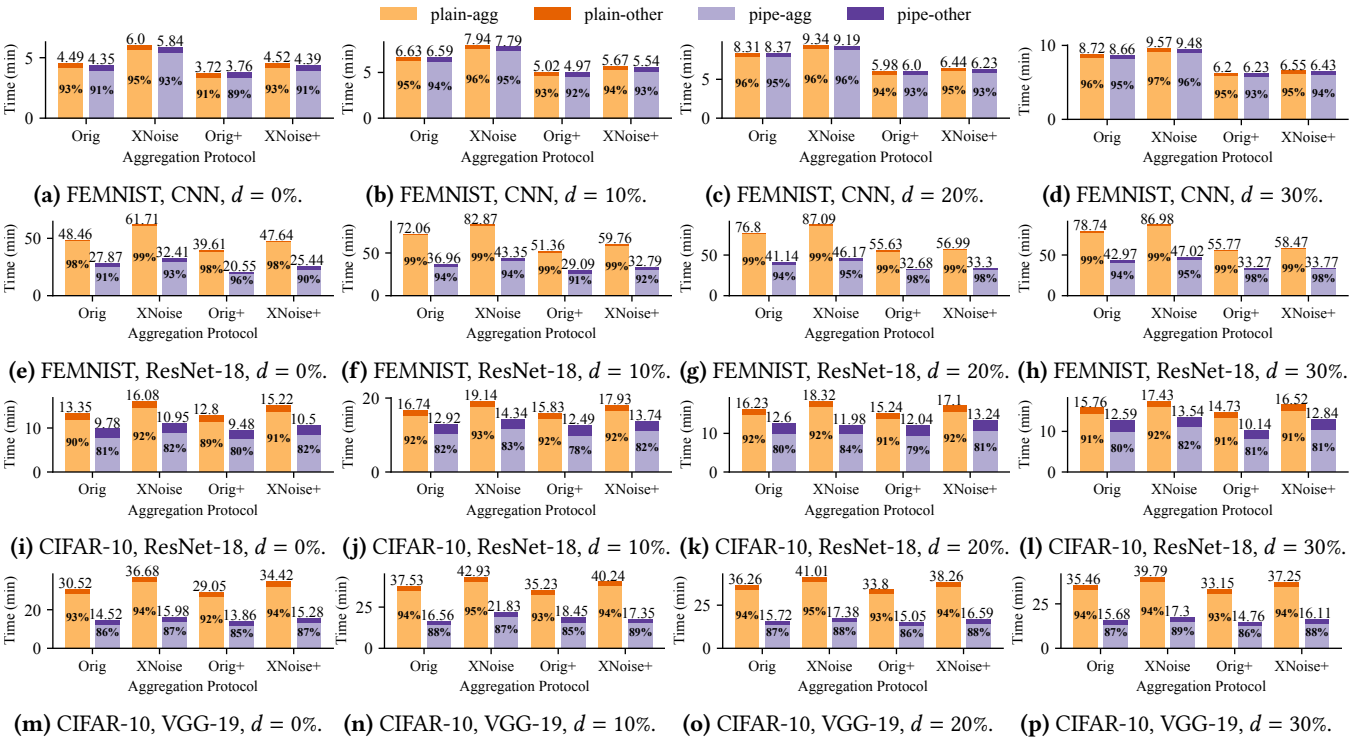
Table 3. Additional per-round network footprint in MB for a surviving client in '**r**ebasing' and **`XNoise`**, compared to `Orig`, across various dropout rate d.

Model size (# parameters)		5M		50M		500M	
d	# sampled clients	r	X	r	X	r	X
0%	100	11.9	0.6	119.2	0.6	1192.1	0.6
	200		2.4		2.4		2.4
	300		5.5		5.5		5.5
10%	100		0.6		0.6		0.6
	200		2.4		2.4		2.4
	300		5.3		5.3		5.3
20%	100		0.6		0.6		0.6
	200		2.3		2.3		2.3
	300		5.2		5.2		5.2
30%	100		0.6		0.6		0.6
	200		2.3		2.3		2.3
	300		5.2		5.2		5.2

`XNoise` Induces Acceptable Overhead in Time. Figure 10 shows the average round time, broken down into two components: 'agg', related to distributed DP operations, and 'other', related to remaining workflow operations such as model training. In a plain execution without pipeline acceleration (yellow bars for `Orig` and `XNoise` in Figure 10a to 10d and Figure 10i to 10l), `XNoise` extends the round time by up to 34% given no dropout, and by up to 19%, 13%, and 12% when the per-round dropout rate is 10%, 20%, and 30%, respectively. This implies a negative relationship between the time cost and dropout severity, as the more clients that drop out, the less noise the server needs to remove in `XNoise` (Definition 2). Such a relationship also implies an acceptable overhead in practice, as Dordis targets scenarios with client dropout. Moreover, the overhead can be further reduced by Dordis's pipeline acceleration (§6.4).

`XNoise`'s Network Overhead is Invariant of Model Size. The noise decomposition in `XNoise` allows for the transmission of noise seeds (§3.1). To examine its practical advantage over 'rebasing', we benchmark their additional network footprint induced to a surviving client compared to `Orig`. As specified in real deployments, the size of a model weight, noise seed, Shamir share of seed, ciphertext of a share (the latter three are configured for `XNoise` only) are set to 2.5, 32, 16, and 120 in bytes, respectively. As seen in Table 3, as the model size increases, the network overhead of `XNoise` remains constant and low, while that of 'rebasing' grows linearly. The communication time of 'rebasing' can thus be prohibitive when the model is large in size (e.g., >500M) and/or the client's bandwidth is low (e.g., <1Mbps). Additionally, a large overhead makes client failures in the middle of noise removal more likely, which 'rebasing' cannot handle without the loss of model utility (§3.1).

It is important to highlight that the cost of Dordis is dependent on the number of sampled clients rather than the



(a) FEMNIST, CNN, $d = 0\%$. **(b)** FEMNIST, CNN, $d = 10\%$. **(c)** FEMNIST, CNN, $d = 20\%$. **(d)** FEMNIST, CNN, $d = 30\%$.

(e) FEMNIST, ResNet-18, $d = 0\%$. **(f)** FEMNIST, ResNet-18, $d = 10\%$. **(g)** FEMNIST, ResNet-18, $d = 20\%$. **(h)** FEMNIST, ResNet-18, $d = 30\%$.

(i) CIFAR-10, ResNet-18, $d = 0\%$. **(j)** CIFAR-10, ResNet-18, $d = 10\%$. **(k)** CIFAR-10, ResNet-18, $d = 20\%$. **(l)** CIFAR-10, ResNet-18, $d = 30\%$.

(m) CIFAR-10, VGG-19, $d = 0\%$. **(n)** CIFAR-10, VGG-19, $d = 10\%$. **(o)** CIFAR-10, VGG-19, $d = 20\%$. **(p)** CIFAR-10, VGG-19, $d = 30\%$.

Figure 10. The round time breakdown for **plain** execution and **pipe**line acceleration. The implemented secure aggregation is SecAgg (w/o +) and SecAgg+ (w/ +). The number of sampled clients for FEMNIST and CIFAR-10 are 100 and 16, respectively. The model size for CNN, ResNet-18, and VGG-19 is 1M, 11M, and 20M, respectively. d is the per-round dropout rate.

overall population size. In FL systems, although the population can be extensive, the number of sampled clients is typically restricted to a few hundred [14, 42]. This limitation is due to the fact that involving additional clients beyond a certain threshold only yields marginal benefits in terms of accelerating convergence [53, 80]. Therefore, the results presented in Table 3 are evaluated from practical settings of sample sizes and are applicable in real-world scenarios.

6.4 Efficiency of Pipeline Acceleration

To study the impact of model sizes, we additionally train a ResNet-18 over the FEMNIST dataset, and a VGG-19 model [71] (20M parameters) over the CIFAR-10 dataset.

Dordis Generally Benefits from Pipelining. Figure 10 shows that Dordis's pipeline acceleration benefits all the evaluated aggregation protocols by providing a generic pipeline architecture (§5). Specifically, utilizing idle resources with pipelined execution can speed up `Orig` by up to 2.3× (resp. 2.2×) when the implemented secure aggregation is SecAgg (resp. SecAgg+), while `XNoise` can achieve a comparable maximum speedup of 2.4× (resp. 2.3×) with pipelining. We also notice that the speedup is similar across different evaluated dropout rates when fixing the use of a protocol.

Dordis Gains More Speedup with Larger Models. The results in Figure 10i to 10p show that CIFAR-10 with ResNet-18 (11M) is accelerated by 1.3-1.5×, while CIFAR-10 with VGG-19 (20M) is accelerated by 1.9-2.5× by pipelining, indicating that larger models benefit more from this approach. Similar observations can be made when comparing the results of FEMNIST with CNN (1M) and with ResNet-18 (11M). This trend can be explained by Amdahl's law [7]: as the aggregation time of larger models has a higher dominant factor p in the round latency (e.g., p = 89-93% in CIFAR-10 with ResNet-18 and p = 93-95% in CIFAR-10 with VGG-19), assuming the same speedup s of the aggregation offered by pipelining, one can expect a higher overall speedup S for larger models as $S = 1/((1-p) + p/s)$.

Dordis Scales with Number of Sampled Clients. As reported by Figure 10e to 10l, CIFAR-10 with ResNet-18 (16 sampled clients) and FEMNIST with ResNet-18 (100 sampled clients) both benefit from pipelined execution, with FEMNIST achieving a larger speedup (1.7-2.0×) than CIFAR-10 (1.3-1.5×). It is important to note that the dominant part of the round time is the aggregation process (`plain-agg`), which is independent of the learning task. Therefore, the superior performance of Dordis on FEMNIST with ResNet-18 over CIFAR-10 with ResNet-18 can be mainly attributed to the larger number of sampled clients, indicating that Dordis

scales well and may even perform better with large-scale training. This further suggests that the reason why FEMNIST with CNN gains negligible speedup (Figure 10a to 10d) is due to its small model size, not large participation scale.

7 Discussion

Random Client Sampling with VRFs. As mentioned in §2.1, we can enforce mild collusion among sampled clients even in the presence of a malicious server through the use of VRFs. Our key insight is that incorporating verifiable randomness can prevent the server from manipulating the sampling process to include a disproportionate number of dishonest clients. Given that dishonest clients only represent a small portion of the entire population (e.g., billions of Apple devices [8]), a VRF-based client selection can ensure that dishonest clients remain a minority in the sampled participants, thus effectively preventing Sybil attacks.

The expected low base rate of dishonest clients results from the prohibitive costs associated with creating and managing a large number of simulated clients. The adversary is hindered by the high expenses of registering numerous identities to a public key infrastructure (PKI) operated by a qualified trust service provider [12, 15]. Furthermore, maintaining a client botnet at scale incurs substantial monetary expenditure [86].

Regarding the use of verifiable randomness, we briefly describe a possible design as introduced in a recent work [38]. Initially, the server initiates a training round by making an announcement. Each client within the population employs a VRF using its private key to generate a random number along with an associated proof, with the current round index serving as input. By comparing the generated random number with a predetermined threshold, a client determines whether it should participate in the ongoing round. Once a client decides to join, it informs the server about its random number and provides the corresponding proof. Upon receiving responses from all participating clients, the server considers them as participants and broadcasts their responses for mutual verification. Ultimately, a participant proceeds with the training only if all verification tests are successfully passed.

In the above design, a client does not need to know every other clients to reproduce the sampling process for the entire population, as the server does. Instead, a sampled client only needs to refer to the identities of other sampled clients and verify their randomness. Moreover, the server can achieve a fixed sample size by first slightly adjusting the selection threshold for over-selection, and then discarding excessive clients based on indiscriminate criteria on their randomness.

8 Related Work

The topic of differentially private FL (DPFL) has seen a surge of interest in recent years. DP-FedAvg [54, 65], Distributed DP-SGD [11], and DP-FTRL [41] initiate DPFL in the central DP model where the server is trusted. Unlike these works, Dordis studies semi-honest and malicious adversaries.

In distributed DP, recent efforts have made significant progress in integrating secure aggregation with DP mechanisms. DDGauss [40] and DSkellam [5] provide end-to-end privacy analysis by combining the DP noise addition with SecAgg. FLDP [75] explores aggregation based on the learning with errors (LWE) problem [68], using residual errors as DP noise. Dordis complements these works by providing dropout resilience and improved execution efficiency.

The work closest to Dordis is [10] which tackles the dropout resilience problem in distributed DP. Unlike Dordis, they adopt a 'rebasing' design that is incompatible with transmitting seeds (thus, poor efficiency) and does not handle client dropout during noise removal (thus, poor robustness) (§3.1). Also, it is unclear how to implement their noise enforcement securely against malicious adversaries.

Replacing secure aggregation with other cryptographic primitives in distributed DP is far less studied in FL mainly due to their inefficiency. For instance, [77] explores the use of homomorphic encryption schemes, but induces an overhead of over 16 minutes to encrypt a model of size 1M.

Finally, several studies have focused on enhancing the SecAgg protocol, both in general-purpose scenarios [12, 39, 72, 73] and in those specific to FL [49, 52, 66, 84], without integration with DP. Dordis is thus independent of these studies in terms of noise enforcement. Dordis also offers the first generic pipeline solution that can expedite them.

9 Conclusion

This paper presents Dordis, an efficient FL framework that enables efficient and dropout-resilient distributed DP in realistic scenarios. To handle client dropout, Dordis designs an efficient and secure "add-then-remove" approach to enforce the required noise at the target level precisely. Dordis also enables pipeline parallelism for accelerated secure aggregation with a generic distributed parallel architecture. Compared to existing distributed DP mechanisms in FL, Dordis enforces privacy guarantees with optimal model utility in the presence of client dropout without requiring manual strategies. It also achieves up to 2.4× faster training completion.

Acknowledgments

We extend our special thanks to Peter Kairouz, Zheng Xu, Marco Canini, and Suhaib A. Fahmy for their valuable discussions and suggestions that helped improve the early shape of this work. We are also grateful to our shepherd, Manuel Costa, and the anonymous EuroSys reviewers for their constructive feedback, which greatly enhanced the quality of this paper. We would like to acknowledge Shaohuai Shi for providing GPU clusters and environment settings. This work was supported in part by RGC RIF grant R6021-20 and RGC GRF grant 16211123.

References

[1] [n. d.]. Reddit Comment Data. https://files.pushshift.io/reddit/comments.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*.

[3] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *CCS*.

[4] Durmus Alp Emre Acar, Yue Zhao, Ramon Matas Navarro, Matthew Mattina, Paul N Whatmough, and Venkatesh Saligrama. 2021. Federated learning based on dynamic regularization. In *ICLR*.

[5] Naman Agarwal, Peter Kairouz, and Ziyu Liu. 2021. The Skellam mechanism for differentially private federated learning. In *NeurIPS*.

[6] Maruan Al-Shedivat, Jennifer Gillenwater, Eric Xing, and Afshin Rostamizadeh. 2020. Federated Learning via Posterior Averaging: A New Perspective and Practical Algorithms. In *ICLR*.

[7] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

[8] Apple. 2023. Apple Reports First Quarter Results. https://www.apple.com/newsroom/2023/02/apple-reports-first-quarter-results.

[9] Arm. 2021. TrustZone technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[10] Chunghun Baek, Sungwook Kim, Dongkyun Nam, and Jihoon Park. 2021. Enhancing differential privacy for federated learning at scale. *IEEE Access* (2021).

[11] Borja Balle, Peter Kairouz, Brendan McMahan, Om Thakkar, and Abhradeep Guha Thakurta. 2020. Privacy amplification via random check-ins. In *NeurIPS*.

[12] James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. 2020. Secure single-server aggregation with (poly) logarithmic overhead. In *CCS*.

[13] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. 2017. Prochlo: Strong privacy for analytics in the crowd. In *SOSP*.

[14] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konečnỳ, Stefano Mazzocchi, Brendan McMahan, et al. 2019. Towards federated learning at scale: System design. In *MLSys*.

[15] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical secure aggregation for privacy-preserving machine learning. In *CCS*.

[16] Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. 2019. Leaf: A benchmark for federated settings. In *NeurIPS Workshop*.

[17] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. 2019. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *USENIX Security*.

[18] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *USENIX Security*.

[19] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. 2019. Distributed differential privacy via shuffling. In *Eurocrypt*.

[20] Cisco. 2020. Cisco Annual Internet Report (2018–2023) White Paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[21] Dwork Cynthia. 2006. Differential privacy. *Automata, languages and programming* (2006), 1–12.

[22] Yevgeniy Dodis and Aleksandr Yampolskiy. 2005. A verifiable random function with short proofs and keys. In *PKC*.

[23] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

[24] Cynthia Dwork and Aaron Roth. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[25] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. 2019. Amplification by shuffling: From local to central differential privacy via anonymity. In *SODA*.

[26] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. 2018. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security* (2018).

[27] Jonas Geiping, Hartmut Bauermeister, Hannah Dröge, and Michael Moeller. 2020. Inverting gradients-how easy is it to break privacy in federated learning?. In *NeurIPS*.

[28] Google. 2021. Your chats stay private while messages improves suggestions. https://support.google.com/messages/answer/9327902.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*.

[30] Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. 2019. Measuring the effects of non-identical data distribution for federated visual classification. *arXiv:1909.06335* (2019).

[31] Intel. 2021. Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[32] Tayyebeh Jahani-Nezhad, Mohammad Ali Maddah-Ali, Songze Li, and Giuseppe Caire. 2023. SwiftAgg+: Achieving asymptotically optimal communication loads in secure aggregation for federated learning. *IEEE Journal on Selected Areas in Communications* (2023).

[33] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*.

[34] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *SOSP*.

[35] Zhifeng Jiang, Wei Wang, Baochun Li, and Bo Li. 2022. Pisces: Efficient Federated Learning via Guided Asynchronous Training. In *SoCC*.

[36] Zhifeng Jiang, Wei Wang, Bo Li, and Qiang Yang. 2022. Towards Efficient Synchronous Federated Training: A Survey on System Optimization Strategies. *IEEE Transactions on Big Data* (2022).

[37] Zhifeng Jiang, Wei, and Ruichuan Chen. 2023. Dordis: Efficient Federated Learning with Dropout-Resilient Differential Privacy. In *arXiv:2209.12528*.

[38] Zhifeng Jiang, Peng Ye, Shiqi He, Wei Wang, Ruichuan Chen, and Bo Li. [n. d.]. Lotto: Secure Participant Selection for Federated Learning with Malicious Server. https://github.com/SamuelGong/Lotto-doc.

[39] Swanand Kadhe, Nived Rajaraman, O Ozan Koyluoglu, and Kannan Ramchandran. 2020. Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning. *arXiv:2009.11248* (2020).

[40] Peter Kairouz, Ziyu Liu, and Thomas Steinke. 2021. The distributed discrete gaussian mechanism for federated learning with secure aggregation. In *ICML*.

[41] Peter Kairouz, Brendan McMahan, Shuang Song, Om Thakkar, Abhradeep Thakurta, and Zheng Xu. 2021. Practical and private (deep) learning without sampling or shuffling. In *ICML*.

[42] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. 2021. Advances and open problems in federated learning. *Foundations and Trends in Machine Learning* 14, 1 (2021).

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

[44] Fan Lai, Yinwei Dai, Sanjay S Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. 2022. FedScale:

Benchmarking model and system performance of federated learning at scale. In *ICML*.

[45] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. Albert: A lite bert for self-supervised learning of language representations. In *ICLR*.

[46] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the black box of machine learning prediction serving systems. In *OSDI*.

[47] Wenqi Li, Fausto Milletarì, Daguang Xu, Nicola Rieke, Jonny Hancox, Wentao Zhu, Maximilian Baust, Yan Cheng, Sébastien Ourselin, M Jorge Cardoso, et al. 2019. Privacy-preserving federated brain tumour segmentation. In *MLMI Workshop*.

[48] Ruixuan Liu, Yang Cao, Masatoshi Yoshikawa, and Hong Chen. 2020. Fedsel: Federated sgd under local differential privacy with top-k dimension selection. In *DASFAA*.

[49] Zizhen Liu, Si Chen, Jing Ye, Junfeng Fan, Huawei Li, and Xiaowei Li. 2023. DHSA: efficient doubly homomorphic secure aggregation for cross-silo federated learning. *The Journal of Supercomputing* (2023).

[50] Ilya Loshchilov and Frank Hutter. 2018. Decoupled Weight Decay Regularization. In *ICLR*.

[51] Heiko Ludwig, Nathalie Baracaldo, Gegi Thomas, Yi Zhou, Ali Anwar, Shashank Rajamoni, Yuya Ong, Jayaram Radhakrishnan, Ashish Verma, Mathieu Sinn, et al. 2020. Ibm federated learning: an enterprise framework white paper v0.1. *arXiv:2007.10987* (2020).

[52] Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. 2023. Flamingo: Multi-Round Single-Server Secure Aggregation with Applications to Private Federated Learning. In *IEEE S&P*.

[53] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*.

[54] H Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. 2018. Learning Differentially Private Recurrent Language Models. In *ICLR*.

[55] Ralph C Merkle. 1978. Secure communications over insecure channels. *Commun. ACM* 21, 4 (1978), 294–299.

[56] Silvio Micali, Michael Rabin, and Salil Vadhan. 1999. Verifiable random functions. In *FOCS*.

[57] Milad Nasr, Reza Shokri, and Amir Houmansadr. 2019. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *IEEE S&P*.

[58] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. 2015. Adding gradient noise improves learning for very deep networks. *arXiv:1511.06807* (2015).

[59] NVIDIA. 2020. Triaging COVID-19 Patients: 20 Hospitals in 20 Days Build AI Model that Predicts Oxygen Needs. https://blogs.nvidia.com/blog/2020/10/05/federated-learning-covid-oxygen-needs/.

[60] Stuart L Pardau. 2018. The California consumer privacy act: Towards a European-style privacy regime in the United States. *J. Tech. L. & Pol'y* 23 (2018), 68.

[61] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*.

[62] Matthias Paulik, Matt Seigel, Henry Mason, Dominic Telaar, Joris Kluivers, Rogier van Dalen, Chi Wai Lau, Luke Carlson, Filip Granqvist, Chris Vandevelde, et al. 2021. Federated Evaluation and Tuning for On-Device Personalization: System Design & Applications. *arXiv:2102.08503* (2021).

[63] Vasyl Pihur, Aleksandra Korolova, Frederick Liu, Subhash Sankuratripati, Moti Yung, Dachuan Huang, and Ruogu Zeng. 2018. Differentially-private" draw and discard" machine learning. *arXiv:1807.04369* (2018).

[64] Tina Piper. 2000. The Personal Information Protection and Electronic Documents Act-A Lost Opportunity to Democratize Canada's Technological Society. *Dalhousie LJ* 23 (2000), 253.

[65] Swaroop Ramaswamy, Om Thakkar, Rajiv Mathews, Galen Andrew, H Brendan McMahan, and Françoise Beaufays. 2020. Training production language models without memorizing user data. *arXiv:2009.10031*.

[66] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. 2023. ELSA: Secure Aggregation for Federated Learning with Malicious Actors. In *S&P*.

[67] Sashank J Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and Hugh Brendan McMahan. 2020. Adaptive Federated Optimization. In *ICLR*.

[68] Oded Regev. 2009. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM* 56, 6 (2009), 1–40.

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

[70] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *IEEE S&P*.

[71] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556* (2014).

[72] Jinhyun So, Başak Güler, and A Salman Avestimehr. 2021. Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning. *IEEE Journal on Selected Areas in Information Theory* 2, 1 (2021), 479–489.

[73] Jinhyun So, Corey J Nolet, Chien-Sheng Yang, Songze Li, Qian Yu, Ramy E Ali, Basak Guler, and Salman Avestimehr. 2022. Lightsecagg: a lightweight and versatile design for secure aggregation in federated learning. In *MLSys*.

[74] Congzheng Song and Vitaly Shmatikov. 2019. Auditing data provenance in text-generation models. In *SIGKDD*.

[75] Timothy Stevens, Christian Skalka, Christelle Vincent, John Ring, Samuel Clark, and Joseph Near. 2022. Efficient Differentially Private Secure Aggregation for Federated Learning via Hardness of Learning with Errors. In *USENIX Security*.

[76] Huangshi Tian, Minchen Yu, and Wei Wang. 2021. CrystalPerf: Learning to Characterize the Performance of Dataflow Computation through Code Analysis. In *ATC*.

[77] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. 2019. A hybrid approach to privacy-preserving federated learning. In *AISec*.

[78] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.

[79] Paul Voigt and Axel Von dem Bussche. 2017. The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing* 10, 3152676 (2017), 10–5555.

[80] Jianyu Wang, Zachary Charles, Zheng Xu, Gauri Joshi, H Brendan McMahan, Maruan Al-Shedivat, Galen Andrew, Salman Avestimehr, Katharine Daly, Deepesh Data, et al. 2021. A field guide to federated optimization. In *arXiv:2107.06917*.

[81] Junxiao Wang, Song Guo, Xin Xie, and Heng Qi. 2022. Protect Privacy from Gradient Leakage Attack in Federated Learning. In *INFOCOM*.

[82] WeBank. 2020. Utilization of FATE in Anti Money Laundering Through Multiple Banks. https://www.fedai.org/cases/utilization-of-fate-in-anti-money-laundering-through-multiple-banks/.

[83] Chengxu Yang, Qipeng Wang, Mengwei Xu, Zhenpeng Chen, Kaigui Bian, Yunxin Liu, and Xuanzhe Liu. 2021. Characterizing impacts of heterogeneity in federated learning upon large-scale smartphone data. In *WWW*.

[84] Shisong Yang, Yuwen Chen, Zhen Yang, BoWen Li, and Huan Liu. 2023. Fast Secure Aggregation with High Dropout Resilience for Federated Learning. In *TGCN*.
[85] Hongxu Yin, Arun Mallya, Arash Vahdat, Jose M Alvarez, Jan Kautz, and Pavlo Molchanov. 2021. See through gradients: Image batch recovery via gradinversion. In *CVPR*.
[86] Joshua C Zhao, Atul Sharma, Ahmed Roushdy Elkordy, Yahya H Ezzeldin, Salman Avestimehr, and Saurabh Bagchi. 2023. Secure aggregation in federated learning is not private: Leaking user data at large scale through model modification. In *arXiv*.

A Artifact Appendix

A.1 Abstract

We have made the artifact available in our GitHub repository's main branch[4] for reproducing the experimental results presented in the paper. Additionally, for long-term accessibility, we have also uploaded the artifact to a Zenodo repository.[5] The artifact comprises the source code of the Dordis, along with configuration files and Python scripts necessary to execute the experiments described in the paper. Further instructions on using the artifact can be found in the subsequent sections and in the README file of the repository.

A.2 Description & Requirements

A.2.1 How to access. The artifact is available at the abovementioned repositories.

A.2.2 Hardware dependencies. Paper experiments were done in the following two modes:

- *Simulation*: When assessing the efficacy of noise enforcement (§6.2), each experiment can be effectively simulated using a single node equipped with multiple GPUs for acceleration. This is because the metrics of interest, namely privacy budget consumption and final model accuracy, remain unaffected by system speed or network bandwidth. As a reference, our evaluation employed a machine with 8 NVIDIA Geforce RTX 2080 Ti GPUs.
- *Cluster Deployment*: During the evaluation of noise enforcement efficiency (§6.3) and pipeline acceleration (§6.4), the respective experiments necessitate an EC2 cluster setup. In this configuration, an `r5.4xlarge` instance serves as the server, while each client node is equipped with a `c5.xlarge` instance. This setup aims to replicate the computational capabilities of mobile devices. Additionally, the artifact includes scripts for simulating network heterogeneity by limiting the bandwidth of the client instances.

A.2.3 Software dependencies. The artifact requires an Anaconda environment with Python 3. While the specific Python packages used are listed in the 'requirement.txt' file of the GitHub repository, one can easily install them using provided scripts.

[4]At https://github.com/SamuelGong/Dordis.
[5]At https://doi.org/10.5281/zenodo.10023704.

A.2.4 Benchmarks. As mentioned in §6.1, the provided artifact utilizes publicly available machine learning datasets, namely CIFAR-10 [43], FEMNIST [16], and Reddit [1]. Additionally, a variety of models are employed, including ResNet-18 [29], a customized CNN [5, 40], VGG-19 [71], and Albert [45]. The artifact includes automated scripts that can be used to download and preprocess all the datasets.

A.3 Set-up

Installation and configuration steps required to prepare the artifact environment are described in the 'Simulation' and 'Cluster Deployment' section of the repository README file.

A.4 Evaluation workflow

A.4.1 Major Claims. As mentioned in §6, the major experimental claims made in the paper are:

- C1: Our noise enforcement scheme, `XNoise`, guarantees the consistent achievement of the desired privacy level, even in the presence of client dropout, while maintaining the model utility. This claim is supported by the simulation experiment (E1) outlined in §6.2, with the corresponding results presented in Figure 8, Table 2, and Figure 9.
- C2: The `XNoise` scheme introduces acceptable runtime overhead, with the network overhead being scalable with respect to the model's expanding size. S This can be proven by the cluster experiment (E2) described in 6.3 whose results are reported in Figure 10 and Table 3.
- C3: The pipeline-parallel aggregation design employed by Dordis significantly boosts training speed, leading to a remarkable improvement of up to 2.4× in the training round time. This finding is supported by the cluster experiment (E3) discussed in §6.4, and the corresponding results are depicted in Figure 10.

A.4.2 Experiments. The artifact includes a dedicated section in the repository's README file titled 'Reproducing Experimental Results'. This section provides a comprehensive guide on conducting the necessary experiments to replicate the main claims mentioned earlier. For every experiment, we meticulously document all the essential setup requirements, estimated costs and timeframes, a concise explanation of the procedure, the anticipated outcomes, and an explicit link to one of the aforementioned three claims.

Received 24 May 2023; accepted 12 September 2023