Buffalo: A Practical Secure Aggregation Protocol for Buffered Asynchronous Federated Learning

Riccardo Taiello*
CERN
Geneva, Switzerland
riccardo.taiello@cern.ch

Clémentine Gritti
INSA Lyon
Villeurbanne, France
clementine.gritti@insa-lyon.fr

Melek Önen
EURECOM
Sophia Antipolis, France
melek.onen@eurecom.fr

Marco Lorenzi
Inria / Université Côte d'Azur
Sophia Antipolis, France
marco.lorenzi@inria.fr

ABSTRACT

Federated Learning (FL) has become a crucial framework for collaboratively training Machine Learning (ML) models while ensuring data privacy. Traditional synchronous FL approaches, however, suffer from delays caused by slower clients (called *stragglers*), which hinder the overall training process. Specifically, in a synchronous setting, model aggregation happens once all the intended clients have submitted their local updates to the server. To address these inefficiencies, Buffered Asynchronous FL (BAsyncFL) was introduced, allowing clients to update the global model as soon as they complete local training. In such a setting, the new global model is obtained once the buffer is full, thus removing synchronization bottlenecks. Despite these advantages, existing Secure Aggregation (SA) techniques—designed to protect client updates from inference attacks—rely on synchronized rounds, making them unsuitable for asynchronous settings.

In this paper, we present **Buffalo**, the first practical SA protocol tailored for BAsyncFL. **Buffalo** leverages lattice-based encryption to handle scalability challenges in large ML models and introduces a new role, the *assistant*, to support the server in securely aggregating client updates. To protect against an actively corrupted server, we enable clients to verify that their local updates have been correctly integrated into the global model. Our comprehensive evaluation—incorporating theoretical analysis and real-world experiments on benchmark datasets—demonstrates that **Buffalo** is an efficient and scalable privacy-preserving solution in BAsyncFL environments.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**;

KEYWORDS

Secure Aggregation, Asynchronous Federated Learning

*Work done at Inria / EURECOM / Université Côte d'Azur.

CODASPY '25, June 4–6, 2025, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-1476-4/2025/06
https://doi.org/10.1145/3714393.3726498

ACM Reference Format:
Riccardo Taiello, Clémentine Gritti, Melek Önen, and Marco Lorenzi. 2025. Buffalo: A Practical Secure Aggregation Protocol for Buffered Asynchronous Federated Learning. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy (CODASPY '25), June 4–6, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3714393.3726498

1 INTRODUCTION

Federated Learning (FL) [32] has rapidly emerged as a dominant paradigm for collaboratively training Machine Learning (ML) models while maintaining the privacy of individual data. In the traditional Synchronous FL (SyncFL) framework, a central server initializes the global model and sends its parameters to a pre-selected subset of clients. These selected clients optimize the model parameters using their local data and transmit their updates to the server for aggregation, which typically consists of weighted averaging. The same process is repeated through several, well defined, rounds until the model reaches a certain level of accuracy. Sharing local model parameters introduces vulnerabilities that can inadvertently expose sensitive client data to risks such as membership inference or model inversion attacks [33, 41]. To mitigate these risks, Secure Aggregation (SA) techniques have been proposed [4, 5, 29–31], which protect client updates prior to transmission.

Despite its advantages, FL faces challenges due to the hardware heterogeneity of the involved clients, differing significantly in storage, communication, and computation capabilities. Notably, the presence of *stragglers*, i.e. slower participating clients, can severely delay the training rounds, especially in traditional FL frameworks which assume that FL clients are strongly synchronized with the server. Such delays can adversely affect the overall performance of the FL system [9, 35].

One strategy to manage *stragglers* in a SyncFL environment is the over-selection of clients [9]. This approach involves selecting a larger subset of clients than necessary in anticipation that some will not complete their tasks promptly. However, this method can lead to inefficiencies, as slow clients that have locally trained the model might not be included in the updated global model, leading to wasted computational resources and potential loss of diverse data contributions. Additionally, it significantly affects SA processes since non-selected clients are treated as dropped, increasing protocol complexities.

Asynchronous FL (AsyncFL) frameworks, as described in [47], address the aforementioned inefficiencies by processing client updates as they arrive, thus eliminating the need for synchronized rounds. However, in pure AsyncFL methods, each submitted local contribution results in an immediate server model update. This approach poses challenges for privacy, as traditional SA techniques become inadequate. Specifically, SA relies on aggregating multiple updates to protect individual contributions, which is not possible in a pure AsyncFL setting where updates are processed individually.

To address these issues, a compatible privacy-preserving AsyncFL framework has been proposed, namely Buffered AsyncFL (BAsyncFL) [35], where a server collects local inputs in a buffer and updates the global model periodically, i.e., every time the buffer is full. Nonetheless, most of the SA solutions are synchronous, necessitating clients' prior knowledge of their participation and complicating their direct application to BasyncFL.

Another challenge encountered in FL settings is the absence of robust aggregation guarantees. This issue is intrinsic to both SyncFL and AsyncFL environments. In particular, a malicious server could deliberately aggregate distinct sets of local inputs and send their corresponding outputs to different clients. As a result, these clients end up training on inconsistent global models, which undermines the overall accuracy expectations [37].

Our contributions in this paper* include the following:

• We introduce **Buffalo**, the first practical SA protocol designed specifically for BAsyncFL. **Buffalo** employs lattice-based encryption on models, coupled with homomorphic encryption on lattice-based keys, to address scalability issues related to the dimensions of ML models. We also define a new role, the *assistant*, to assist the server during the aggregation phase. Informally, clients generate fresh, ephemeral keys for each local update, and *assistants* help the server reconstruct these keys. This ensures efficient SA without requiring synchronized rounds.

• We ensure aggregation integrity to address potential server threats. For instance, a malicious server might distribute different global models—derived from aggregating distinct sets of local inputs—to various clients. This threat, referred to as model inconsistency in [37], poses a significant risk. To mitigate these attacks, Buffalo implements a verifiable SA protocol, allowing clients to confirm the integrity of the local update aggregation performed by the server. This ensures that the same global model is consistently sent back to all clients who contributed to filling the buffer.

• We evaluate **Buffalo** through real-world simulations, comparing its performance against existing solutions adapted for BAsyncFL, using benchmark datasets as well as a novel medical dataset [16]. Our protocol undergoes both theoretical analysis and experimental testing across three real datasets. The results highlight **Buffalo**'s practicality and effectiveness, particularly in medical applications.

2 RELATED WORK

SA for SyncFL. SA has been widely deployed in FL systems to guarantee the privacy of local models. SecAgg [8] is the first single-server SA protocol, using Shamir's secret sharing and masking techniques. Subsequent enhancements include SecAgg+ [5] and ACORN [4], utilizing sparse graphs and lattice-based masking mechanisms. DPSecAgg [43] employs packed secret sharing mechanisms over lattice-based masks. Similarly, LightSecAgg [42] utilizes packed Shamir secret sharing techniques over local inputs. The above solutions manage client dropouts and failures at the cost of system efficiency (e.g., client over-selection).

Recent works have aimed to reduce the number of communication rounds required in the protocol and the trust put on the server. Flamingo [29] and LERNA [27] introduce a new role among the clients, called committee members—referred to as assistants in **Buffalo**—to mitigate client communication complexity. However, this design introduces additional overhead. In Flamingo [29], committee members incur 560ms of CPU time and over 1MB of bandwidth, as the protocol relies on decryptors to reconstruct pairwise masks due to high computational demands. In LERNA [27], security requirements necessitate a committee size of up to 2^{14}. In contrast, **Buffalo** addresses a different challenge by focusing on the Buffered Asynchronous setting and introducing assistants specifically designed for this purpose. Solutions with two or more non-colluding servers allow distributed trust and privacy preservation [2, 13, 38]. ELSA [38] leverages two servers to aggregate local inputs, and detect and filter out boosted gradients to withstand model poisoning attacks. Nevertheless, more-than-one-server designs are not realistic in practice.

SA for BAsyncFL. The aforementioned solutions [4, 5, 8, 27, 29] only operate within a SyncFL framework. In this setting, each selected client submits its updated model in alignment with a well determined FL round. This synchronization typically relies on information shared among the selected clients and/or utilizes round-specific information. LightSecAgg [42], originally designed for synchronous settings, can be adapted to asynchronous environments. However, the entire client population must participate, leading to high computational complexity and communication overhead. DPSecAgg [43] also shows its adaptability to BAsyncFL settings. We choose its asynchronous variant as a comparison baseline in Section 3. To the best of our knowledge, only few FL protocols have been designed for BAsyncFL environments specifically [21, 35].

Recent concurrent works [6, 25] are compatible with BAsyncFL by allowing a single round communication at the client. More specifically, OPA [25] defines the concept of one-shot clients (i.e., clients only participate in the whole FL protocol by submitting their contributions) and uses assistants to help the server during the aggregation key reconstruction. However, OPA does not address the challenge of ensuring aggregation integrity in the presence of a malicious server. Moreover, Willow [6], in addition to enable one-shot clients like in OPA, aims to prevent a malicious server from aggregating a client's contribution several times by introducing an additional party, referred to as the verifier. The latter must certify that the server has added to the buffer each local input at most once. However, such integrity guarantee is weaker than ours since it does not prevent the server from providing distinct aggregated outputs to different clients.

Verifiable SA. SA protocols prevent a curious FL server from learning about the clients' inputs but do not protect against a malicious server that might modify the aggregated model. We thus aim for enabling the clients to verify the correctness of the aggregate computed by the server. The initial solution [48] allows aggregation

*The full version is available here.

correctness verification through a proof generated by the server. However, the communication overhead is proportional to the model size, making the solution impractical in large-scale FL systems. An alternative [20] involves hashing clients' local models for verification. It was later found to be insecure and was fixed in [19]. Subsequently, LightVeriFL [10] utilizes elliptic curve-based secret sharing techniques to enable aggregation integrity, effectively reducing both computation and communication costs, particularly in scenarios involving client dropouts. We adapt their approach.

3 BACKGROUND

In this section, we review both SyncFL and BAsyncFL frameworks, along with SA techniques. Notations can be found in Table 1.

Symbol	Description
n_{tot}	total number of clients
n	buffer size
k	number of assistants
t	number of online honest assistants
d	size of input
$\mathcal{U}$	set of clients s.t. $\lvert\mathcal{U}\rvert = n_{tot}$
$\mathcal{U}_{\text{BUFF}}$	set of clients in the buffer s.t. $\lvert\mathcal{U}_{\text{BUFF}}\rvert = n$
$\mathcal{K}$	set of assistants s.t. $\lvert\mathcal{K}\rvert = k$
x_u	scalar input of client u
$\vec{x}_u$	vector input of client u
$\langle x_u \rangle$	protected scalar input of client u
$\langle \vec{x}_u \rangle$	protected vector input of client u
x	scalar aggregate
$\vec{x}$	vector aggregate
m	size of LWE secret key
τ_u	current FL round of client u
$[s]$	share of a secret s
p, q	prime numbers
N	JL modulus
λ	security parameter
ρ	input sparsity parameter
$\xleftarrow{R}$	chosen uniformly at random

Table 1: Notations

Synchronous FL. FL consists of a distributed ML framework where a set $\mathcal{U}$ of clients ($|\mathcal{U}| = n_{tot}$) collaboratively trains a global model $\vec{x} \in \mathbb{R}^d$ under the guidance of a FL server. One of the first and popular methods used to train an FL model is FedAvg [32]. With FedAvg, at each FL round τ, the server selects a subset $\mathcal{U}^{(\tau)} \subseteq \mathcal{U}$ of clients ($|\mathcal{U}^{(\tau)}| = n \leq n_{tot}$) through *client selection*. Each client $u \in \mathcal{U}^{(\tau)}$ trains the model $\vec{x}_{u,\tau}$ on its private local data $\mathcal{D}_u$, for example through Stochastic Gradient Descent (SGD) [39], and forwards this updated model $\vec{x}_{u,\tau}$ to the server. When the server receives the updated models from all clients in $\mathcal{U}^{(\tau)}$, it proceeds to the aggregation step by computing the average of these models and updating the round counter as follows:

$$\vec{x}_\tau \leftarrow \frac{1}{n} \sum_{u \in \mathcal{U}^{(\tau)}} \vec{x}_{u,\tau} \text{ and } \tau \leftarrow \tau + 1$$

This iteration proceeds until the global model $\vec{x}$ exhibits some desired level of accuracy. This approach requires a SyncFL setting whereby FL clients should be synchronized and participate on a round-by-round basis. Usually, in such a setting, $n_{tot} \in [10^6, 10^{10}]$ and $n \in [50, 5000]$ [17].

Buffered Asynchronous FL. SyncFL settings are usually slowed down by *stragglers*, i.e. slow clients [35]. Specifically, a FL round is completed only when all selected clients have sent their updated model. Hence, the impact of stragglers might become significant, especially when the set of clients in the system is heterogeneous. To mitigate such a problem, some systems [9] employ *client over-selection*, where the size of the subset of selected clients is usually increased by 30% in order to reach the actual sufficient number of model updates to run FedAvg. This means that, to execute FedAvg with 1000 client inputs, 1300 clients are selected, and the FL round concludes whenever the server receives 1000 updates from the fastest clients.

As an alternative, BAsyncFL is proposed to remove the need for synchronization and hence, to avoid the effect of late arrivals. FedBuff [35] has been introduced as a BAsyncFL framework whereby the FL server collects in a buffer local models received from clients, and updates the global model whenever this buffer is full. As reported in Figure 2, each client $u \in \mathcal{U}'$ generates its local update $\vec{x}_{u,\tau_u}$ during its local round τ_u. The training round is specific to each client: given another client $v \neq u$, $\tau_v \neq \tau_u$. When the first n clients fill the buffer $\mathcal{U}_{\text{BUFF}}$, the server computes the aggregate and resets the buffer. This process is repeated until a convergence criterion is reached. In this setting, stragglers' inputs are still taken into account as they will eventually fill a buffer.

Figure 1 illustrates SyncFL and BAsyncFL environments. In a SyncFL setting (*part a*), the server randomly selects two clients, here clients 1 and 3 (*step 1*). Then, the selected clients locally train and produce updated local models $\vec{x}_{1,\tau}$ and $\vec{x}_{3,\tau}$, respectively (*step 2*). Finally, the server aggregates these local models (*step 3*). In a BAsyncFL setting (*part b*), all available clients start local training asynchronously (*step 1*). Here, clients 1 and 3 are the fastest to finish their training. They send their updated local models $\vec{x}_{1,\tau_1}$ and $\vec{x}_{3,\tau_3}$, respectively, to fill the buffer on the server. Once the buffer is full, the server aggregates the received local models (*step 3*).

Secure Aggregation. Several studies [33, 41] have shown that, although FL clients train the model locally and keep their datasets $\mathcal{D}_u$ private in their premises, model updates that are shared with the FL server do leak information about the local datasets. Hence, local model updates should also remain confidential even against the FL server. As already shown in [5, 8, 27, 29, 30], the main solution to prevent such a leakage is the use of SA which enables the FL server, named the *aggregator*, to compute the average of model updates, i.e. the global model parameters, without having access to the clients' individual updates.

Towards SA for BAsyncFL. An initial study [42] highlights the incompatibility of current SA protocols with the BAsyncFL setting. This incompatibility stems from the fact that clients know in advance which other clients are participating to the FL round in SyncFL settings, and protect their input accordingly. In general, the n_{tot} clients in $\mathcal{U}$ must learn who are the n clients selected for a given round τ. Moreover, in order to overcome *stragglers*, over-selection is performed, meaning that $n + 0.3 \cdot n$ clients have been actually selected (when fixing over-selection at 30% as suggested above).

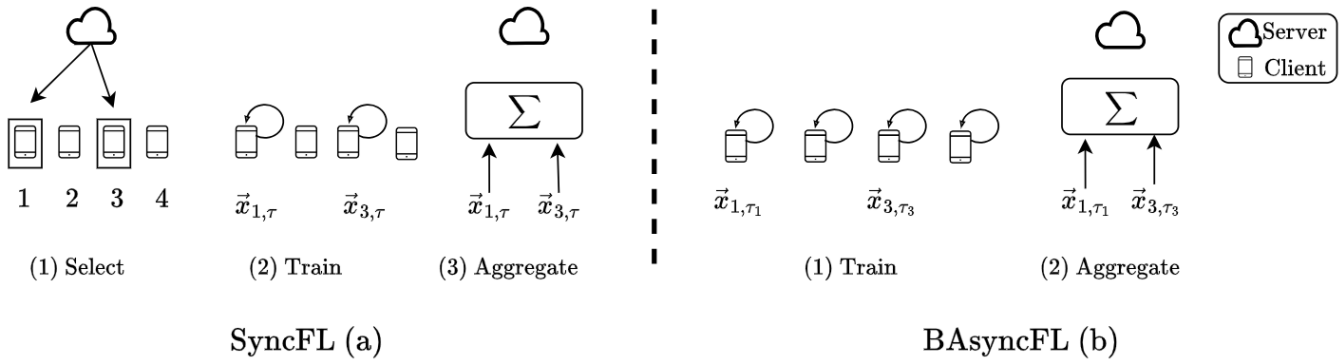

Figure 1: Illustrations of SyncFL (a) and BAsyncFL (b).

Setup: Server initializes the global model $\vec{x}_0$, the empty buffer set $\mathcal{U}_{\text{BUFF}} = \emptyset$ and a set of available clients $\mathcal{U}' = \mathcal{U}$

ServerAggregation($\vec{x}_\tau, \mathcal{U}_{\text{BUFF}}, \mathcal{U}'$)

Server repeats steps 1-4 until convergence criteria is reached

(1) Run **ClientUpdate**($\vec{x}_\tau$) on $\mathcal{U}'$ asynchronously

(2) If Client u's input has been submitted:
  Receive input $\vec{x}_{u,\tau_u}$ from Client u
  $\mathcal{U}_{\text{BUFF}} \leftarrow \mathcal{U}_{\text{BUFF}} \cup \{u\}$
  $i \leftarrow i + 1$

(3) If $i == n$:
  $\vec{x}_\tau \leftarrow \sum_{u \in \mathcal{U}_{\text{BUFF}}} \vec{x}_{u,\tau_u}$
  Reset buffer: $\mathcal{U}_{\text{BUFF}} \leftarrow \emptyset$, $i \leftarrow 0$, $\tau \leftarrow \tau + 1$

(4) Set available clients $\mathcal{U}' = \mathcal{U}_{\text{BUFF}}$

ClientUpdate($\vec{x}$)

Client $u \in \mathcal{U}'$ proceeds as follows

(1) Receive global model $\vec{x}$ from Server

(2) $\vec{y}_{u,0} \leftarrow \vec{x}$

(3) Perform local SGD updates: $\vec{y}_{u,q} = \textbf{LocalSGD}(\vec{y}_{u,0}, q, \eta)$

(4) Compute update difference: $\vec{x}_{u,\tau_u} \leftarrow \vec{y}_{u,0} - \vec{y}_{u,q}$

(5) Return $\vec{x}_{u,\tau_u}$ to Server

Figure 2: FedBuff Algorithm

Once this knowledge has been acquired, each selected client submits their updated input using some information specifically shared with other selected clients. Such a constraint comes against the idea behind buffered asynchronicity where clients train and submit their inputs at their own pace, without following what other clients are doing. Therefore, to successfully develop SA protocols for BAsync FL settings, we must overcome the above constraint by avoiding clients requiring to know which other clients participate to the current round.

Following the last remark, we identify two potential SA protocols that can be easily transformed to be compatible with BASyncFL: LightSecAgg [42] and DPSecAgg [43]. Indeed, in both original schemes, clients do not need to know who is participating to the actual round τ. Nevertheless, those solutions have been first designed for SyncFL, and thus assume that all n_{tot} clients are online during the aggregation phase. Informally, clients secretly share their inputs with all clients to enable successful aggregation. This is due to the fact that clients lack the information related to client participation for a specific round. We recall that $n_{tot} >> n$, hence this design is really inefficient. One way to overcome such a limitation is to let the server provide this missing information to the clients, resulting into an extra communication round. Another way to make LightSecAgg and DPSecAgg compatible in BAsyncFL settings is to introduce several special clients that must remain online, called *assistants*, whose task is to help the server reconstruct the aggregation key and hence the aggregate. This is the direction we choose to follow when designing **Buffalo**.

Aggregation Integrity. LightVeriFL [10] combines homomorphic hash functions, digital signatures and commitments to enable clients to verify whether the server has correctly computed the aggregate, thereby preventing a malicious server to distribute to clients different aggregates obtained from distinct sets of local updates [37]. We choose to use a similar combination of cryptographic tools to guarantee aggregation integrity in **Buffalo**. Nevertheless, LightVeriFL forces to hash the entire client local model at every training round, while it has been observed that most of the client model parameters from one round to another one remain unchanged. When developing our solution, we ensure that clients do not need to recompute the hash value of their entire local input, but to only hash the small parts that differ between two consecutive inputs. Consequently, the computation cost at the client is decreased significantly.

Threat Model. Similar to [29], we assume a malicious adversary that corrupts the server and up to a fraction γ of the total number n_{tot} of clients in the system. As mentioned above, we introduce special clients, called the assistants, in addition to regular clients. Those assistants help the server recover the final aggregate. Consequently, we must consider the involvement of both clients and assistants when elaborating the security proof. We distinguish between the fraction γ_n of corrupted regular clients and the fraction γ_k of corrupted assistants. In particular, we examine the following types of client and assistant failures: *(i)* honest parties that disconnect or are too slow to respond as a result of unstable network conditions, power loss, etc; *(ii)* arbitrary actions by an adversary that controls the server and some clients and assistants. Given a threshold t and the number k of assistants in the system, we require the number of honest and alive assistants to be $t > \frac{2k}{3}$ [8].

As highlighted in [37], the threat model should also account for model inconsistency attacks orchestrated by a malicious server. In such attacks, the server may bypass SA steps by distributing distinct

global models to different clients. While this attack has primarily been demonstrated in SyncFL contexts, it can be countered by embedding the training round number within the encryption of local models, leveraging client synchronization. To address this, we extend the threat model to incorporate aggregation integrity for BAsyncFL environments. Specifically, this involves enabling clients to verify that the received global model corresponds to the aggregate of client inputs from the full buffer. However, it is important to note that this guarantee does not mitigate poisoning attacks originating from malicious clients. Detecting and addressing such poisoned local inputs requires additional robust mechanisms, as outlined in [4]. We consider denial of services attacks and client model poisoning attacks out of scope.

4 BUILDING BLOCKS

In this section, we introduce the main cryptographic primitives utilized as foundational elements in our two protocols. We recall that notations can be found in Table 1.

Secure Aggregation

We are interested in SA schemes that are homomorphic with relation to the secret key sk_u and input x_u:

$$\sum_{u\in\mathcal{U}} \textbf{Protect}(x_u, sk_u) = \textbf{Protect}(-\sum_{u\in\mathcal{U}} sk_u, \sum_{u\in\mathcal{U}} x_u)$$

Here, we describe the Joye-Libert (JL) scheme and a lattice-based scheme relying on the Ring Learning With Errors (RLWE) problem. For sake of simplicity, we denote the latter as the LWE scheme.

Joye-Libert SA. Let us consider n clients and one aggregator. The JL scheme is defined with three algorithms [24]:

- $(sk_0, \{sk_u\}_{u\in[1,n]}, pp) \leftarrow$ **JL.Setup**(λ): Given the security parameter λ, this algorithm generates two large, equal-size prime numbers p and q and sets the modulus $N = pq$. It randomly generates n client secret keys $sk_u \in \mathbb{Z}_{N^2}$ and computes the aggregator secret key $sk_0 = -\sum_{u=1}^{n} sk_u$. Then, it defines a cryptographic hash function $F : \mathbb{Z} \rightarrow \mathbb{Z}^*_{N^2}$. It outputs the $n+1$ secret keys and the public parameters $pp = (N, F)$.
- $y_{u,\tau} \leftarrow$ **JL.Protect**$(pp, sk_u, \tau, x_{u,\tau})$: This algorithm encrypts private input $x_{u,\tau} \in \mathbb{Z}_N$ for the time period τ using the secret key $sk_u \in \mathbb{Z}_{N^2}$, resulting in the ciphertext $y_{u,\tau} = (1 + x_{u,\tau}N) \cdot F(\tau)^{sk_u} \mod N^2$.
- $x_\tau \leftarrow$ **JL.Agg**$(pp, sk_0, \tau, \{y_{u,\tau}\}_{u\in[1,n]})$: This algorithm aggregates the n protected inputs from clients received at the time period τ to obtain $y_\tau = \prod_{u=1}^{n} y_{u,\tau}$. It then decrypts y_τ to recover the plain aggregate $x_\tau = \sum_{u=1}^{n} x_{u,\tau} = (F(\tau)^{-sk_0} \cdot y_\tau - 1)/N \mod N$.

The JL scheme ensures *Aggregator Obliviousness* under the Decision Composite Residuosity (DCR) assumption in the random oracle model and assuming that each client u encrypts only one input $x_{u,\tau}$ per time period τ [24, 36].

LWE-based SA. Several SA solutions [4] have their security relying on the RLWE assumption. Such SA schemes are parameterized by a ring R of degree m over $\mathbb{Z}$, an integer modulus $q > 0$ defining a quotient ring $R_q = R/qR$, and two distributions χ_s, χ_e over R. Let d be the length of the client's vector input $\vec{x}_u$. Let us consider n clients and one aggregator. The LWE scheme is defined with three algorithms [4]:

- $pp \leftarrow$ **LWE.Setup**(λ): Given the security parameter λ, this algorithm generates the public matrix $A \in \mathbb{Z}_p^{m\times d}$ which is defined as the public parameters $pp = A$.
- $\langle\vec{x}_u\rangle \leftarrow$ **LWE.Protect**$(pp, \vec{s}_u, \vec{x}_u)$: To encrypt a vector $\vec{x}_u \in \mathbb{Z}^d$, the algorithm first samples two vectors, namely the client's secret key $\vec{s}_u \leftarrow \chi_s \subseteq \mathbb{Z}_q^m$ and the error vector $\vec{e} \leftarrow \chi_e \subseteq \mathbb{Z}_q^m$. The ciphertext $\langle\vec{x}_u\rangle \in \mathbb{Z}_D^m$ is computed as follows: $\langle\vec{x}_u\rangle = A\vec{s}_u + D\cdot\vec{e} + \vec{x}_u \mod q$, where D is the chipertext modulus.
- $\vec{x} \leftarrow$ **LWE.Agg**$(pp, \vec{s}_0, \{\langle\vec{x}_u\rangle\}_{u\in[1,n]})$: Let $\vec{s}_0 = \sum_{u\in[1,n]} \vec{s}_u$ be the aggregation key. This algorithm computes the aggregate $\vec{x}$ from the n chipertexts using $\vec{s}_0$ as follows: $\vec{x} = (\sum_{u\in[1,n]}\langle\vec{x}_u\rangle - A\vec{s}_0) \mod D$.

The RLWE scheme guarantees *Aggregator Obliviousness* under the Hint-RLWE problem, assuming that each client u encrypts only one input $\vec{x}_u$ per round using the same secret key $\vec{s}_u$ [4, 26].

Other Cryptographic Primitives. In our protocol, we also employ the following cryptographic primitives for secure communication among clients and server.

Through a secure Key Agreement protocol, denoted as **KA**, any client will use their secret key and the public key of another client to produce a shared secret key. In practice, it can be instantiated with a Diffie–Hellman KA protocol followed by a key derivation function [8].

Authenticated Encryption, denoted as **AE**, combines confidentiality and integrity guarantees for messages exchanged between two parties. It consists of a key generation algorithm that outputs a secret key, an encryption algorithm **AE.Enc** that takes as input the secret key and a message, and outputs a ciphertext, and a decryption algorithm **AE.Dec** that takes as input the ciphertext and the secret key, and outputs the original plaintext.

We also consider a digital signature scheme that is existentially unforgeable under chosen message attacks. A signing algorithm **Sig.Sign** takes as input a secret key and a message, and outputs a signature, and a verification algorithm **Sig.Ver** takes as input a public key, a signature and a message, and outputs '1' if and only if the signature is valid for the given message. In practice, it can be instantiated with the Elliptic-Curve Digital Signature Algorithm (ECDSA) followed by a key derivation function [23].

Aggregation Integrity

To ensure the aggregation operation's integrity, hash functions, signatures, commitments and secret sharing techniques are employed in LightVeriFL [10]. We choose to use the Threshold ElGamal (TEG) encryption scheme instead of the Shamir Secret Sharing (SS) scheme (see Appendix A) to reach constant computational costs. In this subsection, we describe the hash process presented in [10], and show how we can benefit from its homomorphic properties to improve the aggregation check step.

Threshold ElGamal Encryption. The following TEG scheme is additively homomorphic with respect to messages (represented as EC points) [28]. The TEG scheme uses the Shamir SS scheme [40], where the algorithm **SS.Share** shares a secret s into n shares while

the algorithm **SS.Recon** allows recovering this secret s by collecting t out of the n shares (see Appendix A). The TEG scheme is defined by four algorithms:

• $(pk, \{[sk]_u\}_{u\in\mathcal{U}}) \leftarrow$ **TEG.Setup**$(t, \mathcal{U}, \lambda)$: Let t be the threshold for successful secret reconstruction and $\mathcal{U}$ be the set of participants. The algorithm selects $sk \in \mathbb{Z}_p$ where p is a prime number, and computes $pk = g^{sk}$ where g is a generator of the cyclic group built from some elliptic curve. It then shares sk into $[sk]_u$ for $u \in \mathcal{U}$ using the algorithm **SS.Share**.

• $\langle m\rangle \leftarrow$ **TEG.Encrypt**(pk, m): To encrypt the message m, the algorithm computes the ciphertext $\langle m\rangle = (c_0, c_1) = (g^r, m \cdot pk^r)$ where $r \in \mathbb{Z}_p$ is the encryption randomness.

• $[m]_u \leftarrow$ **TEG.PartialDecrypt**$([sk]_u, \langle m\rangle)$: The partial decryption $[m]_u$ of the message m is computed as follows: $[m]_u = (c_0^{[sk]_u}, c_1)$ where $\langle m\rangle = (c_0, c_1)$.

• $m \leftarrow$ **TEG.Decrypt**$(\{(u, [m]_u)\}_{u\in\mathcal{U}'})$: The algorithm computes the Lagrange interpolation on the exponent as follows: $c_0^{sk} = \prod_{u\in\mathcal{U}'} (c_0^{[sk]_u})^{\lambda_u}$ where the coefficients λ_u are defined in the algorithm **SS.Recon**, and $\mathcal{U}' \subseteq \mathcal{U}$ such that $|\mathcal{U}'| \geq t$. The algorithm finally computes the original message m as follows: $m = (c_0^{sk})^{-1} \cdot c_1$.

The security of the TEG scheme relies on m being computationally hidden w.r.t. the Discrete Logarithm (DL) problem [28].

Homomorphic Hashing. Let us consider the cyclic group $\mathbb{G}$ of prime order p with generator g. Given d distinct elements $g_1, \ldots, g_d \in \mathbb{G}$, the hash of a vector $\vec{x}_u \in \mathbb{Z}_p^d$ is defined as follows [7]:

$$h_u = H(\vec{x}_u) = \prod_{i=1}^{d} g_i^{\vec{x}_u[i]} \tag{1}$$

where $\vec{x}_u[i]$ represents the ith element of the vector $\vec{x}_u$. This hash function H is additively homomorphic, meaning that for any two vectors $\vec{x}_u, \vec{x}_v$, we have $H(\vec{x}_u + \vec{x}_v) = H(\vec{x}_u) \cdot H(\vec{x}_v)$. Additionally, the hash function H is incrementally computable. Specifically, let two vectors $\vec{x}_u$ and $\vec{x}_v$ differ in only one element: $\vec{x}_u[j] \neq \vec{x}_v[j]$ and $\forall i \neq j, \vec{x}_u[i] = \vec{x}_v[i]$. Thanks to the homomorphic property of H, given $h_u = H(\vec{x}_u)$, one can easily compute h_v using h_u. Indeed, $h_v = H(\vec{x}_v) = h_u \cdot g_j^{-\vec{x}_u[j]+\vec{x}_v[j]}$. Hence, instead of computing h_v from scratch using Eq. 1 (as it is done in LightVeriFL [10]), one can save exponentiation calculations by using the aforementioned technique. We thus define the function H_{INC} as follows:

$$H_{\text{INC}}(\vec{x}_v, \vec{x}_u, h_u) = \begin{cases} h_u \cdot g_j^{-\vec{x}_u[j]+\vec{x}_v[j]} & \text{if } \vec{x}_u[j] \neq \vec{x}_v[j] \\ \text{do nothing} & \text{otherwise} \end{cases} \tag{2}$$

To improve even further the efficiency of the hashing process, H and H_{INC} can be implemented using Elliptic Curve (EC) points [10].

In our solution, we promote the use of H_{INC} as much as possible. Drawing from extensive research in distributed optimization [18, 44, 45], which shows that not all local ML parameters (here, gradients) change in each client round, we exploit parameter sparsification and quantization to avoid the full computation of Eq. 1. Utilizing this assumption of sparsity, each client computes the input hash for a given round τ_u and stores the local model $\vec{x}_{u,\tau_u}$. In the next client round $\tau_u + 1$, the client assesses the changes between $\vec{x}_{u,\tau_u}$ and $\vec{x}_{u,\tau_u+1}$, and employs the incremental hashing property to compute partial hashes only for the changed parameters using Eq. 2. Let d be the size of the model and ρ denote the fraction of parameters that changed between two client rounds τ_u and $\tau_u + 1$. The aforementioned method requires only $O(d)$ comparisons and $O(\lfloor \rho \cdot d \rfloor)$ EC exponentiations, thus proving to be more efficient than directly hashing the input $\vec{x}_{u,\tau_u+1}$, despite incurring the cost of storing the previous local model input $\vec{x}_{u,\tau_u}$ in practice.

Commitments. A commitment scheme that is perfectly hiding and computationally binding under the DL assumption is needed for aggregation verification. A committing algorithm **COM.Commit** takes as input a message and a witness, and outputs a commitment, and an opening algorithm **COM.Open** takes as input a commitment, a message and a witness, and outputs '1' if and only if the commitment is valid for the given message. To prove the security of **Buffalo** against malicious adversaries, a specific commitment scheme with additional properties of *equivocability* and *extractability* is necessary [46]. Equivocation means that there exist a trapdoor value and an algorithm **COM.Equiv** allowing the simulator to open a commitment to an arbitrary value. Extractability means that the simulator can leverage a trapdoor value to extract the message from a commitment by invoking the algorithm **COM.Ext**. In practice, it can be instantiated with commitments schemes from [1].

5 OUR FL PROTOCOL

We present **Buffalo**, a SA protocol in the context of a BAsyncFL setting. The design principles of **Buffalo** are the following:

(1) As opposed to the synchronized setting, clients who contribute to a given buffer are not known in advance. Hence, the aggregation key cannot be pre-computed or pre-defined. Therefore, **Buffalo** uses an ephemeral aggregation key to aggregate local inputs once the buffer is full, based on the buffer set $\mathcal{U}_{\text{BUFF}}$. This aggregation key is computed through the help of assistants. They are special clients who receive other clients' contributions to the aggregation key, and collaboratively construct the final version of that key. It is worth noting that, as opposed to the state-of-the-art solutions that make use of assistants to reduce/optimize some computational costs [27, 29], the purpose here is to deal with potential clients turning offline after uploading their contributions to the buffer.

(2) In addition to being the first SA solution for the buffered asynchronous setting, **Buffalo** exhibits good performance results at the client thanks to the use of a lattice-based batched encryption scheme to encrypt model parameters, as proposed in [4]. The number of RLWE coefficients usually is in $[2^{11}, 2^{12}]$, and each of them needs to be protected and further aggregated. In order to avoid clients sharing as many partial keys as the number of ring coefficients, **Buffalo** introduces an additional layer of protection. Informally, each client's input is first encrypted using a LWE key and the LWE key is further encrypted using a JL key. This key is then secretly shared with all the assistants, only. Thanks to this approach, **Buffalo** achieves better client computation by sharing a single scalar value instead of a vector value. Such a performance improvement is very important in the context of an asynchronous setting wherein saving resources becomes fundamental and taking into account inputs from slow clients pivotal.

(3) **Buffalo** ensures that every client $u \in \mathcal{U}_{\text{BUFF}}$ can verify the aggregation of the global model obtained by the server, and in

particular, that their input $\vec{x}_{u,\tau_u}$ has been included in the process. Each client generates a homomorphic hash over the local update $\vec{x}_{u,\tau_u}$ and commits to it. Then, both the hash value and the commitment witness are masked with some randomness. Instead of secretly sharing those random masks as in [10], we choose to use the TEG scheme to encrypt them since this scheme provides constant encryption time. The resulting TEG ciphertexts are sent to the assistants who are in charge of applying threshold decryption to obtain the masks. This design choice allows constant communication and computation costs compared to using the classical SS scheme as in [10]. Specifically, **Buffalo** reduces the regular client computation by a factor of k^2 and communication by a factor of k.

(4) We leverage the incremental nature of the homomorphic hash function H_{INC} to reduce computational overhead. As demonstrated in [10], computing a homomorphic hash function over a locally updated model of dimension d remains a significant bottleneck. While most steps in the protocol can be parallelized during the training of local models [30, 42], the critical hashing step must await the end of the training of the local model $\vec{x}_{u,\tau_u}$ to be executed [10]. Hence, employing the incremental hashing aspect is beneficial in the context of BASyncFL.

Description. The main protocol steps, reported in Figure 3, is defined over three phases: the Setup phase during which clients first register to the server and generate their keying material, the Online phase during which aggregation occurs asynchronously (i.e. whenever the buffer is full), and the Verification phase during which the aggregate correctness is checked. For the sake of space, operations denoted as executed on an input pair are actually operations executed twice, once for each input taken individually (e.g. line 6 in **Online - Round 3**: **TEG.PartialDecrypt** on the pair of encrypted hash and witness masks).

In the Setup phase, similar to other solutions [8, 29], each client creates a pair of secret and public keys and a pair of signing and verification keys, and sends the public and verification keys to the Public Key Infrastructure (PKI) to register them. Furthermore, each pair of client and assistant u and i establish a pairwise key $c_{u,i}$. A Trusted Dealer (TD)[†] generates the public parameters, in particular the JL modulus N and the public LWE matrix A, and registers them to the PKI.

The Online phase is split into three rounds:

(**Round 1**) Each client u in the set $\mathcal{U}$ generates a secret LWE key $\vec{s}_{u,\tau_u}$ and a secret JL key sk_{u,τ_u} at their local round τ_u. The JL key sk_{u,τ_u} protects the LWE key $\vec{s}_{u,\tau_u}$ using a fixed 'round' $\tau = \tau_0$. Each private input vector $\vec{x}_{u,\tau_u}$ is protected with the LWE key $\vec{s}_{u,\tau_u}$. The server collects the n first submitted protected inputs and encrypted secret LWE keys. The clients owning those elements are finally included in a buffer set $\mathcal{U}_{\text{BUFF}} \subseteq \mathcal{U}$. Each client u then secretly shares their key sk_{u,τ_u} such that t out of these n shares can reconstruct it using the SS scheme. They send the shares of sk_{u,τ_u}, encrypted using the AE scheme, to the server. The latter broadcasts those elements to all assistants in the set $\mathcal{K}$.

To prepare the Verification phase, the clients generate random EC points r_{u,τ_u} and $(z_{u,\tau_u}, z'_{u,\tau_u})$. They compute the local model hash h_{u,τ_u} (using Eq. 2 except at their first own round $\tau_u = 1$), and commit to the latter using the witness r_{u,τ_u}. They also mask h_{u,τ_u} and r_{u,τ_u} using the values z_{u,τ_u} and z'_{u,τ_u}, respectively. The clients then protect the hash and witness masks $(z_{u,\tau_u}, z'_{u,\tau_u})$ using the TEG scheme, and sign the resulting encryptions along with the commitment c_{u,τ_u}. They send both masked elements, protected masks, signatures and commitments to the server.

(**Round 2**) The assistants follow the same consistency check process as in [8, 29] over the set $\mathcal{U}_{\text{BUFF}}$. The latter is signed by each assistant and the resulting signature is forwarded to the server, which passes it on to all assistants in $\mathcal{K}$. This step ensures consistency over the set $\mathcal{U}_{\text{BUFF}}$ across the entire system.

(**Round 3**) Each assistant in $\mathcal{K}_2 \subseteq \mathcal{K}$ receives the encrypted shares of the secret JL keys of clients in $\mathcal{U}_{\text{BUFF}}$. It computes the aggregated value $[sk_0]_i$, that is its share of the server's JL aggregation key sk_0. The assistant then forwards it to the server. The latter must receive at least t of these aggregated shares in order to successfully reconstruct sk_0. Once this key is retrieved, the server can have access to the LWE aggregation key $\vec{s}_0$ through the aggregation of the clients' encrypted LWE keys. The server finally recovers the aggregated model $\vec{x}$ using the key $\vec{s}_0$. To prepare the Verification phase, at least t remaining online assistants in $\mathcal{K}_3 \subseteq \mathcal{K}$ help the server construct the aggregated masking pair (z_0, z'_0), which enables the construction of the aggregated hash h_0 and witness r_0.

In the Verification phase, all the clients in $\mathcal{U}_{\text{BUFF}}$ can verify if the server has correctly aggregated their inputs. More precisely, each client receives and aggregates the commitments of all other clients in $\mathcal{U}_{\text{BUFF}}$, verifying the aggregated commitment c_0 using the aggregated witness r_0 and the aggregated hash value h_0. Finally, the client computes the hash h_{agg} of the received global model and checks its consistency with h_0. Notably, this verification step does not require synchronization, as the aggregated information is collected beforehand. Consequently, clients can perform the verification step to ensure aggregation integrity prior to initiating a new round of local training.

Security Analysis. We briefly analyse the security of **Buffalo**. The full, hybrid-based, proof is given in Appendix B. Let the set of corrupted clients be denoted as $C \subset \mathcal{U}$, where $|C| = \gamma$, the set of corrupted regular clients as C_n, where $|C_n| = \gamma_n$, and the set of corrupted assistants as $C_k \subset \mathcal{K}$, where $|C_k| = \gamma_k$.

Note that we make the distinction between the fraction of corrupted regular clients (γ_n) and the fraction of corrupted decryptors (γ_k), similar to [29]. Those metrics already reflect the total number of corrupted parties (γ). More specifically, the majority of corrupted parties are regular clients (hence γ_n is close to γ) and not assistants (γ_k is upper bound with $2^{\theta(\lambda)}$ where λ is the security parameter). While it is true that assistants help achieve SA in the BAsync setting, these parties can be selected from the set of regular clients, are considered as being dynamic, and can drop.

The security of the LWE and JL schemes guarantees that parties in C cannot distinguish the protected input $\vec{x}_{u,\tau_u}$ (protected with a LWE key) and the LWE key $\vec{s}_{u,\tau_u}$ (protected with a JL key) of an honest client u from random values. The security of the SS scheme guarantees that assistants in C_k cannot distinguish the protected JL key sk_{u,τ_u} of an honest client u from a random value. More precisely, the security of the SS scheme ensures that if at most $t-1$ assistants in C_k have access to shares of $[sk_0]_u$ (i.e. each assistant has at most one

[†]Alternative methods exist in decentralized settings to avoid the participation of a TD [12].

Parties: Server and clients in $\mathcal{U}$, such that $|\mathcal{U}| = n_{tot}$, where *assistants* belong to $\mathcal{K} \subseteq \mathcal{U}$ such that $|\mathcal{K}| = k$.
Public Parameters: input domain $\mathbb{Z}_L^d$; buffer set $\mathcal{U}_{\text{BUFF}}$ whose size is $|\mathcal{U}_{\text{BUFF}}| = n$; security parameter λ for cryptographic primitives; secret sharing threshold t.
Prerequisites: For $\mathcal{K}$ and $r \in \{2, 3\}$, we denote $\mathcal{K}_r$ the set of *assistants* that execute **Round** r, and we denote $\mathcal{K}'_r$ the set of *assistants* that completed without dropping out. It holds that $\mathcal{K}'_r \subseteq \mathcal{K}_r \subseteq \mathcal{K}_{r-1}$ for all $r \in \{2, 3\}$.

Setup

Client $u \in \mathcal{U}$:
Key registration
(1) $(c_u^{PK}, c_u^{SK}) \leftarrow$ **KA.Gen**(pp^{KA})
(2) $\{d_u^{SK}, d_u^{PK}\}_{u \in \mathcal{U}} \leftarrow$ **Sig.Setup**(λ)
(3) Register c_u^{PK} and d_u^{PK} to PKI
Assistant channel key setup
(4) $\forall i \in \mathcal{K}, c_{u,i} \leftarrow$ **KA.Agree**(c_u^{SK}, c_i^{PK})
Trusted Dealer (TD):
(5) $(\perp, \perp, pp^{\text{JL}}) \leftarrow$ **JL.Setup**(λ)
(6) $pp^{\text{LWE}} \leftarrow$ **LWE.Setup**(λ)
(7) $(pk, \{[sk]_i\}_{i \in \mathcal{K}}) \leftarrow$ **TEG.Setup**$(t, \mathcal{K}, \lambda)$
(8) Register pp^{JL}, pp^{LWE} and pk to PKI
(9) Send $[sk]_i$ to Assistant $i \in \mathcal{K}$
Assistant $i \in \mathcal{K}$:
(10) Receive $[sk]_i$ from TD

Online - Round 1

Client $u \in \mathcal{U}$:
Input protection and authentication
(1) $\vec{s}_{u,\tau_u} \xleftarrow{R} \chi_s$ *// Generate LWE secret key for round τ_u*
(2) $\langle \vec{x}_{u,\tau_u} \rangle \leftarrow$ **LWE.Protect**$(pp^{\text{LWE}}, \vec{s}_{u,\tau_u}, \vec{x}_{u,\tau_u})$ *// Protect input using **LWE***
(3) $h_{u,\tau_u} = H_{\text{INC}}(\vec{x}_{u,\tau_u}, \vec{x}_{u,\tau_u-1}, h_{u,\tau_u-1})$ *// Compute input hash*
(4) $(z_{u,\tau_u}, z'_{u,\tau_u}, r_{u,\tau_u}) \xleftarrow{R} E(\mathbb{F}_p)$ *// Generate hash mask, witness mask and witness*
(5) $\tilde{h}_{u,\tau_u} = h_{u,\tau_u} + z_{u,\tau_u}$ *// Mask local input hash*
(6) $c_{u,\tau_u} \leftarrow$ **COM.Commit**$(h_{u,\tau_u}, r_{u,\tau_u})$ *// Commit to input hash*
(7) $\sigma_{u,\tau_u} \leftarrow$ **Sig.Sign**(d_u^{SK}, c_{u,τ_u}) *// Sign commitment*
Key protection and authentication
(8) $sk_{u,\tau_u} \xleftarrow{R} \mathbb{Z}_{N^2}$ *// Generate JL secret key for round τ_u*
(9) $\langle \vec{s}_{u,\tau_u} \rangle \leftarrow$ **JL.Protect**$(pp^{\text{JL}}, sk_{u,\tau_u}, \tau_0, \vec{s}_{u,\tau_u})$ *// Protect LWE key using JL*
(10) $\{(i, [sk_{u,\tau_u}]_i)\}_{i \in \mathcal{K}} \leftarrow$ **SS.Share**$(sk_{u,\tau_u}, t, \mathcal{K})$ *// Secret share JL secret key*
(11) $\forall i \in \mathcal{K}, \epsilon_{u,i} \leftarrow$ **AE.Enc**$(c_{u,i}, u \,||\, i \,||\, [sk_{u,\tau_u}]_i)$ *// Encrypt JL key shares*
(12) $\tilde{r}_{u,\tau_u} = r_{u,\tau_u} + z'_{u,\tau_u}$ *// Mask witness*
(13) $\langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle \leftarrow$ **TEG.Protect**$(pk, (z_{u,\tau_u}, z'_{u,\tau_u}))$
// Protect hash and witness masks using TEG
(14) $\sigma'_{u,\tau_u} \leftarrow$ **Sig.Sign**$(d_u^{SK}, \langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle)$ *// Sign encrypted masks*
(15) Send $\langle \vec{x}_{u,\tau_u} \rangle$, $\langle \vec{s}_{u,\tau_u} \rangle$, $\{\epsilon_{u,i}\}_{i \in \mathcal{K}}$, c_{u,τ_u}, σ_{u,τ_u}, σ'_{u,τ_u}, $\tilde{h}_{u,\tau_u}$, $\langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle$, $\tilde{r}_{u,\tau_u}$ to Server
Server:
(16) Collect $\{\langle \vec{x}_{u,\tau_u} \rangle, \langle \vec{s}_{u,\tau_u} \rangle, \{\epsilon_{u,i}\}_{i \in \mathcal{K}}, c_{u,\tau_u}, \sigma_{u,\tau_u}, \sigma'_{u,\tau_u}, \tilde{h}_{u,\tau_u}, \langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle, \tilde{r}_{u,\tau_u}\}_{u \in \mathcal{U}_{\text{BUFF}}}$
(17) If $|\mathcal{U}_{\text{BUFF}}| < t$, abort; otherwise, broadcast $\{\epsilon_{u,i}\}_{u \in \mathcal{U}_{\text{BUFF}}} \mathcal{U}_{\text{BUFF}}$, $\{\sigma'_{u,\tau_u}, \langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle\}_{u \in \mathcal{U}_{\text{BUFF}}}$ to Assistant $i \in \mathcal{K}$

Online - Round 2

Assistant $i \in \mathcal{K}_2$:
Consistency check
(1) Receive $\{\epsilon_{u,i}, \langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle, \sigma'_{u,\tau_u}\}_{u \in \mathcal{U}_{\text{BUFF}}}$ and $\mathcal{U}_{\text{BUFF}}$
(2) Assert that $|\mathcal{U}_{\text{BUFF}}| = n$; if not, abort
(3) $\sigma''_i \leftarrow$ **Sig.Sign**$(d_i^{SK}, \mathcal{U}_{\text{BUFF}})$
(4) $\forall u \in \mathcal{U}_{\text{BUFF}}, 1 \leftarrow$ **Sig.Ver**$(d_u^{PK}, \langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle, \sigma'_{u,\tau_u})$; if not, abort
(5) Send σ''_i to Server
Server:
(6) Assert that $|\mathcal{K}'_2| \geq t$; if not, abort
(7) Collect $\{\sigma''_i\}_{i \in \mathcal{K}'_2}$ and broadcast to Assistants in $\mathcal{K}'_2$

Online - Round 3

Assistant $i \in \mathcal{K}_3$:
Aggregation key share construction
(1) Receive $\{\sigma''_j\}_{j \in \mathcal{K}'_2}$
(2) $\forall j \in \mathcal{K}'_2, 1 \leftarrow$ **Sig.Ver**$(d_j^{PK}, \mathcal{U}_{\text{BUFF}}, \sigma''_j)$; if not, abort
(3) $\forall u \in \mathcal{U}_{\text{BUFF}}, [sk_{u,\tau_u}]_i \leftarrow$ **AE.Dec**$(c_{i,u}, u \,||\, i \,||\, \epsilon_{u,i})$ *// Decrypt JL secret key*
(4) $[sk_0]_i = \sum_{u \in \mathcal{U}_{\text{BUFF}}} [sk_{u,\tau_u}]_i$ *// Compute share of JL aggregation key*
(5) $\langle (z_0, z'_0) \rangle = \prod_{u \in \mathcal{U}_{\text{BUFF}}} \langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle$ *// Compute aggregated encrypted hash and witness masks*
(6) $[(z_0, z'_0)]_i =$ **TEG.PartialDecrypt**$([sk]_i, \langle (z_0, z'_0) \rangle)$ *// Compute partial TEG decryption*
(7) Send $[sk_0]_i$ and $[(z_0, z'_0)]_i$ to Server
Server:
Aggregation
(8) Assert that $|\mathcal{K}'_3| \geq t$; if not, abort
(9) Collect $\{[sk_0]_i, [z_0]_i\}_{i \in \mathcal{K}'_3}$
(10) $sk_0 \leftarrow$ **SS.Recon**$(\{[sk_0]_i\}_{i \in \mathcal{K}'_3}, t)$ *// Reconstruct JL aggregation key*
(11) $\vec{s}_0 \leftarrow$ **JL.Agg**$(pp^{\text{JL}}, sk_0, \tau_0, \{\langle \vec{s}_{u,\tau_u} \rangle\}_{u \in \mathcal{U}_{\text{BUFF}}})$ *// Compute LWE aggregation key*
(12) $\vec{x} \leftarrow$ **LWE.Agg**$(pp^{\text{LWE}}, \vec{s}_0, \{\langle \vec{x}_{u,\tau_u} \rangle\}_{u \in \mathcal{U}_{\text{BUFF}}})$ *// Compute aggregated input*
(13) $(z_0, z'_0) \leftarrow$ **TEG.Decrypt**$(\{[(z_0, z'_0)]_i\}_{i \in \mathcal{K}'_3})$ *// Compute full TEG decryption*
(14) $h_0 = \prod_{u \in \mathcal{U}_{\text{BUFF}}} \tilde{h}_{u,\tau_u} - z_0$ *// Unmask aggregated input hash*
(15) $r_0 = \prod_{u \in \mathcal{U}_{\text{BUFF}}} \tilde{r}_{u,\tau_u} - z'_0$ *// Unmask aggregated witness*
(16) Broadcast $\vec{x}$, $\{c_{u,\tau_u}, \sigma_{u,\tau_u}\}_{u \in \mathcal{U}_{\text{BUFF}}}$, h_0, r_0 to Client $u \in \mathcal{U}_{\text{BUFF}}$

Verification

Client $u \in \mathcal{U}_{\text{BUFF}}$:
(1) Receive $\{c_{v,\tau_v}\}_{v \in \mathcal{U}_{\text{BUFF}}}$, $\vec{x}$, h_0 and r_0
(2) $\forall v \in \mathcal{U}_{\text{BUFF}}, 1 \leftarrow$ **Sig.Ver**$(d_v^{PK}, c_{v,\tau_v}, \sigma_{v,\tau_v})$; if not, abort
(3) $c_0 = \prod_{v \in \mathcal{U}_{\text{BUFF}}} c_{v,\tau_v}$ *// Compute aggregated commitment*
(4) Check **COM.Commit**$(h_0, r_0) = c_0$; if not, abort
(5) $h_{agg} = H(\vec{x})$ *// Compute global model hash*
(6) Check if $h_{agg} = h_0$; if not, abort

Figure 3: Buffalo protocol steps.

share and $|C_k| < t$), then they cannot reconstruct the key. The TEG scheme ensures that the server, along with assistants in the set C_k, cannot distinguish the protected masking pair $\langle (z_{u,\tau_u}, z'_{u,\tau_u}) \rangle$ from random values. Thus, the security of the TEG scheme guarantees that, given the protected pair $\langle (z_0, z'_0) \rangle$, up to $t-1$ assistants in C_k cannot collaboratively decrypt it.

Moreover, when the server is a *malicious* adversary, it can try to convince some honest assistants that the set of clients in the buffer is $\mathcal{U}_{\text{BUFF}}$ while indicating to other honest assistants that the set of clients in the buffer is $\mathcal{U}^*_{\text{BUFF}} = \mathcal{U}_{\text{BUFF}} \setminus \{u\}$ for a client u. If this occurs, the server can reconstruct sk_0 for $\mathcal{U}_{\text{BUFF}}$ and sk^*_0 for $\mathcal{U}^*_{\text{BUFF}}$. Then, it can compute $sk_{u,\tau_u} = sk_0 - sk^*_0$. This is prevented during **Round 2**, through a consistency check step over the set $\mathcal{U}_{\text{BUFF}}$ [8, 29]. Since we assume that there are $k - t$ corrupted assistants, the server can obtain $k - t$ shares of sk_0 and sk^*_0, respectively. Furthermore, the server has the ability to convince $\frac{t}{2}$ honest assistants that the client u is in the buffer and the other $\frac{t}{2}$ honest assistants that u is not, thereby collecting shares of sk_0 and sk'_0 accordingly. Therefore, to ensure *Aggregator Obliviousness* regarding the JL scheme, we require that $k - t + \frac{t}{2} < t \implies t > \frac{2k}{3}$.

During the Verification phase, clients in the buffer set $\mathcal{U}_{\text{BUFF}}$ receive two to-be-verified elements from the server. The first element is the aggregated hash h_0 of the other clients in the buffer, verified using the individual (signed) commitments c_0. If the server sends an incorrect aggregated hash h_0 at this step, clients in $\mathcal{U}_{\text{BUFF}}$

can detect the error during the aggregated commitment check step thanks to the computationally binding property of the underlying commitment scheme. The second element is the aggregated model $\vec{x}$. The clients will not accept an incorrect global model if the hash h_{agg} of this value does not match the constructed hash h_0 sent by the server. This is ensured by the collision resistance of the homomorphic hash function H we use.

Further Extensions

Decentralized setup. To enhance readability, we initially present the protocol with the participation of a TD for the key generation of the JL and TEG schemes. It is worth to note that their Setup phase can be run among the assistants in a decentralized manner, i.e. without the intervention of a TD. Informally, the public LWE parameters can be generated following [15], the common modulus N of the JL scheme can be generated using the decentralized solution proposed in [12], and the TEG setup phase can follow the decentralized process presented in [29]. However, at the current stage, we are unaware of a threshold LWE encryption scheme that does not require a TD for its Setup phase.

Dynamic assistants. To balance the computation costs over assistants, it could be beneficial to offer dynamicity over the set $\mathcal{K}$, i.e. the set of assistants changes after some amount of aggregation operations. This could be achieved by assuming that regular clients pull down from the server information related to the set $\mathcal{K}$ of the selected assistants before sending the protected inputs. However, we want to minimize the communication overhead, hence we must avoid that clients require to know who are those assistants. Unfortunately, the current version of **Buffalo** implies exchanging pairwise keys between each pair of client and assistant. Consequently, each time the set $\mathcal{K}$ changes, new pairwise keys need to be generated.

Another way of enabling dynamic assistants would be to use the additively homomorphic TEG scheme (based on ECs) which allows all clients to encrypt their local inputs with the same public key, independently of the set $\mathcal{K}$. As a consequence, the set $\mathcal{K}$ of assistants can change using transfer share techniques as in [29]. Unfortunately, replacing the SS steps by such a TEG scheme would negatively affect the overall performance of the system. More precisely, the TEG decryption of the LWE and JL keys would consist of computing the discrete logarithm of a point in a large field. Thus, enabling realistic dynamic assistants in **Buffalo** while maintaining acceptable overheads is not straightforward.

6 COMPLEXITY ANALYSIS

We analyze the complexity of **Buffalo** and compare it with AsyncDPSecAgg, i.e. the asynchronous extension of DPSecAgg [43]. We report the comparison results in Table 2. We refrain from direct comparison with AsyncLightSecAgg, i.e. the asynchronous extension of LightSecAgg [42], due to its higher complexity in practical applications (see Section 2).

	AsyncDPSecAgg	**Buffalo**
Client Computation	Client: $O(k^2\frac{m}{\lfloor\delta_k k-t\rfloor}+d)$ Assistant: $O(n\frac{m}{\lfloor\delta_k k-t\rfloor})$	Client: $\boldsymbol{O(k^2+m+d)}$ Assistant: $\boldsymbol{O(n)}$
Client Communication	Client: $O(k\frac{m}{\lfloor\delta_k k-t\rfloor}+d)$ Assistant: $O(n\frac{m}{\lfloor\delta_k k-t\rfloor})$	Client: $\boldsymbol{O(k+m+d)}$ Assistant: $\boldsymbol{O(n)}$
Server Compuation	$O(k^2\frac{m}{\lfloor\delta_k k-t\rfloor}+nm+nd)$	$\boldsymbol{O(k^2+nm+nd)}$

Table 2: Complexity analysis for one BAsyncFL round (n: buffer size; k: number of assistants; m: number of LWE key coefficients, t: threshold value; d: input dimension; δ_k: fraction of dropped assistants).

• *Client Computation:* The encryption cost of the private input $\vec{x}_{u,\tau_u}$ in **Buffalo** is equivalent to the one in AsyncDPSecAgg. The encryption cost of the LWE secret key in AsyncDPSecAgg is calculated as $O(k^2\frac{m}{\lfloor\delta_k k-t\rfloor})$, due to the packed variant of the SS scheme. In contrast, in **Buffalo**, the cost of protecting this key through the JL scheme is $O(m)$. Furthermore, the cost for sharing the JL key using the SS scheme is $O(k^2)$ since such process deals with a scalar value. Indeed, our protocol does not secretly share a vector but a scalar value, and hence improves the overall performance, as empirically shown in Section 7. Regarding the assistants, the cost only corresponds to the summation of $O(n)$ shares in **Buffalo**, whereas it reaches $O(n\frac{m}{\lfloor\delta_k k-t\rfloor})$ in AsyncDPSecAgg.

• *Client Communication:* In both **Buffalo** and AsyncDPSecAgg, each client u sends a protected input $\langle\vec{x}_{u,\tau_u}\rangle$ of dimension $O(d)$. However, there are notable differences between the two solutions beyond this point. In AsyncDPSecAgg, clients transmit $O(k\frac{m}{\lfloor\delta_k k-t\rfloor})$ secret shares, and each assistant receives $O(n\frac{m}{\lfloor\delta_k k-t\rfloor})$ of them. In contrast, our protocol involves sending the LWE secret key, of size m, to the server. This is accompanied by the protected JL secret key. Each regular client thus ends up sending k shares of the JL secret key, while each assistant receives n of these shares. In practice, our experimental results have demonstrated that the bandwidth required for communication in AsyncDPSecAgg is greater compared to **Buffalo** at assistants, since the number of shares is proportional to the size m of the LWE secret key.

• *Server Computation:* At **Round 3**, the server constructs the JL aggregation key sk_0 from t shares, requiring a computation cost of $O(k^2)$, in contrast with $O(k^2\frac{m}{\lfloor\delta_k k-t\rfloor})$ for AsyncDPSecAgg due to the reconstruction of $\vec{s}_0$. Additionally, for both protocols, the server aggregates the protected LWE secret keys and the protected private inputs received from clients and unmasks the aggregated results, which requires computation costs of $O(n\cdot m)$ and $O(n\cdot d)$, respectively.

• *Server Communication:* The message exchanges in both protocols only occur between the server and clients. Hence, the server communication cost is equal to n times each client communication cost.

7 EXPERIMENTAL RESULTS

In this section, we experimentally evaluate the performance of **Buffalo**. In order to conduct a comparative study, in addition to our solution, we have also implemented AsyncDPSecAgg [43] (which is more efficient than AsyncLightSecAgg [34, 42]). Note that AsyncDPSecAgg is not secure in our considered threat model.

Experimental Setting

Our implementations use Python with the *Olympia* framework [34] and Pybind11 to wrap the C++ LWE SHELL library[‡]. The code for

[‡]https://github.com/google/shell-encryption/tree/master

this project is available on GitHub[§]. Experiments are conducted on a single-threaded processor, using a machine equipped with an Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz and 126 GB of RAM. For the sake of a fair comparison, **Buffalo** and AsyncDPSecAgg are implemented using the same building blocks and libraries mentioned in [10, 34]. We only evaluate the Online phase of the protocols. The Verification step implementation is the same as the one evaluated in [10]. Model parameter updates are converted to 8-bit fixed-point values by multiplying by a factor of 10.

• *FL Parameters:* To accurately evaluate the performance of the two schemes, we consider several scenarios that simulate realistic environments with the following varying parameters: buffer size n in {64, 128, 256, 512}; model dimension d in $\{30K, 260K, 1.2M\}$; assistant number k in {60, 120, 360} with a threshold t set as $\frac{2k}{3}$. The performance of the two solutions is evaluated by measuring the execution time (i.e. computation cost) and the bandwidth (i.e. communication cost) at both the client and server sides. The values shown for each scenario are the result of the average of measurements from five independent executions.

• *SA Parameters:* Following [4], the LWE error distribution χ_e with a discrete Gaussian distribution has a standard deviation equal to 4.5. The degree of the ring is set to $m = 2^{11}$ and the prime modulus $q = 1(\mod N)$, with $\lambda > 128$ bits of security and a buffer size $n \leq 10^4$ [3]. The size of the JL modulus N is set to $\log_2(N) = 2048$. For both the LWE and JL schemes, we apply packing techniques from [4, 30] to pack multiple plaintexts in one single ciphertext.

• *BAsyncFL Environment:* We emulate realistic BAsyncFL scenarios using the FLSim framework[¶] with Pytorch. The FLSim framework allows us to emulate asynchronicity. We use the same staleness distribution (i.e. the delay in clients' updates that can occur in asynchronous systems) as FedBuff [35], namely a half-normal distribution with standard deviation equal to 1.25.

• *Test Datasets:* We consider three FL use cases with two datasets from the LEAF benchmark [11], namely CELEBA and SENT140, and one new medical dataset called REPLACE-BG [16]. Stochastic Gradient Descent (SGD) was employed as the client training algorithm with a learning rate η, batch size b and a specific number e of local SGD steps.

– The CELEBA is a binary image classification dataset that contains celebrity pictures. We use the Convolutional Neural Network (CNN) proposed in [47]. We set $\eta = 0.01$, $b = 8$, $e = 10$.

– The SENT140 dataset is a text classification dataset for binary sentiment analysis. We use an LSTM model with 1.2M parameters. We set $\eta = 0.1$, $b = 32$, $e = 10$.

– The REPLACE-BG dataset was obtained from a cohort of 202 adult participants. It consists of three primary features: *(i)* interstitial glucose levels measured in milligrams per deciliter (mg/dL) using Dexcom G4 Platinum sensors; *(ii)* insulin boluses administered in units (U); *(iii)* carbohydrate (CHO) content measured in milligrams (mg). We have implemented the pre-processing steps for this dataset following the procedure described in [22]. Namely, the data were prepared as inputs to a CNN-Long Short-Term Memory (CNN-LSTM) architecture, a model commonly used for sequential data prediction tasks. The CNN-LSTM network is trained to predict blood glucose levels for the subsequent hour based on data from the last three hours, including glucose levels, insulin boluses, and CHO content [14]. We set $\eta = 0.1$, $b = 64$, $e = 20$.

[§]https://github.com/rtaiello/buffalo

[¶]https://github.com/facebookresearch/FLSim/tree/main

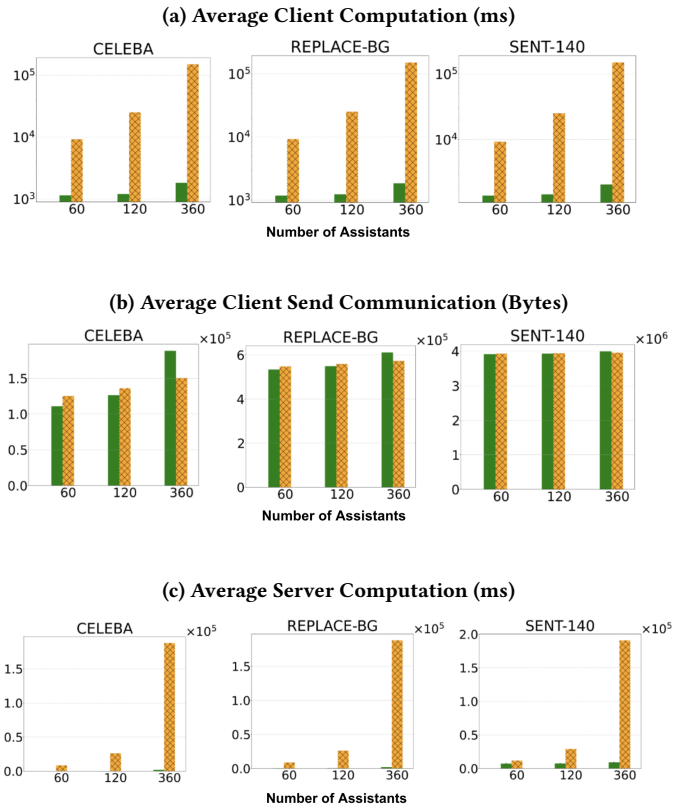

Figure 4: Performance evaluation of Buffalo (green) and AsyncDPSecAgg (crossed-orange). Buffer size is fixed to $n = 512$ while varying the number $k \in \{60, 120, 360\}$ of assistants.

Client and Server Performance

We first evaluate the performance of **Buffalo** and compare it with AsyncDPSecAgg [43]. We consider the three aforementioned use cases and fix the buffer size to $n = 512$. We first measure the computation and communication costs at the client with respect to the number k of assistants. Figure 4 shows the experimental results. We observe that, thanks to the use of the JL scheme for the aggregation of the LWE key, **Buffalo** outperforms AsyncDPSecAgg in terms of computation both at the client and server sides. We recall that the LWE key is secretly shared with assistants in AsyncDPSecAgg while this is the JL key in **Buffalo**. The communication cost remains similar for both solutions since this cost mainly depends on the transmission of protected inputs of dimension d. Below, we will report the performance comparison at the assistant.

Assistant Performance

We report the performance costs for the Online phase at assistants in Table 3. In **Buffalo**, assistants firstly perform homomorphic operations, and participate in reconstructing a single message. Those steps provide minimal overhead, namely < 100 ms in CPU time and < 0.2 MB in bandwidth. In [29], committee members take part of the reconstruction of multiple messages, incurring significant overhead worth 560 ms CPU time and > 1 MB in bandwidth (see Figures 5, 6 and 7 in [29]). In addition, **Buffalo** only requires 512 assistants at most. In [27], there is a huge number of contributing clients ($n > 20k$) and of committee members ($k = 2^{14}$) to reach the desired security level, which impedes the overall performance of the protocol.

Buffer Size	Computation (ms)	Communication (MB)
64	0.31	0.02
128	0.63	0.03
256	1.26	0.07
512	62.54	0.13

Table 3: Assistant performance costs. The buffer size n varies in $\{64, 128, 256, 512\}$.

Overall Performance

For each FL task, namely REPLACE-BG, CELEBA and SENT140, we consider Client Updates (CU), determined by the product of the buffer size and the total number of rounds necessary to achieve the Target Metric (TM). We set the number of assistants as $k = 60$, with a dropout rate $\delta_k = 0.01$.

In Table 4, we depict the total execution time in hours (h) and the bandwidth consumption in GigaBytes (GB) of **Buffalo** and AsyncDPSecAgg, and compare them with the case where aggregation is performed in cleartext (called FedBuff). We observe that **Buffalo** always outperforms AsyncDPSecAgg by at least a factor of $\times 1.4$ in terms of computation, including the overhead induced by the verification step for aggregation integrity, which is not present in AsyncDPSecAgg.

Protocol	Time (h)	Bandwidth (GB)	Avg ($= \lfloor \rho \cdot d \rfloor$)	Storage (MB)
REPLACE-BG ($d = 260K$; CU=$24K$; TM: RMSE 11.5% ↓; $n = 16$; $n_{tot} = 180$)				
FedBuff	28.41	6.42	N/A	N/A
AsyncDPSecAgg	100.46	16.37	N/A	N/A
Buffalo	73.43	14.63	1653	0.26
CELEBA ($d = 31K$; CU=$48K$; TM: Accuracy 90.0% ↑; $n = 128$; $n_{tot} = 2336$)				
FedBuff	6.75	1.51	N/A	N/A
AsyncDPSecAgg	142.10	13.20	N/A	N/A
Buffalo	54.61	10.04	2310	0.03
SENT140 ($d = 1.2M$; CU=$99K$; TM: Accuracy 80.0% ↑, $n = 256$; $n_{tot} = 3482$)				
FedBuff	6.55	92.26	N/A	N/A
AsyncDPSecAgg	$> 7d.$	395.93	N/A	N/A
Buffalo	77.98	390.01	1786	1.20

Table 4: Costs of execution time and bandwidth consumption, under various FL tasks. ↑ means that higher is better and ↓ lower is better.

We also evaluate the per-buffer average number of updated model parameters, which corresponds to Avg $= \lfloor \rho \cdot d \rfloor$, and the input storage cost in MegaBytes (MB). **Buffalo** consistently shows savings, especially when applied to the SENT140 task and its underlying ML model. On average, 1786 elements are considered in the model hashing instead of $1.2M$. Such a difference is primarily due to the underlying embedding layers since the latter are designed to process high-dimensional, typically sparse, data [18].

Figure 5 shows the computation costs of the two hash functions H (using Eq. 1) and H_{INC} (using Eq. 2). In Figure 5 (a), the number of assistants is equal to $k = 60$, while varying the input sparsity parameter ρ of the model in $\{0.01, 0.1, 0.5\}$. In Figure 5 (b), for each of the three datasets, we use a specific average input sparsity Avg $= \lfloor \rho \cdot d \rfloor$, as detailed in Table 4. In Figure 5 (a), we vary the input sparsity parameter ρ while keeping the input size fixed at $d = 10^5$. We see that using H_{INC} enhances the computational performance compared to employing H. In Figure 5 (b), we directly use the average sparsity parameter specific to each task along with their respective inputs. We notice a clear performance improvement of at least $4\times$ by using H_{INC} compared to hashing with H.

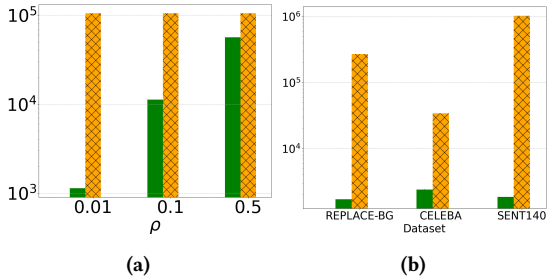

Figure 5: Computation costs of the hash functions H_{INC} w.r.t. Eq. 2 (green) and H w.r.t. Eq. 1 (crossed-orange). The input dimension is fixed to $d = 10^5$ and the buffer size to $n = 2^6$.

8 CONCLUSION

We introduced **Buffalo**, a SA protocol for BAsyncFL. To achieve asynchronicity, **Buffalo** uses lattice-based techniques for input protection and the participation of assistants to help the server construct on-the-fly secret aggregation keys. **Buffalo** also offers verification mechanisms to ensure aggregation integrity. Our evaluation, through theoretical analysis and implementation based on real-world datasets, shows the efficiency and practicality of our solution. Future work will extend our threat model to include malicious clients attempting to poison the global model, by considering robust client input validation.

ACKNOWLEDGMENTS

We thank the CODASPY reviewers for their comments, and D. Pasquini and H. Sreedhar for their help. This work has been supported by the French government, through the ANR (Agence Nationale de la Recherche) projects ANR-23-IACL-0001, ANR-22-FAI1-0003-02, ANR-23-CPJ1-0060-01, ANR-19-CE45-0006-01, and ANR-23-CPJ1-0060-01.

REFERENCES

[1] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, Janno Siim, and Michal Zajkac. 2019. DL-Extractable UC-Commitment Schemes. In *Applied Cryptography and Network Security*. 385–405.

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

[3] Martin R Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology* 9, 3 (2015), 169–203.

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

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

[6] James Bell-Clark, Adrià Gascón, Baiyu Li, Mariana Raykova, and Phillipp Schoppmann. 2024. Willow: Secure Aggregation with One-Shot Clients. *Cryptology ePrint Archive* (2024).

[7] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. 1994. Incremental cryptography: The case of hashing and signing. In *14th Annual International Cryptology Conference (CRYPTO)*. Springer, 216–233.

[8] 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 *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1175–1191.

[9] Kallista A. Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, H. Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. 2019. Towards Federated Learning at Scale: System Design. *CoRR* abs/1902.01046 (2019).

[10] Baturalp Buyukates, Jinhyun So, Hessam Mahdavifar, and Salman Avestimehr. 2022. LightVeriFL: Lightweight and Verifiable Secure Federated Learning. In *Workshop on Federated Learning: Recent Advances and New Challenges*.

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

[12] Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, Abhi Shelat, Muthu Venkitasubramaniam, and Ruihan Wang. 2021. Diogenes: lightweight scalable RSA modulus generation with a dishonest majority. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 590–607.

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

[14] Ran Cui, Chirath Hettiarachchi, Christopher J Nolan, Elena Daskalaki, and Hanna Suominen. 2021. Personalised short-term glucose prediction via recurrent self-attention network. In *2021 IEEE 34th International Symposium on Computer-Based Medical Systems (CBMS)*. IEEE, 154–159.

[15] Thomas Espitau, Guilhem Niot, and Thomas Prest. 2024. Flood and submerse: Distributed key generation and robust threshold signature from lattices. In *Annual International Cryptology Conference*. Springer, 425–458.

[16] Grazia Aleppo et al. 2017. REPLACE-BG: a randomized trial comparing continuous glucose monitoring with and without routine blood glucose monitoring in adults with well-controlled type 1 diabetes. *Diabetes care* 40, 4 (2017), 538–545.

[17] Peter Kairouz et al. 2019. Advances and Open Problems in Federated Learning. *CoRR* abs/1912.04977 (2019).

[18] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *2021 ACM SIGCOMM 2021 Conference*. 676–691.

[19] Xiaojie Guo. 2022. Fixing Issues and Achieving Maliciously Secure Verifiable Aggregation in"VeriFL: Communication-Efficient and Fast Verifiable Aggregation for Federated Learning". *Cryptology ePrint Archive* (2022).

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

[21] Andrew Hard, Antonious M Girgis, Ehsan Amid, Sean Augenstein, Lara McConnaughey, Rajiv Mathews, and Rohan Anil. 2024. Learning from straggler clients in federated learning. *arXiv preprint arXiv:2403.09086* (2024).

[22] Mehrad Jaloli and Marzia Cescon. 2023. Long-term prediction of blood glucose levels in type 1 diabetes using a cnn-lstm-based deep neural network. *Journal of diabetes science and technology* 17, 6 (2023), 1590–1601.

[23] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The Elliptic Curve Digital Signature Algorithm (ECDSA). *Int. J. Inf. Secur.* 1, 1 (Aug. 2001), 36–63.

[24] Marc Joye and Benoît Libert. 2013. A Scalable Scheme for Privacy-Preserving Aggregation of Time-Series Data. In *Financial Cryptography and Data Security*, Ahmad-Reza Sadeghi (Ed.). Springer Berlin Heidelberg.

[25] Harish Karthikeyan and Antigoni Polychroniadou. 2024. OPA One-shot Private Aggregation with Single Client Interaction and its Applications to Federated Learning. *Cryptology ePrint Archive* (2024).

[26] Joohee Lee, Dongwoo Kim, Duhyeong Kim, Yongsoo Song, Junbum Shin, and Jung Hee Cheon. 2018. Instant privacy-preserving biometric authentication for hamming distance. *Cryptology ePrint Archive* (2018).

[27] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. 2023. LERNA: Secure Single-Server Aggregation via Key-Homomorphic Masking. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 302–334.

[28] Julio Lopez and Ricardo Dahab. 2000. An overview of elliptic curve cryptography. *Institute of Computing, Sate University of Campinas, Sao Paulo, Brazil, Tech. Rep* (2000).

[29] 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. *Cryptology ePrint Archive, Paper 2023/486* (2023).

[30] Mohamad Mansouri, Melek Önen, and Wafa Ben Jaballah. 2022. Learning from Failures: Secure and Fault-Tolerant Aggregation for Federated Learning. In *38th Annual Computer Security Applications Conference*. 146–158.

[31] Mohamad Mansouri, Melek Onen, Wafa Ben Jaballah, and Mauro Conti. 2023. Sok: Secure aggregation based on cryptographic schemes for federated learning. *Proc. Priv. Enhancing Technol* 1 (2023), 140–157.

[32] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *20th International Conference on Artificial Intelligence and Statistics*. PMLR.

[33] Milad Nasr, Reza Shokri, and Amir Houmansadr. 2019. Comprehensive Privacy Analysis of Deep Learning: Passive and Active White-box Inference Attacks against Centralized and Federated Learning. In *2019 IEEE Symposium on Security and Privacy (SP)*.

[34] Ivoline C Ngong, Nicholas Gibson, and Joseph P Near. 2023. OLYMPIA: A Simulation Framework for Evaluating the Concrete Scalability of Secure Aggregation Protocols. *arXiv preprint arXiv:2302.10084* (2023).

[35] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. 2022. Federated learning with buffered asynchronous aggregation. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 3581–3607.

[36] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology — EUROCRYPT '99*. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

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

[39] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[40] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* (1979).

[41] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy (SP)*.

[42] Jinhyun So, Chaoyang He, 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. *Proceedings of Machine Learning and Systems* 4 (2022), 694–720.

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

[44] Nikko Ström. 2015. Scalable distributed DNN training using commodity GPU cloud computing. (2015).

[45] Jun Sun, Tianyi Chen, Georgios B Giannakis, Qinmin Yang, and Zaiyue Yang. 2020. Lazily aggregated quantized gradient innovation for communication-efficient federated learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44, 4 (2020), 2031–2044.

[46] Elina van Kempen, Qifei Li, Giorgia Azzurra Marson, and Claudio Soriente. 2023. LISA: LIghtweight single-server Secure Aggregation with a public source of randomness. *arXiv preprint arXiv:2308.02208* (2023).

[47] Cong Xie, Sanmi Koyejo, and Indranil Gupta. 2019. Asynchronous federated optimization. *arXiv preprint arXiv:1903.03934* (2019).

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