Federated Learning Secure Aggregation Protocol Implementation

Vincent Yuan

Supervisors: Lie He, Ignacio Saleman, Mary-Anne Hartley, Martin Jaggi

***Abstract*—EPFL Fall 2021 Semester project report of Yuan Vincent. This project has been done at the Machine Learning and Optimization (MLO) laboratory, under the supervision of Lie He, Ignacio Saleman, Mary-Anne Hartley and Martin Jaggi.**

Federated learning (FL) allows Machine Learning models to be trained on data spread across several clients while preserving clients' data privacy. Bonawitz et al. designed a FL aggregation scheme that securely computes sums of vector without disclosing them to the central server.

There was not any open-sourced implementation of this scheme in JavaScript to our knowledge extent. We successfully implemented the scheme in JavaScript during the project, solving numerous challenges and making implementation choices.

DeAI is a open-source project enabling collaborative learning. We plan on leveraging the gained insights to implement the scheme in the DeAI platform, enhancing its security guarantees.

CONTENTS

I. INTRODUCTION

Privacy in Data Science and Machine Learning (ML) is a sensitive topic. Data-related scandals like Cambridge Analytica [1] raised the concern of protecting privacy in a world fond of data. Machine learning models can cost millions to train [2] and are vulnerable to theft.

Privacy-Preserving Machine Learning (PPML) techniques aim at avoiding or mitigating data breaches in ML projects. We focused on this topic during the project, and especially on the implementation of the Secure Aggregation Protocol from Bonawitz et al. [3].

The Secure Aggregation Protocol from Bonawitz et al. securely aggregates vectors from different clients in a Federated Learning (FL) setting. It is robust to dropouts and requires a low communication overhead. To our knowledge extent, open-sourced implementations of the protocol in JavaScript don't exist. We implemented the protocol in JavaScript which led us to make practical implementation choices.

DeAI [4] is an open-source project enabling collaborative and privacy-preserving training of ML models. We discovered and apprehended the DeAI [4] platform codebase during the semester.

We combined the insights gained by implementing the model with our understanding of the DeAI platform acquired during the semester to prepare the implementation of the Secure Aggregation scheme in DeAI. This implementation will add even more security guarantees for DeAI users.

After presenting the current PPML state of research, we will introduce the Secure Aggregation scheme and dive into our implementation. Then, we will develop on how to implement the scheme in DeAI. We will finish by explaining the limits of the scheme and the related future work.

II. BACKGROUND

A. Machine Learning and privacy

Machine Learning (ML) [5] leverages data and algorithms to imitate how humans learn. It is a widely spread technique applied for numerous tasks, from movie recommendations to human genome analysis [6].

Neural networks models are commonly used for ML tasks. They are composed of high dimensional vectors which output the desired value given an input. Training a neural network

is usually done by updating the model using a mini-batch stochastic gradient descent rule [7].

Neural networks ingest data to create their model and their result relies heavily on data quality and quantity. Data has become a crucial issue, and recent scandals have increased the public's awareness towards the use of their personal data. Indeed, data can contain confidential information that people are not willing to disclose. For example, medical data [8] can reveal a predisposition to future diseases.

B. Federated Learning

Data sharing, especially the one containing confidential information, has become a major challenge for ML Model improvements. It is an example of the privacy/utility tradeoff: a higher amount of data enabled by data sharing will lead to a better model, but at the cost of data privacy. The lack of data sharing among organizations might come from institutional regulations such as the GDPR. It might also come from organizations willing to keep their data for themselves while sharing it is necessary for the common good. For example, researchers around the world have quickly reacted to the Ebola outbreak, to then face a lack of data. The organizations owning the data were not willing to share it which slowed down the fight against the virus [9], [10].

For several years, ML research has been focusing on a new scheme: collaborative learning. In this scheme, multiple clients with similar data train locally a model. Their model is then exchanged among them and aggregated to create an improved model based on several clients, thus on more data. This protocol allows every clients to get an improved model by sharing only their model and not their data. This scheme is gaining popularity for multiple reasons, one of the major being that it is supposed to preserve data privacy. As an example, Google uses Federated Learning (FL) to enhance their Android Keyboard [11], leveraging data from millions of users.

When deciding to use collaborative learning, we must make choices about the architecture. Two main branches exists: Federated Learning relies on the existence of one or more servers that will receive, aggregate, and broadcast the clients' model, while Decentralized Learning is based on clients directly exchanging their model between themselves without a central server.

The explosive growth of FL research in less than a decade [12], [13] has raised new problems related to this task, such as improving efficiency or addressing inter-operability challenges [14]–[16]. It appeared that Federated Learning also faced privacy challenges.

C. Privacy in Federated learning

Even though clients never transmit plain data to other parties, preserving data privacy is still an important challenge in FL. Ongoing research has shown various attacks on clients' privacy.

Given a data record and a black-box access to a model, membership inference attacks determine if the record was used to train the model [17]. This attack has been extended to the FL scheme - an adversarial client can infer the presence of exact data points in others' training data [18].

Another attack allows the adversary to train a Generative Adversarial Network (GAN) [19] to infer the training set distribution and to generate prototypical samples of the training set, reducing the privacy benefits of FL [20].

A malicious central server can focus on one client and stores its model differences over the iterations. From these differences, the central server can infer the gradient and recover the client's private data [21].

One Privacy-Preserving Machine Learning (PPML) technique aiming at mitigating attacks is Differential Privacy (DP) [22]. It bounds the extent of data leakage from a model, ensuring that the output is similar whether any single individual's record is included in the dataset or not. It can be applied in FL from the clients' perspective to hide their contributions during training [23], [24]. However, there is a utility/privacy tradeoff when using DP, and some attacks are effective despite DP's use [20].

Another PPML technique that can be applied to FL is Secure Multiparty Computation (SMC). It relies on sharing private data or models to non-colluding servers that will securely aggregate them into a joint model. By limiting the number of parties among which the trust is split and assuming an honest majority among them, a model can be safely collaboratively created. However, SMC protocols exchanging data rely on Yao's garbled circuit which is not easily scalable [25]–[28]. Other protocols ensuring SMC rely on Shamir Secret Sharing or Homomorphic encryption - the scheme we will implement performs SMC using Shamir Secret Sharing. These cryptographic techniques come with a computational cost impacting the protocol efficiency.

D. Topology and threat model

Just as there are different possibilities for collaborative learning settings, there are different threat models in privacy.

The honest-but-curious (HBC) adversary is a legitimate participant in a communication protocol who will not deviate from the defined protocol but will attempt to learn all possible information from legitimately received messages [29]. In constrast, the malicious adversary can deviate from the defined protocol arbitrarily.

In a Federated Learning setting, we can consider as adversaries the server, a given number of clients ranging from 1 to $N-1$ (if we assume there are N clients in the scheme), or the two combined.

III. A privacy-preserving Secure Aggregation scheme

Bonawitz et al. [3] propose a practical Secure Multiparty Computation protocol for Federated learning, where one server securely aggregates n clients' model. The aggregation is the sum of the clients' model, which can be combined with the Federated Average (FedAvg) protocol [30].

We aim at providing to the readers a comprehension of the scheme. Nonetheless, we strongly recommend readers to

read the original paper [3] if their goal is to get a complete understanding of the protocol and its properties as we will not provide exhaustive explanations.

A. Protocol properties

Dropout handling: The aggregation scheme handles dropouts, i.e. clients dropping during the protocol. A threshold t indicates the minimal number of clients needed to finish the aggregation properly.

Security guarantees: The protocol provides security guarantees in the case of HBC adversaries. These guarantees can also hold for malicious adversaries if we assume the existence of a public-key infrastructure (which leads to slight modifications of the protocol).

- If only a joint set of clients is adversarial, the clients jointly learn nothing more than their own inputs.
- If only the server is adversarial, it can not learn any other information than the aggregation of the clients' model, assuming $t > \lceil \frac{n}{2} \rceil + 1$.
- If the server and n_C clients are adversarial, they can only learn the sum of the other clients' input, assuming that $n_C < \lfloor \frac{n}{3} \rfloor$.

Low cost: The protocol's computation, communication and storage cost are low considering the model is dropout-resistant. Assuming n clients and a data vector of size m, the client-side computation cost is $\mathcal{O}(n^2 + mn)$, its communication cost is $\mathcal{O}(n + m)$, and its storage cost is $\mathcal{O}(n + m)$. Server-side, the computation cost is $\mathcal{O}(mn^2)$, the communication cost is $\mathcal{O}(n^2 + mn)$, and the storage cost is $\mathcal{O}(n^2 + m)$. Details and breakdown of these performance analyses can be found in the original paper [3].

B. Cryptographic primitives

In this subsection, we will promptly review the cryptographic primitives used in the scheme. Exhaustive explanations can be found in the original paper [3].

1) **Shamir Secret Sharing**: Shamir's t-out-of-n Secret Sharing [31] splits a secret s among n shares. Any subset of t shares can reconstruct s, but any subset with less than t shares gives no information about s.
2) **Key and Number Agreements:** Asymmetric cryptography relies on public-private key pairs (PK, SK). Given two users u and v with public-private keys (s_u^{PK}, s_u^{SK}) and (s_v^{PK}, s_v^{SK}), the key agreement and the number agreement respectively generate the same private shared key $s_{u,v}$ or the same number from the key pairs (s_u^{SK}, s_v^{PK}) and (s_v^{SK}, s_u^{PK})
3) **Authenticated Encryption:** Authenticated encryption is a form of symmetric encryption which simultaneously assure the confidentiality and authenticity of data. It relies on a shared secret key used to encrypt and decrypt messages.
4) **PRG:** A pseudorandom generator [32] **PRG** generates a random mask in a given interval. It can be seeded for results reproducibility. The PRG must output any possible value with the same probability.

C. Protocol intuition

In this subsection, we will give to the readers a technical intuition about the protocol.

We assume an ordered set of users U, with every user $u \in U$ holding a private vector x_u representing the model parameters. We assume that the elements of x_u and $\sum_{u \in U} x_u$ are in $\mathbb{Z}_R$ for some large prime R.

For each pair of users (u,v), $u < v$, we suppose they agree on a shared mask $d_{u,v}$ of the same size . If u adds $d_{u,v}$ to its model vector x_u and v subtracts it from its model vector x_v, the server will receive masked inputs and will not able to infer the original values. When summing the inputs, the mask will be canceled and the aggregated result will be the sum of every users vector.

Mathematically, each user u computes:

$$y_u = x_u + \sum_{v \in U : u < v} d_{u,v} - \sum_{v \in U : u > v} d_{u,v} \; (mod\, R)$$

and sends it to the server, which computes:

$$\begin{aligned} z &= \sum_{u \in U} y_u \\ &= \sum_{u \in U} \Big(x_u + \sum_{v \in U : u < v} d_{u,v} - \sum_{v \in U : u > v} d_{u,v} \Big) \\ &= \sum_{u \in U} x_u \; (mod\, R) \end{aligned}$$

However, the current protocol is not dropout tolerant. If a user u drops, each user $v \in U \setminus \{u\}$ will not have its mask $d_{u,v}$ canceled. A solution would be to notify the dropout to the remaining users and ask for their shared mask with user u. With this solution, we have 2 problems:

1) When recovering the masks, other users might drop out. The server must require an additional recovery phase for the newly dropped users, and so on.
2) Targeting an user u, a malicious server can ask to all the other users $v \in U \setminus \{u\}$ their pairwise masks with u. The server can then reconstruct u's private vector.

The protocol solves these problems by using conjointly:

1) A pseudorandom generator (PRG) allowing clients u and v to agree on a common seed $s_{u,v}$ rather than an entire mask $d_{u,v}$. These shared seeds will be computed by a number agreement from u and v public-private key pairs.
2) A secret sharing scheme allowing pairwise seeds to be recovered even if additional parties drop out during the recovery phase, as long as a minimum number of clients (equal to the threshold t) remains alive.
3) A double-masking structure: Each user u samples an additional random seed b_u to generate a self-mask.

We can combine the different techniques to get a simplified version of the protocol:

1) **Advertise Keys**: Each user u generates its public-private key pair (s_u^{PK}, s_u^{SK}). It sends the public key to the server,

which broadcasts it to the other users. Then, u receives the public keys of the other users.

2) **Share Keys**: Each user u samples a random seed b_u and generate the $s_{u,v}$ values. Each $s_{u,v}$ is generated from s_u^{SK} and s_v^{PK} number agreement. User u computes t-out-of-n secret shares for b_u and s_u^{SK} and share them with the other users. Then, u receives the other users' shares.
3) **Masked Input Collection**: Each user u sends to the server:

$$y_u = x_u + PRG(b_u) + \sum_{v \in U : u < v} PRG(s_{u,v}) - \sum_{v \in U : u > v} PRG(s_{u,v}) \; (mod\, R)$$

4) **Unmasking**: The server must make an explicit choice with respect to each user u: from each surviving member v, the server can request either a share of the secret key s_u^{SK}, or a share of b_u. After collecting t shares of s_u^{SK} for all dropped users and t shares of b_u for all remaining users, the server can cancel the masks and retrieve the sum.

The unmasking round is the reason why we have to limit $t \geq \lceil n/2 \rceil + 1$ in the case of an adversarial server and $t \geq \lceil 2n/3 \rceil + 1$ in the case of a clients-server collusion.

We note that this simplified protocol sends the secret shares in plaintext to the server. In the exhaustive protocol, each pair of users (u, v) encrypts their secret shares with a symmetric key. This symmetric key is computed from a key agreement. The keys used for encryption must be different from the keys used to generate the shared seeds.

D. Protocol description and example

In this subsection, we will present the protocol by describing it more exhaustively and illustrating it in an example.

In the description as well as the example, we assume a HBC adversary. Providing security against malicious adversaries requires a public key infrastructure signing the public keys and an extra consistency check round. Despite these slight changes, the protocol is significantly the same.

In the example, we assume 5 clients, sorted by alphabetical order: Alice, Bob, Charlie, Daniel, and Eve. Eve, Daniel, and Charlie will respectively drop at rounds 0, 1, and 2.

We will describe the general protocol in black while illustrating it with an example in blue. We use schemes to visualize the different steps of the example. The separation between clients operations, server operations and client-server communication are shown in the schemes.

For simplicity, we will only focus on Bob to show the clients' operations, and on the Bob-Alice pair to show the pairwise computations. However, all clients perform the same operations as Bob, and all pairs perform the same computation as the pair Bob-Alice.

Setup:

We assume an initial number of user n and a threshold value t, both shared among the clients and the server. We also assume an ordering between client IDs and a private authenticated channel between the clients and the server.

The initial number of users is n=5, and the threshold value is t=2. We note that t and n do not satisfy the relation security guarantee relation. This does not affect our example. We assume an alphabetical ordering.

Round 0 (AdvertiseKeys):

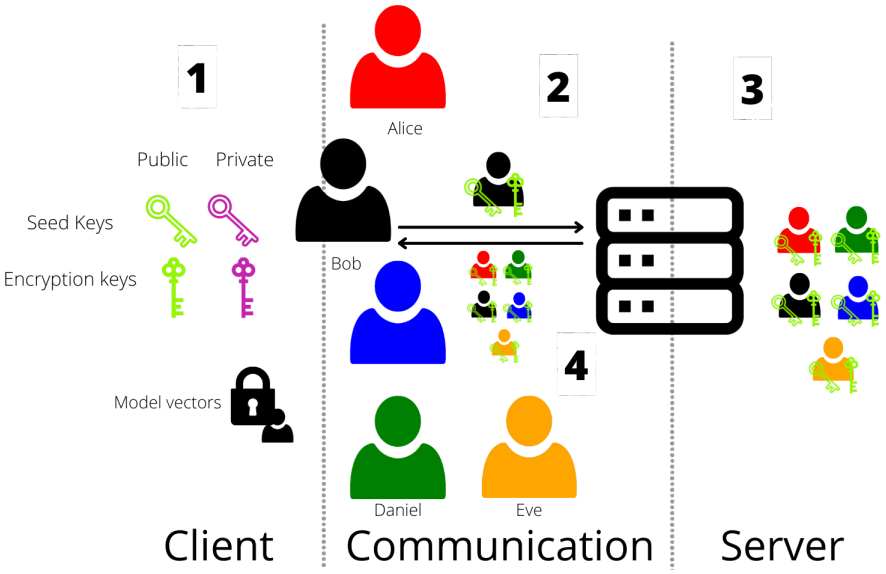

Figure 1. Round 0: Advertise Keys

User u:

1) Generate two key pairs: one for the encryption (c_u^{PK}, c_u^{SK}) and one for the generation of the shared seeds (s_u^{PK}, s_u^{SK})
   Bob generates the two pairs of keys. Please note the shape and color of the keys.
2) Send public keys (c_u^{PK}, s_u^{PK}) to the server and move to the next round.

Server:

3) Collect at least t messages from individual users, and denote with U_1 this set of users.
4) Broadcast to all users U_1 the list $\{(v, c_v^{PK}, s_v^{PK})\}_{v \in U_1}$ and move to the next round.

Round 1 (ShareKeys):

In this round, Eve drops before sending its ciphertext to the server (e.g. before step 6).

User u:

1) Receive the list broadcasted by the server. Assert that $|U_1| \geq t$, that all the public key pairs are different.
   Bob receives 5 clients for $|U_1|$, which is greater than the threshold t=2.
2) Sample a random element b_u (to be used as seed for a PRG).
3) Generate t-out-of-$|U1|$ shares of s_u^{SK}: $\{(v, s_{u,v}^{SK})\}_{v \in U_1}$ and b_u: $\{(v, b_{u,v}^{SK})\}_{v \in U_1}$
   Bob creates 2-out-of-5 shares of its shared seed generation secret key and its random element.
4) For all other clients v, generate an Authenticated Encryption symmetric key by combining u's encryption secret key with v's encryption public key. The user v will generate the same key with its encryption secret key combined with u's encryption public key.
   Bob generates the symmetric encryption key with Alice by combining its encryption secret key with Alice's encryption public key.
5) For each other $v \in U_1 \setminus \{u\}$, compute $e_{u,v}$ by encrypting the message $(u||v||s_{u,v}^{SK}||b_{u,v})$ with the key previously generated with v.
   Bob generates the ciphertext to share with Alice. He puts in the metadata himself as the sender and Alice as the receiver.
6) Send all the ciphertexts $e_{u,v}$ to the server. Store values generated in this round, and move to the next round.

Server:

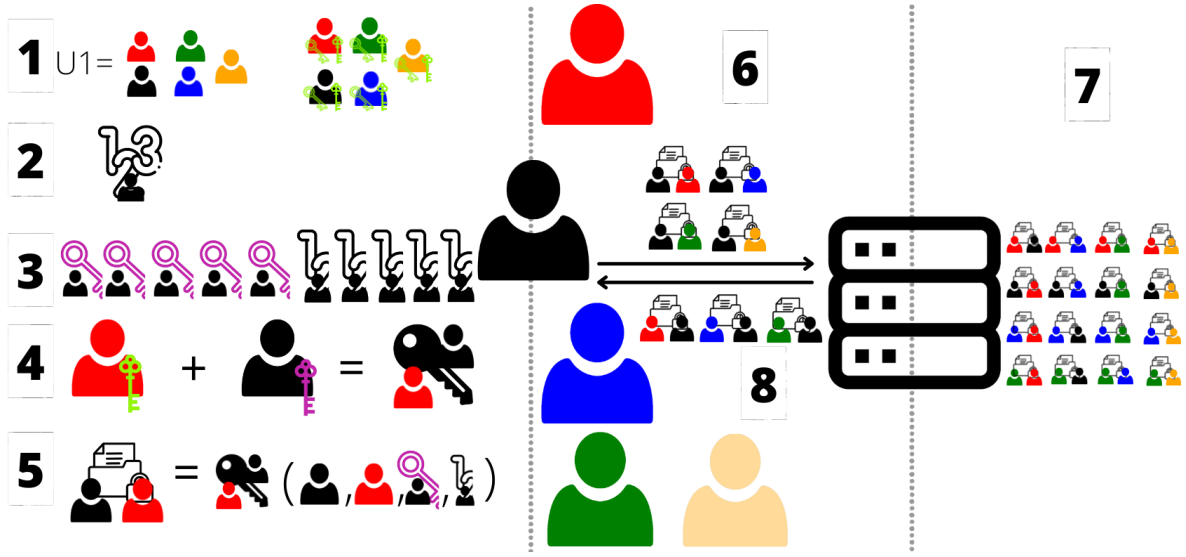

Figure 2. Round 1: Share Keys

7) Collect a list of ciphertexts from at least t users (denote with $U_2 \subseteq U_1$ this set of users).
8) Send to each user $u \in U_2$ all ciphertexts encrypted for it: $\{e_{v,u}\}_{v \in U_2}$ and move to the next round.
Eve has dropped from the protocol, thus the server only sends ciphertexts from Alice, Charlie, and Daniel to Bob.

Round 2 (MaskedInputCollection):

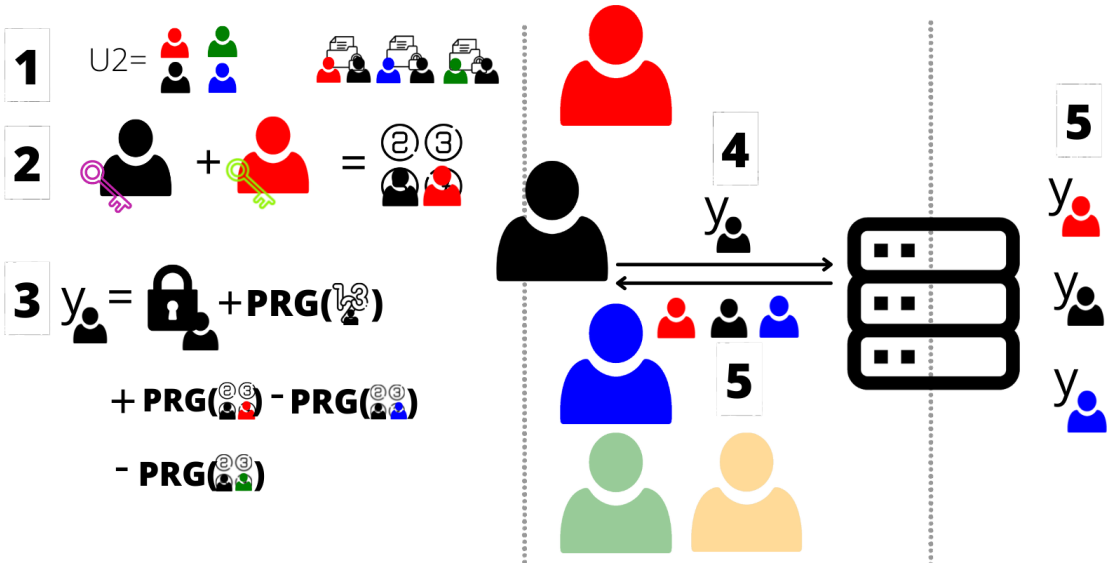

Figure 3. Round 2: Masked Input Collection

In this round, Daniel drops before sending its masked gradient to the server (e.g. before step 4).

User u:

1) Receive from the server the list of ciphertexts $\{e_{v,u}\}_{v \in U_2}$ (and infer the set U_2). If the list is of size $<$ t, abort.
Bob infers that there are still 4 clients remaining, so he continues the protocol.
2) For each other $v \in U_2 \setminus \{u\}$, compute $s_{u,v}$ by combining s_u^{SK} and s_v^{PK}, expand this value using a PRG into a random vector $p_{u,v} = \Delta_{u,v} \cdot PRG(s_{u,v})$ where $\Delta_{u,v} = 1$ when $u > v$ and $\Delta_{u,v} = -1$ when $u < v$. Additionally, define $p_{u,u} = 0$.
Bob computes its shared seed with Alice $s_{Alice,Bob}$ by combining its seed secret key with Alice's seed public

key.

3) Compute the user's own private mask vector $p_u = PRG(b_u)$. Then, compute the masked input vector $y_u = x_u + p_u + \sum_{v \in U_2} p_{u,v} \ (mod\, R)$.
Bob adds its own private mask vector to the secret. Alice is smaller than Bob, so he adds the value generated from the Alice-Bob seeded PRG to the masked gradient. Bob is smaller than Charlie and Daniel, so he subtracts the value generated by the Bob-Charlie seeded PRG and the Bob-Daniel seeded PRG to the masked gradient.
4) Send y_u to the server and move to the next round.

Server:

5) Collect 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.
Daniel has dropped in the meantime and is not in the set U_3.

Round 3 (Unmasking):

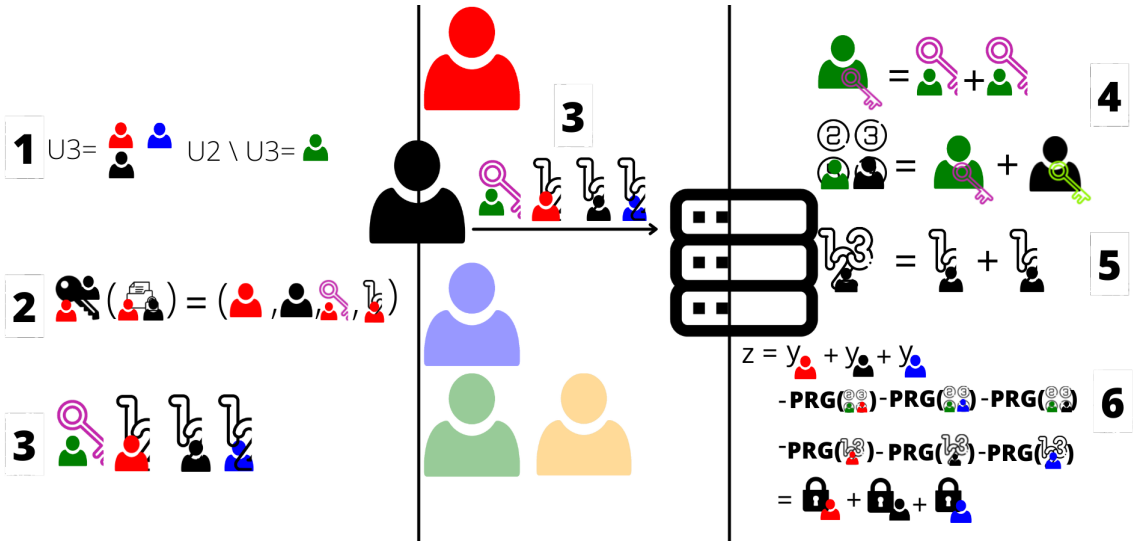

Figure 4. Round 3: Unmasking

In this round, Charlie drops before sending the shares to the server (e.g. before step 3).

User u:

1) Receive from the server U_3. Verify that $U_3 \subseteq U_2$, that $|U_3| \geq t$.
With Daniel drop out, there are still 3 clients remaining. It is greater than the threshold of 2.
2) For each other user v in $U_2 \setminus \{u\}$, decrypt the ciphertext $e_{v,u}$ received in the **MaskedInputCollection** round into $(v'||u'||s_{v,u}^{SK}||b_{v,u})$ and assert that $u = u' \wedge v = v'$.
Bob decrypts the ciphertext from Alice with the encryption key previously computed.
3) Send a list of shares to the server, composed of $s_{v,u}^{SK}$ for users $v \in U_2 \setminus U_3$, and $b_{v,u}$ for users in $v \in U_3$.
The set U_3 is composed of Alice, Bob, and Charlie, while Daniel has dropped between U_2 and U_3. Thus, Bob sends his share of Daniel's secret key, while he sends his share of Alice, Charlie and himself self mask seed.

Server:

4) Collect responses from at least t users. For each user $u \in U_2 \setminus U_3$, reconstruct s_u^{SK} and use it (together with the public keys received in the **AdvertiseKeys** round) to recompute $p_{v,u}$ for all $v \in U_3$ using the PRG.
The server receives the shares from Alice and Bob, reaching the threshold of 2. It can reconstruct Daniel's secret key, to then use it along with Bob's public key to retrieve $s_{Bob,Daniel}$.
5) For each user $u \in U_3$, reconstruct b_u.
The server reconstructs Bob self mask seed.
6) With PRG, compute and output $z = \sum_{u \in U_3} x_u$ as $\sum_{u \in U_3} x_u = \sum_{u \in U_3} y_u - \sum_{u \in U_3} p_u + \sum_{u \in U_3, v \in U_2 \setminus U_3} p_{v,u}$.
The server removes the pairwise mask of Alice, Bob, and Charlie with Daniel. Then, it removes the self mask from Alice, Bob, and Charlie. In the end, it retrieves the model vectors sum of Alice, Bob, and Charlie.

IV. Local implementation

The final goal is to implement the protocol directly in DeAI [4], an open-source web platform dedicated to collaborative learning. We have implemented a simplified version of the protocol in JavaScript [33] and Node.js [34], as the first step towards this goal. Implementation code can be found on GitHub.

This local implementation is a meaningful first move towards the protocol implementation in DeAI. To our knowledge extent, an open-sourced JavaScript implementation of the scheme does not exist. The insights gained when facing JavaScript design choice will be reapplied when integrating the protocol in DeAI.

In this section, we will overview how the protocol has been implemented in JavaScript and Node.js, explain some implementation choices, and present the performance benchmark.

A. Overview

1) General: The local implementation thoroughly follows the protocol and possesses its properties. The implementation can securely compute sums of vectors in a constant number of rounds and is robust to failures. It uses plain Javascript and two Node.js packages.

This implementation can be run for an arbitrary number of clients, and dropouts can be simulated during the protocol at any stage.

2) Simplifications: The local implementation uses some communication and model simplification.

It relies on a non-interactive sequential code, on the contrary of a web application that is interactive between the clients and the server. This simplification allows the server to directly fetch information from the clients and to directly notify them. Thus, the communication protocol is not comparable to what it will be in the DeAI implementation.

The non-interactive sequential code performs all clients computations on the same device, while it is done in parallel by each client in the protocol. This sequential code simplification has been made to focus on the client-side and the server-side parts of the protocol, rather than on an API between the clients and the server which would be almost entirely redesigned when implementing the protocol in DeAI.

The local implementation simplifies the model shared to only one numeric value and not a vector. Generalizing from a single value to a vector is relatively simple. However, each task in DeAI uses different model parameters, which will lead to vector changes when implementing in DeAI depending on the model. Thus, we decided to not implement the protocol on vectors in the local implementation to directly perform it in DeAI.

In the local implementation, clients directly compute their pairwise seeds with other clients when receiving the public keys at Round 1 instead of Round 2. This change does not affect the protocol and simplifies the implementation.

3) Libraries: The implementation relies on three libraries, all cryptography-related. We follow the cryptography principle of using existing libraries rather than trying to implement algorithms by ourselves. The three libraries are:

- **SubtleCrypto** [35]: This library provides several low-level cryptographic functions. It can generate keys based on various algorithms, import and export keys, encrypt and decrypt messages based on a key, and derive a key or a number from a master key.
- **random-seed** [36]: This library enables the creation of seeded pseudorandom generators.
- **shamir** [37]: This library implements Shamir's Secret Sharing algorithm [31].

4) Helper Class: We defined a helper class, providing cryptographic functions. This helper class only contains static functions.

5) Data Storage Classes: The clients and the server store information about other clients: the server stores client IDs and public keys, while the clients store in addition to that the derived shared key and seed. The classes *ClientForClient* and *ClientForServer* structure how one client is stored in another client or in the server. Each client, as well as the server, has a property storing known clients and their information. This property is an object having as keys clients' ID and as value an instantiation of these classes.

6) Client and Server Classes: The client and server classes perform all computations and storage of the protocol based on the rounds.

The client ID is an alphanumeric ID, and we use the lexicographic order to order the different IDs between them.

A Boolean *isUp* in the client class represents a client dropout during the protocol. This Boolean is initially set to True. Once it is changed to False, the server will discard the client in its computation and the client will not perform any further computations.

7) Communication: The server directly fetches in the clients the values computed that are stored as properties. It broadcasts information to clients by calling clients' functions.

8) Running it all together: :

The functions *runSimple* and *runPersonalized* launch the protocol. At the end of the two functions, the program logs the protocol aggregation result as well as the direct sum of clients' secret values.

The function *runSimple* launches a protocol with 4 clients and a threshold of 2 users. The user can manually change the secret value and simulate any dropout. Client IDs are randomly defined but can be manually changed if needed.

The function *runPersonalized* takes as arguments the number of clients n, the threshold t, and the number of dropouts at 6 possible spots. The spots when clients can drop are after client computation and after server computation in rounds 1, 2, and 3. The function allows simulating all possible dropout combinations. The client IDs, as well as the secret values, are randomly defined following the uniform sampling limits.

B. Implementation choices

Choices had to be done when transitioning from a theoretical algorithm to a practical implementation. In this subsection,

we will go through the most important choices made during the implementation of the algorithm, explain the reasoning, and describe the impact of these choices.

1) Cryptographic Primitive: We used the following cryptographic algorithms to implement the protocol:

1) For Key Agreement, we used Elliptic-Curve Diffie-Hellman over the P-521 Curve.
2) For Authenticated Encryption, we used AES-GCM with 256-bit keys.
3) For Pseudorandom Number Generator, we used Gibson Research Corporation's Ultra-High Entropy Pseudo-Random Number Generator [38].

We decided to follow the original paper [3] algorithms for Key Agreement and Authenticated Encryption, augmenting the security with longer keys at the cost of efficiency. We consider the protocol to be efficient enough to increase the computation time in favour of better security guarantees.

For the Pseudorandom Number Generator, we decided to use Gibson Research Corporation Ultra-High Entropy Pseudo-Random Number Generator [39] instead of AES in counter-mode because the PRG is natively developed in JavaScript and is extremely fast.

Using AES-GCM for Authenticated Encryption, we send the initialization vector (e.g. a Nonce) in plaintext when exchanging the ciphertext. Nonce can safely be exchanged in plaintext. A malicious server can tamper the nonce, but it is as relevant as a malicious server directly tampering with the ciphertexts.

2) Shamir Secret sharing: The shamir library [37] provides an implementation of Shamir's t-out-of-n Secret Sharing [31]. It asks for a string as a secret, encodes it into a polynomial of t degree, and extracts n shares $(x, f(x))$ from the polynomial. The polynomial's parameters can be reconstructed from t shares using linear interpolation.

For each user u, the plaintexts $(u||v||s_{u,v}^{SK}||b_{u,v})_{v\in U_1\setminus\{u\}}$ must be a string. Client IDs are already alphanumeric, and $b_{u,v}$ is a number that can easily be converted to a string. To switch from an ECDH key s_u^{SK} to a string representation, we use the *exportKey* function from SubtleCrypto to transform the key into the JSON Web Key format, which can then be converted to a string. We concatenate the different elements of the plaintext in a string, using the '—' as a separator between them. When deciphering the ciphertexts, we split the string based on the separator to retrieve the different elements. To reconstruct the secret from t shares $(x, f(x))$, the server requires t shares of $f(x)$ but also their respective index x. Thus, we concatenate the index of the share x to the plaintext before encrypting it - the plaintext of the message from u to v becomes $(u||v||s_{u,v}^{SK}||b_{u,v}||x)$. During the Unmasking round, the index is extracted by the client and sent to the server.

3) Uniform Sampling: Defining the range of values [0, R), uniformly sampling over this range, and performing the $(mod\ R)$ operation is necessary for the protocol security. If it is not done, an attacker might infer the gradient value by looking at the masked gradient. For example, assume that the range of values is [0, 100), the model gradient is a single value x_u and there are 10 clients. If x_u, the pairwise masks and the self mask all take as value 100, the masked gradient will then take 1100 as value. The server can then infer from the masked gradient the true value of the gradient.

We thus have to define the range of values [0, R), and uniformly sample over this range while performing the $(mod\ R)$ operation everytime there is a gradient-related computation. It seems counter-intuitive to perform these operations in JavaScript. Indeed, the values possibly taken by the gradient are not bounded and can take positive and negative values.

We will take advantage of the two's complement bit representation of an integer [40], the fixed and float points representations [41], and the integer overflows and underflows [42] to perform uniform sampling. We will switch from the JavaScript floating-point representation which does not allow overflows to a fixed point representation encoded as a bitstring of size k, taking integer values in the range [0, $2^k - 1$]. This bitstring will be interpreted with the two's complement representation to reach negative and positive values.

We define the total number of bits used to represent a number as k. We also define the number of fractional part digits as *fracNbDigits*. We know that a digit needs $log_2(10) \approx 3.32$ bits to be represented. Hence, we can compute a loose bound on the number of bits used by the fractional part of the number, *fracNbBits* $= \lceil fracNbDigits * log_2(10) \rceil$.

We can then estimate the number of bits available for the integer part *intNbBits* = *k* - *fracNbBits*. With the two's complement representation, we can compute loose bounds of the value represented by the bitstring *looseBounds* = $[-2^{intNbBits-1}, 2^{intNbBits-1} - 1]$.

Mask values must be generated as a random bitstring of size k, and arithmetic operations should be done directly on the bitstring. This procedure allows bitstring's overflow and underflow that will act as a modulo. We note that the sum of the gradient values must be able to be represented by the bitstring. The sum will overflow or underflow if it is not the case and the final result will not make sense. Assuming there are n clients, we can define loose bounds of the gradient values range *looseBoundsSecret* = $[-\frac{2^{intNbBits-1}}{n}, \frac{2^{intNbBits-1}-1}{n}]$.

We have implemented this protocol by leveraging JavaScript *ArrayBuffer* and *DataView*. These native JavaScript objects allow us to switch from the JavaScript floating-point number representation to a 32-bit bitstring representation. Arraybuffer operations achieve overflows and underflows, while they raise an error in traditional JavaScript number operations.

In our implementation, we opted for the following bit repartition: *k* = 32, *fracNbDigits* = 4, *fracNbBits* = 14, *intNbBits* = 18, *looseBounds* = $[-2^{17}, -2^{17} - 1] = [-131072, 131071]$.

We observe that using uniform distribution drastically reduces the values range that can be taken by the gradient. In our 32-bit implementation, assuming there are n = 100 clients, the gradient values can only be in the interval $[-1310, 1310]$, with a precision of 4 fractional digits. As we don't know the model input and thus the values taken by the gradient, this bound might be too restrictive. Limitations might happen when

the gradient needs to take a high absolute value or a precise value.

C. Benchmark

We benchmarked the implementation in our laptop, a Lenovo T480s with an Intel Core i7-8650U CPU and no GPU. We computed the running time of a client who doesn't drop out during the protocol and of the server. The client and server running times were computed by summing the times in which the program ran respectively the Server instanciation or one Client instanciation.

We assumed that clients drop out at the end of round 1, as it is the "worst case" dropout. Remaining clients will still generate masks and symmetric encryption keys with the dropped clients. The server will have to generate dropped clients' secret key, a more expensive task than simply joining the shamir parts. We simulated 0%, 20%, 40% and 60% dropouts.

Two different settings were benchmarked:

1) **Setting A:** Simulating 5 to 50 running clients with a step of 5. Each simulation was performed 10 times and averaged. It is more precise and is based on a smaller number of clients.
2) **Setting B:** We simulated 20 to 200 clients with a step of 20. Each simulation was performed 2 times and averaged. It is less precise and is based on a higher number of clients.

Our laptop technical limitation prevented us to simulate more clients or to increase the number of iterations per benchmark. Curious readers can check the original paper's prototype performance [3] which was performed with more iterations and on an higher number of clients. Readers should be aware that the paper's results and this benchmark were performed under different conditions and are not comparable: the hardware, the programming language, the data vector size and the cryptographic algorithms used are different.

Figure 5. Wall-clock running time per client in setting A as the number of clients increases.

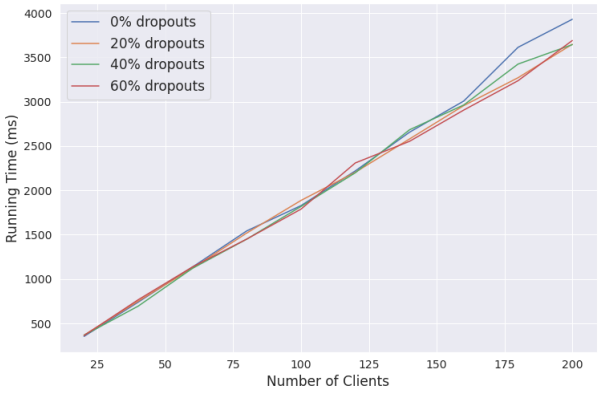

Figure 6. Wall-clock running time per client in setting B as the number of clients increases.

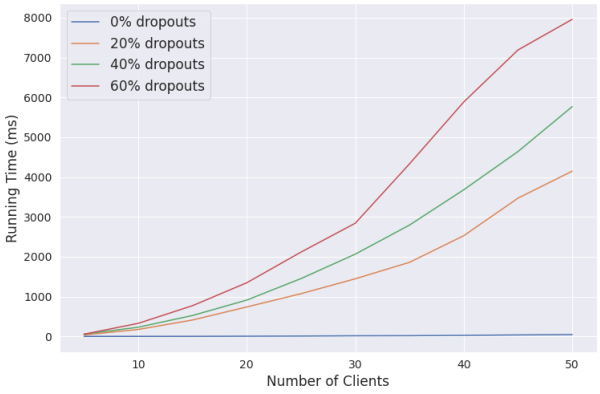

Figure 7. Wall-clock running time per server in setting A as the number of clients increases.

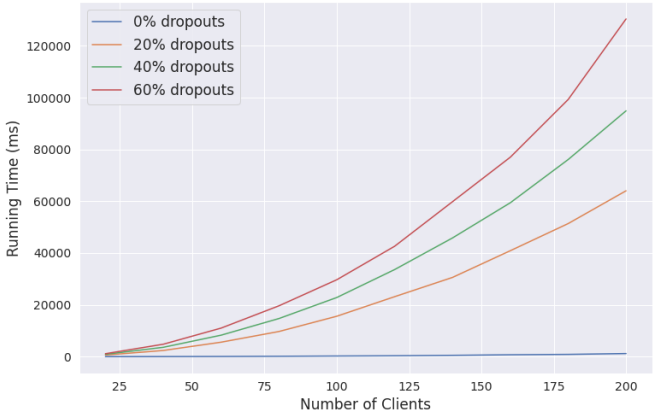

Figure 8. Wall-clock running time per server in setting B as the number of clients increases.

	Num. Clients	Dropouts	AdvertiseKeys	ShareKeys	MaskedInputColl.	Unmasking	Total
Client	50	0	11.2 ms	974.3 ms	0.3 ms	5.7 ms	991.5 ms
Server	50	0	0.1	5.6 ms	2.1 ms	34.1 ms	41.8 ms
Server	50	20%	0.3 ms	4.3 ms	1.5 ms	4138.9 ms	4145.1 ms
Server	50	40%	0.0 ms	4.1 ms	0.7 ms	5760 ms	5765 ms
Server	50	60%	0.1 ms	4.4 ms	0.4 ms	7949.9 ms	7955 ms
Client	200	0	6 ms	3877.5 ms	0.5 ms	45.5 ms	3230 ms
Server	200	0	0.0	75.5 ms	66.0 ms	976.5 ms	1118 ms
Server	200	20%	0.5 ms	85.5 ms	51.5 ms	63829.5 ms	63967 ms
Server	200	40%	0.5 ms	83.0 ms	30.5 ms	94734 ms	94848 ms
Server	200	60%	0.0 ms	79.5 ms	15.0 ms	130248.0 ms	130343 ms

Table I
RUNNING TIMES PER ROUND.

The computation cost comes almost entirely from the cryptographic operations, e.g. key and number agreement and PRG generation. We also note that the presence of dropouts does not significantly change clients' running time. In contrast, client dropouts significantly increase the server's running time.

V. DEAI IMPLEMENTATION

DeAI [4] is an open-source project enabling collaborative and privacy-preserving training of ML models. DeAI offers both decentralized and federated learning.

DeAI runs in a browser or from a mobile app. Client's data never leave local devices: local deep learning models are computed directly, and exchanged with peers or the central server.

We aim at implementing the secure aggregation protocol in DeAI in the future. As such, we will briefly present DeAI, its structure and its functioning. We will then discuss on the next steps to implement the aggregation scheme in DeAI.

A. DeAI presentation

In this subsection, we will describe the DeAI project structure. We will focus on the functioning of the project's part that are the most interesting for the protocol implementation.

DeAI is divided into two parts: a client-side application written in Vue.js and a server-side application written in Node.js.

The client can choose between a federated or a decentralized setting. Choosing one or the other will activate different endpoints in the server. We will only focus on the federated setting as it is the only relevant one with the protocol.

The communication follows a traditional server-client scheme: only the clients can send requests to the server's endpoints. The server can't notify the clients or ping them.

The code for the client-side application is located at the folder *mobile-browser-based-version/src*. The different files described below have this folder as their path's root. Relevant client-side files are:

- *helpers/communication/federated/api.js* is a wrapper responsible for sending requests to the server.
- *helpers/communication/federated/client.js* chooses the correct arguments for the requests and processes the values returned by the server. This file also defines the federated client abstraction.
- *components/building_frames/containers/TrainingFrame.vue* links the user's interaction with the application from the front-end to the back-end.
- *helpers/training/federated/training_informant.js* collects information about the status of the model training-loop.
- *helpers/training/TrainingManager.js* handles the model memory management and training.
- *helpers/memory/helpers.js* handles the loading and the saving of the trained TFJS model.

During a training round, the client behaves as such:

- At the beginning of a model training round, the client creates the model or load the previous one if it is not the first round.
- The client makes sure that it has been selected by the server for this training round. It regularly pings the server, waiting for a positive response to begin its local training.
- After training locally its model with its local data, the client sends its model to the server. Then, it regularly pings the server, asking if the aggregation has been done.
- When the aggregation has been done, the client fetches the aggregated model in the server and begins another training round.

The code for the server-side application is placed in the folder *mobile-browser-based-version/server*. The different files described below have this folder as their path's root. Relevant server-side files are:

- *router/router.js* defines the different endpoints to reach the server.
- *logic/federated/handlers.js* handles the request and performs all server computations.

During a training round, the server behaves as such:

- Clients trying to connect to the server are stored by the server. If a training round has already begun, they are placed in a queue for the next training round. The server ensures that the clients are alive by regularly checking if the clients have recently pinged the server and drops the inactive clients.
- When a training round begins, the server selects the combination of the previous round's clients with the queuing clients as the new round selected clients. The server answers positively if any of the selected clients asks the server if it has been selected.

- The server waits for a given number of clients to send their model. This number of clients is based on a threshold. When the threshold is reached, the server begins a countdown before aggregating received models.
- After the aggregation has been made, the server starts a new round. The most recent aggregated model can always be fetched by the clients.

B. Discussion on the implementation

In this subsection, we will discuss on how to implement the secure aggregation protocol in DeAI. We warn the reader that it is only a discussion and is prone to change during the implementation. To distinguish between the DeAI rounds and the protocol rounds, we will call the DeAI training rounds "rounds" and the protocol rounds "subrounds".

The implementation of the protocol should rely on the current implementation of DeAI. At each round, we need to achieve the subrounds 0, 1, and 2 before sending the masked gradient. Then, the round 3 must be accomplished to retrieve the gradient aggregation. We can add this routine after the model has been trained, performing the protocol on the callback *onEpochEnd*.

Communication-wise, the local implementation allows the server to directly broadcast and fetch data to the clients. We will switch to a client-server application where the server is not able to notify the clients. We advise to reuse the current communication implementation of DeAI.

When the server is collecting information from the clients, it can open an endpoint. The clients can send their information through a request to this endpoint. As soon as the threshold t of requests is reached, a countdown that close the endpoint when completed can be triggered. This procedure allows to reach the threshold t and to potentially exceed it while ensuring communication efficiency.

When the server needs to send messages to the client, it can open an endpoint. This endpoint will be accessible by clients, who will be able to fetch the desired information. The clients will have to constantly ping this endpoint until they receive a valid answer.

Data storage classes should be stored in the memory folders. One memory folder can be created on server-side of the implementation. The cryptographic helper class should be stored in the helper folders.

We would like the user to be able to choose between the secure aggregation scheme and traditional federated learning. Indeed, choosing the secure aggregation scheme increases privacy guarantees but at the cost of efficiency. We don't want to remove a choice to the user when implementing a new feature.

With the ongoing development of DeAI, it is likely that other features will be added to the application. The user should be able to choose which features he'd like to enable.

Currently, an user can choose between decentralized or federated learning. Depending on the user's choice, the application will use different endpoints and a different logic. We agree with this implementation for the choice between decentralized and federated learning as the schemes are totally different. Nonetheless, we don't recommend to use a totally new set of endpoints for the secure aggregation protocol.

Even though the protocol adds communication and computation to traditional FL, it follows the same basic logic. We think it is better to exploit the already implemented code base and endpoints. Refactoring completely the application and defining a new set of endpoints everytime a new feature is added is not a viable and scalable option.

We recommend to use a middleware architecture. With this architecture, a client will be able to choose the features it would like to use when connecting to the FL server. Depending on the chosen features, the same endpoint can redirect to different back-end implementations. These back-end implementations will be carried out by extending the traditional FL code base, using it as a foundation while providing new endpoints and overwriting its internal logic if needed.

This approach add new features and leverage the existing code base at the same time. With this approach, the client will be able to choose the features he'd like to enable. It will then use the same endpoints, while the back-end logic will have been changed based on the client's choice.

VI. Limits and Future work

A. Limits

Vulnerability to privacy-related attacks: By masking clients' gradient to the server, the protocol prevents the attacks from a malicious server targeting specifically one client like [21]. However, client-side attacks are still possible even with a secure aggregation scheme. For example, the protocol is still vulnerable to membership inference attacks [17], [18] or GAN-based attacks [20].

Addressing malicious clients: The protocol prevents the server to learn any information aside from the aggregation. However, one malicious client can stop the server to get a correct aggregated model by sending an arbitrarily bad model. As the protocol protects the client's model privacy, the server will not be able to check the model and will accept the input.

Values range: The range of values that can be taken by the model decreases when the number of clients increases, as explained in section IV.B.3). It is due to the use of the floating-point representation to perform uniform sampling, less flexible than the fixed-point representation.

B. Future work

Implementing the protocol in DeAI: With the insights gained in the local implementation, the next step is to implement the secure aggregation scheme in DeAI.

Handling malicious adversaries: The protocol could be improved to be resistant against malicious adversaries if we use a Public Key Infrastructure.

Extending the protocol to 64 bits: We have previously explained why uniform sampling sets a restrictive bound to the possible values taken by the gradient. Expanding the fixed-point number representation from 32 to 64 bits is the solution to extend the range of the possible values taken by the gradient.

Increasing efficiency with gradient compression: In a distributed setting, the gradient exchange is mostly redundant. As it requires a significant communication bandwith, compressing the gradient [43] will increase the communication efficiency.

Extending the protocol to a decentralized topology: The protocol is valid on a federated learning setting. We could develop the protocol to be extended into a decentralized topology. However, some clients might not be willing to share their peers, and different topologies can exist for a decentralized setting.

Allowing model comparisons while still preserving the privacy: The secure aggregation protocol prevents gradient comparison, and thus personalized federated learning with schemes such as Weight Erosion [44], WAFFLE [45] or interoperability [16]. Being able to combine these two properties would allow us to have a customized model that does not reveal the gradient. This would be possible if a metric could be used to compare gradients without revealing them. We believe in the possibility that techniques like differential privacy [22] or vector embeddings [46] can be used to calculate these metrics. It will be necessary to prove that it is not possible to retrieve information on the gradients from the combination of the metrics and the masked gradient.

VII. Conclusion

By reviewing the literature around Privacy-Preserving Machine Learning, we highlighted the privacy threats against a Federated Learning platform like DeAI. We explained how the secure aggregation scheme from Bonawitz et al. can add security guarantees to Federated Learning, and described its functioning to the reader.

We successfully implemented the protocol using Javascript as there were not any open-sourced implementation in this programming language. We made implementation choices and paved the way for a future implementation of the protocol in DeAI. This future implementation will make the platform even more secure against privacy-related attacks, hence attracting a new category of users who were concerned about security breaches.

References

[1] Jim Isaak and Mina J Hanna. User data privacy: Facebook, cambridge analytica, and privacy protection. *Computer*, 51(8):56–59, 2018.

[2] Gpt-3 cost article. https://venturebeat.com/2020/06/01/ai-machine-learning-openai-gpt-3-size-isnt-everything/.

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

[4] Deai github. https://github.com/epfml/DeAI. Accessed: 2022-01-19.

[5] Issam El Naqa and Martin J Murphy. What is machine learning? In *machine learning in radiation oncology*, pages 3–11. Springer, 2015.

[6] Maxwell W Libbrecht and William Stafford Noble. Machine learning applications in genetics and genomics. *Nature Reviews Genetics*, 16(6):321–332, 2015.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[8] Jules J Berman. Confidentiality issues for medical data miners. *Artificial intelligence in medicine*, 26(1-2):25–36, 2002.

[9] Nathan L Yozwiak, Stephen F Schaffner, and Pardis C Sabeti. Data sharing: Make outbreak research open access. *Nature News*, 518(7540):477, 2015.

[10] B Goldacre, S Harrison, KR Mahtani, and C Heneghan. Who consultation on data and results sharing during public health emergencies. background briefing. sep 2015 centre for evidence-based medicine background briefing: Who consultation on data and results sharing during public health emergencies background briefing for who consultation on data and results sharing during public health emergencies. 2015.

[11] Google fl application. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html. Accessed: 2022-01-19.

[12] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.

[13] H Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agüera y Arcas. Federated learning of deep networks using model averaging. *arXiv preprint arXiv:1602.05629*, 2016.

[14] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *arXiv preprint arXiv:1912.04977*, 2019.

[15] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

[16] David Roschewitz, Mary-Anne Hartley, Luca Corinzia, and Martin Jaggi. Ifedavg: Interpretable data-interoperability for federated learning. *arXiv preprint arXiv:2107.06580*, 2021.

[17] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2017.

[18] Luca Melis, Congzheng Song, Emiliano De Cristofaro, and Vitaly Shmatikov. Exploiting unintended feature leakage in collaborative learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 691–706. IEEE, 2019.

[19] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[20] Briland Hitaj, Giuseppe Ateniese, and Fernando Perez-Cruz. Deep models under the gan: information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 603–618, 2017.

[21] Zhibo Wang, Mengkai Song, Zhifei Zhang, Yang Song, Qian Wang, and Hairong Qi. Beyond inferring class representatives: User-level privacy leakage from federated learning. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2512–2520. IEEE, 2019.

[22] Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages, and Programming*, pages 1–12. Springer, 2006.

[23] Robin C Geyer, Tassilo Klein, and Moin Nabi. Differentially private federated learning: A client level perspective. *arXiv preprint arXiv:1712.07557*, 2017.

[24] H Brendan McMahan, Daniel Ramage, Kunal Talwar, and Li Zhang. Learning differentially private recurrent language models. *arXiv preprint arXiv:1710.06963*, 2017.

[25] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *arXiv preprint arXiv:2004.02229*, 2020.

[26] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 35–52, 2018.

[27] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017.

[28] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019(3):26–49, 2019.

[29] Andrew Paverd, Andrew Martin, and Ian Brown. Modelling and automatically analysing privacy properties for honest-but-curious adversaries. *Tech. Rep*, 2014.

[30] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep

networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.
[31] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
[32] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM journal on Computing*, 13(4):850–864, 1984.
[33] Javascript website. https://developer.mozilla.org/fr/docs/Web/JavaScript. Accessed: 2022-01-19.
[34] Node.js website. https://nodejs.org/en/about/. Accessed: 2022-01-19.
[35] Subtle crypto library documentation. https://developer.mozilla.org/fr/docs/Web/API/SubtleCrypto. Accessed: 2022-01-19.
[36] random seed library documentation. https://www.npmjs.com/package/random-seed. Accessed: 2022-01-19.
[37] shamir library documentation. https://www.npmjs.com/package/shamir. Accessed: 2022-01-19.
[38] Gibson research corporation website. https://www.grc.com/otg/uheprng.htm. Accessed: 2022-01-19.
[39] Gibson research corporation website. https://www.grc.com/otg/uheprng.htm. Accessed: 2022-01-19.
[40] Two's complement wikipedia page. https://en.wikipedia.org/wiki/Two%27s_complement. Accessed: 2022-01-19.
[41] Article on fixed and float points representations. https://pediaa.com/difference-between-fixed-point-and-floating-point/. Accessed: 2022-01-19.
[42] Overflow wikipedia page. https://en.wikipedia.org/wiki/Integer_overflow. Accessed: 2022-01-19.
[43] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.
[44] Felix Grimberg, Mary-Anne Hartley, Martin Jaggi, and Sai Praneeth Karimireddy. Weight erosion: An update aggregation scheme for personalized collaborative machine learning. In *Domain Adaptation and Representation Transfer, and Distributed and Collaborative Learning*, pages 160–169. Springer, 2020.
[45] Martin Beaussart, Felix Grimberg, Mary-Anne Hartley, and Martin Jaggi. Waffle: Weighted averaging for personalized federated learning. *arXiv preprint arXiv:2110.06978*, 2021.
[46] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 302–308, 2014.