IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 8 November 2023, accepted 15 December 2023, date of publication 25 December 2023,
date of current version 9 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3347039

RESEARCH ARTICLE

Federify: A Verifiable Federated Learning Scheme Based on zkSNARKs and Blockchain

GHAZALEH KESHAVARZKALHORI[1], CRISTINA PÉREZ-SOLÀ[1], GUILLERMO NAVARRO-ARRIBAS[1], JORDI HERRERA-JOANCOMARTÍ[1], AND HABIB YAJAM[2]

[1]Department of Information and Communications Engineering, Autonomous University of Barcelona, 08193 Bellaterra, Spain
[2]Department of Computer and Electrical Engineering, University of Tehran, Tehran 1417935840, Iran

Corresponding author: Ghazaleh Keshavarzkalhori (Ghazaleh.Keshavarzkalhori@uab.cat)

This work was supported in part by the Spanish Ministry under Grant PID2021-125962OB-C33 SECURING/NET, Grant PID2021-125962OB-C31 SECURING/CYBER, and Grant TSI-063000-2021-151 FREE6G-Security; and in part by the Catalan Agència de Gestió d'Ajuts Universitaris i de Recerca (AGAUR) under Grant SGR2021-00643 and Grant SGR2021-01508.

ABSTRACT Federated learning (FL) has emerged as an alternative to traditional machine learning in scenarios where training data is sensitive. In federated learning, training is held at end devices, and thus data does not need to leave users' devices. However, most approaches to federated learning rely on a central server to coordinate the learning process which, in turn, introduces its own security and privacy problems. We propose Federify, a decentralized federated learning framework based on blockchain that employs homomorphic encryption and zero knowledge proofs to provide security, privacy, and transparency. The scheme successfully preserves the confidentiality of both the data used for training and the local models using homomorphic encryption. All model parameters are publicly verifiable using zkSNARKs, and transparency of both the learning process and the incentive mechanism is achieved by delegating coordination to a public blockchain. The evaluation of the proof of concept of our framework demonstrates its viability both in terms of required computational resources and the cost to train on a public generic blockchain such as Ethereum.

INDEX TERMS Federated learning, verifiable, zkSNARK, blockchain, decentralization, privacy, Ethereum.

I. INTRODUCTION

Federated learning is a machine learning technique that enables multiple parties and devices to train a shared machine learning model based on a dataset that is distributed among them without sharing the data. Since parties do not share datasets, this approach can provide data privacy to some degree. However, the information that parties exchange during the process of training can leak some information about the training dataset.

The emergence of big data has pushed the development of machine learning. Every day large amounts of data are produced that can be used for training machine learning models in fields such as language processing, biometrics, and the Internet-of-Things. However, due to various reasons (e.g. privacy concerns and bandwidth limitations), this data is scattered among many devices and under different sovereignties. This leads to more demand for federated learning.

Another vital challenge that federated learning faces is the need for a verifiable learning process. Training should be conducted in a way that participants cannot interfere with the production of the final model in a way that causes it to generate faulty results. Therefore, in a verifiable federated learning method, a small group of malicious participants should not be able to alter the final trained machine learning model.

Google was the first to introduce federated learning as an industrial solution to the privacy issues of centralized machine learning. The idea of federated learning, as discussed by McMahan et al. [24], is referred to as local training of machine learning models with local data and aggregation of multiple models to retrieve a global model. Similarly, many later works proposed schemes that train the model using a

The associate editor coordinating the review of this manuscript and approving it for publication was Mauro Tucci.

central server, which gathers local models from clients and aggregates them to create a better model. While this model addresses the issue of data islands, it heavily relies on a centralized structure.

Blockchain [27] is a revolutionary idea that comes up when the notion of distribution is brought up. As a consequence, recent research works propose using blockchain or decentralized ledger technology to tackle the problem of having a single point of failure in the federated learning system. A proper decentralized federated learning system based on blockchain technology guarantees that no single party has an upper hand in controlling the training process. All participants have equal power in the system, since the blockchain distributes trust and has transparent transactions in the immutable ledger.

This paper proposes Federify, a federated learning scheme based on smart contracts, zkSNARK proofs, and partially homomorphic threshold cryptography that trains a Gaussian naive Bayes model. Utilizing zkSNARK verification of the data users submit to the smart contract, the proposed scheme prevents model poisoning attacks.

Additionally, we use partially homomorphic threshold encryption to distribute trust among semi-trusted participants. As a result, if the set of semi-trusted participants do not collude with each other, user data privacy will be fully preserved and only the final trained model will be publicly available to everyone. On-chain smart-contracts omit the need for a centralized authority and provide a transparent incentive mechanism for participating users. One important contribution of this research is providing an open-source implementation of our work alongside a performance analysis of the implementation which demonstrates its viability. The following list summarizes the contributions of this paper:

- We propose a decentralized federated learning framework based on blockchain with security, privacy, and transparency guarantees.
  - We introduce the usage of zkSNARK proofs to reject outlier data and prevent model poisoning.
  - We propose to use partially-homomorphic threshold encryption for preserving user data privacy during on-chain model aggregation.
  - The framework includes an incentive mechanism that motivates and rewards participants for their contributions.
  - We evaluate the security properties offered by the scheme.
- We implement an open-source proof of concept of the framework.
  - We analyze the performance of the open-source implementation.
  - We demonstrate the viability of the framework.

The rest of the paper is organized as follows. Section II presents the basic concepts upon which our solution is built: it presents the concept of blockchain-based smart contracts, the federated learning approach to machine learning, the cryptographic primitives used in the scheme, and the classification technique. Then, Section III explains the Federify framework and specifies the algorithms and actions performed in each of its phases. Following this, Section IV describes the implementation that constitutes our proof of concept for the scheme. After that, Section V provides the results of evaluating the proof of concept on the Ethereum network and discusses its security properties. Later on, Section VI explains other works that have some similarities with the proposed scheme and compares their contribution, highlighting the benefits of our framework with respect to the state of the art. Finally, Section VII wraps up the paper with conclusions and further work.

II. PRELIMINARIES

To fully understand Federify, some preliminary knowledge is required. The following section introduces these concepts and explains how they are employed.

A. BLOCKCHAIN AND SMART CONTRACTS

Blockchain has attracted so much attention in the research area of computer science since its inception in 2008 [27]. In brief, it is a distributed immutable ledger, which all the nodes of the network work together to update and keep track of. This ledger is kept in the form of blocks of data that each contain the hash of the previous block.

Ethereum [6] is a platform based on blockchain which supports smart contracts. Smart contracts are defined as ''a computerized transaction protocol that executes the terms of a contract'' by Nick Szabo [36] in the mid 1990s. Ethereum nodes execute smart contracts by the Ethereum virtual machine. When a contract is deployed on the network, the code cannot be altered and all the states are executed automatically, providing execution integrity.

Smart contracts on Ethereum are coded in Solidity and managed by a contract account created by an external account owned by a person. A contract account stores the code of the corresponding contract and is charged some amount of Ether to exchange values with respect to the code. Each line of code executed on Ethereum demands a certain value of gas, which is the minimum metric of value on the Ethereum platform. The external functions of the contract can be accessed by each account by sending a transaction to them. By doing so, some gas is used, which is limited by 30 million per block at the time of writing.

B. FEDERATED LEARNING

Federated learning (FL) is the distributed solution to machine learning. A set of clients $\{C_0, C_1, \ldots, C_N\}$ train a local model with their local data, and send their model or updates to a central server, which aggregates them to obtain a global model. The process is usually iterative, where the central node distributes the global model back to the clients, they send back differences between their local models and the global one, the server updates the global model,

and so on. The process finishes when some condition is met, usually based on the convergence of the global model.

The FL approach was initially praised for its privacy guarantees, given that the data do not leave the client device. There is no need to share data, that might contain sensitive information. This has proven to be a bit illusory [1], [21], [46]. Information leakage can occur at different stages, in the local model of each client, or even in the aggregation can lead to different types of disclosure.

Moreover, FL also needs to face the possibility of having some dishonest clients, which might attempt to make the machine learning model to misbehave. Either to target a specific outcome or simply to make it unusable.

In this sense, there are commonly two types of problems that can arise in FL: security problems aimed to attack the model and process, and privacy problems where the attacker aims to obtain sensitive information from the clients. FL literature usually addresses these problems separately, and just a few proposals attempt to address them at the same time. Some examples of the latter are [19] and [23].

One of the major motivations of our proposal is that it can be considered a fully distributed FL framework, that does not require a central server to aggregate the local models. This decentralization is performed by relying on a blockchain and smart contracts to perform the tasks that are commonly done by a server. The main consequence is that we do not need to place any level of trust in the server and its operations. Note that common FL works assume an *honest*, or *semi-honest* server model. That is a server that follows the protocol but might try to infer information. In our case, no trust needs to be placed in the aggregation operations since it is auditable by any participant having access to the blockchain (which could be public).

Privacy attacks are commonly divided into client-side and server-side. Client-side attacks are usually limited, and only considered when the number of participants is relatively small. It might seem that in our case the server-side does not directly apply, but common server-side attacks can be also considered in scenarios without a server. In some sense the smart contract and its resulting computations and data, which might get published in the blockchain, could lead to some disclosure, similarly to the information that a server could infer in a more common FL scenario. This makes the distinction between server, client or even external attacker a bit unnecessary. We consider that privacy attacks can be performed by any entity having access to the blockchain. The common privacy attacks in FL are:

- Membership inference attacks: in this case, the attacker attempts to determine if some specific data is part of the training data set [14], [25], [28], [43].
- Property inference attacks refer to the disclosure of properties of the training data not the data itself (e.g. location, testing environment, ...) [12], [25], [42].
- Distribution estimation attacks: here, the attacker attempts to determine the distribution of the client data [18].
- Reconstruction attacks: possibly the most ambitious attack, where the attacker tries to obtain the original data [13], [32], [45], [46].

Security attacks are usually performed by malicious clients, which aim to influence in the final model. The literature commonly characterizes them into two types:

- Model poisoning, where the malicious client tries to manipulate the local model parameters before they are aggregated into the global one [2], [5], [40].
- Data poisoning: the client injects biased or wrong data into the training data [4], [11].

These attacks also might differ on the goal of the attacker, distinguishing between targeted and untargeted attacks. The targeted attacks try to force the global model to make a specific incorrect prediction, while the untargeted ones attempt to prevent the convergence of the global model or force it to converge to an arbitrary wrong state.

C. THRESHOLD ElGamal CRYPTOSYSTEM

The proposed scheme provides confidentiality of both the data used for training and the local models through homomorphic encryption using ElGamal encryption. By sharing the private key among multiple decrypting parties, the threshold ElGamal cryptosystem ensures that the plain text data is not accessible to any single party. The ElGamal cryptosystem also has the property of partial homomorphism that can be used for adding content of two ciphertexts [30]. In a (t, m)-threshold decryption scheme, for the decryption of a ciphertext, participation of at least t out of the total m number of decrypting parties is required. The value of t is predetermined by protocol designers. There are five algorithms that are part of a partial homomorphic threshold ElGamal cryptosystem:

– $P \leftarrow \mathsf{Setup}(s_1, \ldots, s_m)$. Every user U_k of the set of all m users has its secret key s_k and public key $P_k = s_k G$. The global public key of the scheme is:

$$P = \sum_{k=1}^{m} P_k$$

Then, every user U_k shares his private key s_k using a polynomial secret sharing scheme. To do so, every user U_k chooses at random a polynomial $f_k(z)$ of degree $t-1$:

$$f_k(z) = f_{k,0} + f_{k,1} z + f_{k,2} z^2 + \cdots + f_{k,t-1} z^{t-1}$$

such that $f_k(0) = f_{k,0} = s_k$. U_k also computes a commitment for every polynomial coefficient $F_{kj} = f_{k,j} G$, for $j = \{0, \cdots, t-1\}$ and publicly publishes $\{F_{kj}\}_{j=0}^{t-1}$. Afterward, U_k secretly sends the secret share (i, s_{ki}) with $s_{ki} = f_k(i)$ to U_i for $i = \{1, \cdots, m\}$ alongside its signature. Then every user U_i can verify the signature from user U_k and the correction of the share s_{ki} by ensuring the next

equations hold:

$$s_{ki}G = f_k(i)G = \sum_{l=0}^{t-1}(F_{kl}) \cdot i^l$$
$$F_{k0} = f_{k,0}G = P_k$$

– $(R, C) \leftarrow \mathsf{Enc}_P(r, \mathbf{m})$. The encryption of a given plain-text $\mathbf{m}$ on the elliptic curve[1] with the public key P is done by taking a random $r \in (1, n-1)$ and calculating the cipher-text:

$$(R, C) = (rG, \mathbf{m} + rP)$$

where n is the order of G in the curve.

– $(R, C) \leftarrow \mathsf{HAdd}(R_1, C_1, R_2, C_2)$. Homomorphic addition of two ciphertexts (R_1, C_1) and (R_2, C_2) is:

$$(R, C) = (R_1 + R_2, C_1 + C_2)$$

– $(R, C_{s_k}) \leftarrow \mathsf{PartDec}(s_k, (R, C))$. Each user U_k partially decrypts the cipher-text obtaining:

$$C_{s_k} = (C - s_k R)$$

The complete decryption of an encrypted value (R, C) coming from a multiple party key, $P = \sum_{k=1}^{m} P_k$, can be obtained by applying a partial decryption over all subkeys, that is:

$$\mathbf{m} = \mathsf{PartDec}(s_1, \mathsf{PartDec}(s_2, \mathit{PartDec}(\cdots, \mathsf{PartDec}(s_m, (R, C)))))$$

The plaintext is indeed recovered because (R, C) have been obtained by encrypting with the global public key P, thus:

$$C = \Sigma_{k=1}^{m} s_k R + \mathbf{m}$$
$$C - \Sigma_{k=1}^{m} s_k R = \mathbf{m}$$
$$C_{s_k} = C - s_k R$$
$$C_{s_k} - (s_1 R + s_2 R + \cdots s_{k-1} R + s_{k+1} R + \cdots + s_m R) = \mathbf{m}$$

Note that partial decryption order is not important because of the commutative property of addition. Moreover, if user U_k does not cooperate in decryption, then any set of t users can recover his secret s_k using the shares s_{ki} they have [30], so C_{s_k} can be always computed assuming t users are honest.

D. zkSNARKs

A Zero Knowledge Proof (ZKP) is a probabilistic proof in which no extra knowledge is dispensed to the parties except the fact to be proved. Zero-knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARK) proofs have the same property but are more widespread with use cases such as zCash, which is an implementation of the Zerocash [34] payment system. A zkSNARK has the following properties:

- Completeness: There is a need to ensure that any correct proof of the algorithm conforms to the protocol in order to ensure it's working correctly.
- Soundness: Using such an algorithm requires that it makes sure no false proofs are verified. As such, the prover acting faulty could not submit false proofs of a known fact.
- Zero-knowledge: As far as other parties are concerned, no extra knowledge must be divulged to them other than what has already been proved.
- None-interactive: The provider and the verifier exchange only one message during the verification process.

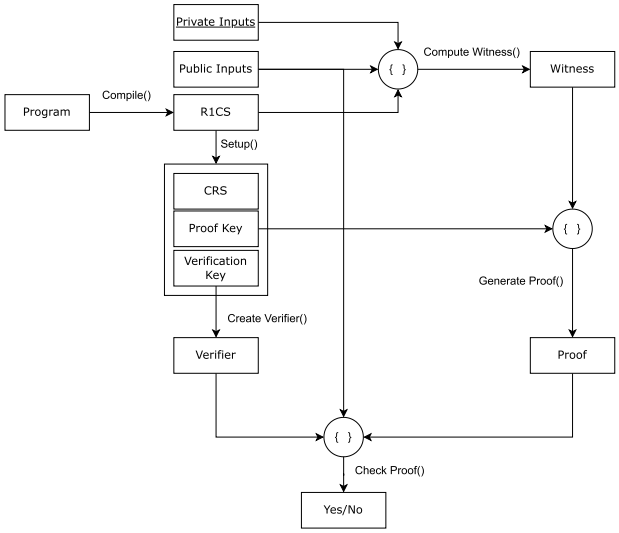

FIGURE 1. Steps of creating a zkSNARK proof from a program.

Figure 1 shows the process of creating and verifying the proof of a program.

zkSNARK usage requires an initial setup phase, generating a common reference string (CRS) shared between the prover and the verifier. The CRS is connected to the verification process by setting a set of variables corresponding to the fact being verified. As a result of the generation of a CRS, there is no longer any need to exchange back and forth messages during the verification process and hence the proof becomes non-interactive.

When creating a zkSNARK, the proving program must be converted into an arithmetic circuit. Arithmetic circuits only consist of the basic four mathematical functions (addition, subtraction, multiplication, and division), which makes a program easier to process. Although arithmetic circuits are simple, they are not the most convenient to work with in terms of privacy. The next steps in generating zkSNARKs are the key points of how all their features are attainable.

The next step in a zkSNARK generation will check if each of the computation steps is correctly computed or, in other

[1]Notice that $\mathbf{m}$ needs to be a point of the elliptic curve. In case $\mathbf{m}$ is an integer, we can take $\mathbf{m}$ as the x-coordinate and compute the y-coordinate to obtain both components of the point.

words, check the constraints of a simplified equation. A Rank 1 Constraint System (R1CS) would check if the values are correctly computed. The act of creating the R1CS from a program is referred to as compilation throughout this paper. The more complex the program, the more constraints are there.

Afterwards, the prover will compute the inputs to the R1CS, the witness, for the specified input values. Program inputs can be passed as private and public. Both of these inputs are needed when generating a witness, but when verifying a proof only public inputs are used. Methods for using these inputs in the R1CS are beyond the scope of this paper.

It is possible to generate the witness with specific input values, but the verification process requires a general method independent of inputs. The next step in zkSNARK generation will generalize these constraints by comparing them between polynomials rather than numbers. Quadratic Arithmetic Programs (QAP) confirm the equality of polynomials by checking random points between them. Polynomials with different coefficients will fail at most points if equality does not hold between them. If you check two polynomials at random points to determine if they are equal, the result will be correct with a high probability.

An initial setup is required between the prover and the verifier in a zkSNARK generation process, prior to generating random valid points for the validation process and CRS mentioned previously. This setup must not be done solely by the prover since if the randomness of the points is known to the prover he/she can create false proofs. The output of this initial setup is a Verification key and Proof key which are the bonded values of the CRS and the checking points.

E. GAUSSIAN NAIVE BAYES

The naive Bayes classifier is a probabilistic classification algorithm where the conditional probability of assigning a data point to a specific class is computed for each feature within the dataset. The final probability of a data point belonging to a particular class is determined through the cumulative probability of its individual features, which can be derived using Bayes' theorem.

When applied to data classification, Gaussian naive Bayes assumes that all features follow a normal distribution. In this context, Gaussian distributions can be expressed as:

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

This probability function is solely dependent on the mean, μ, and variance, σ. Therefore, the naive Bayes classifier calculates the mean and variance of the features in the training dataset, subsequently employing this information to classify the test data.

If data are classified within N classes and have M features, then μ_i^j and σ_i^j represent the mean and the standard deviation of feature i for class j. The model parameters are thus the mean and the standard deviation for each feature and class, resulting to a total of $2MN$ model parameters.

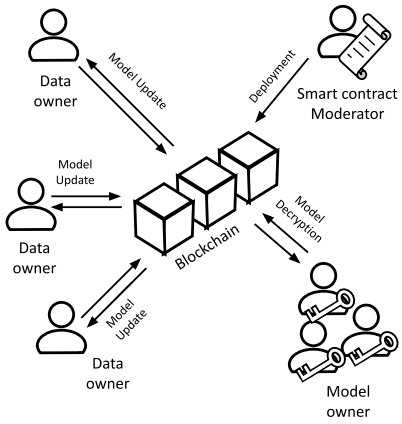

FIGURE 2. Entities participating in the scheme.

In order to classify a data point X, we would compute the probability of this point belonging to each class using the values of its features. Let us denote x_i as the value of the i^{th} feature of X. Then, P_i^j represents the probability associated with the i^{th} feature within class j, which can be derived by applying the Gaussian naive Bayes function utilizing the computed class parameters (μ_i^j and σ_i^j).

$$P_i^j(X) = \frac{1}{\sqrt{2\pi(\sigma_i^j)^2}} e^{-\frac{(x_i-\mu_i^j)^2}{2(\sigma_i^j)^2}}$$

In a naive Bayes classifier, it is assumed that the features of the data are independent, so the final probability of a data point being classified as a specific class is the multiplication of all the conditional probabilities of its features. Hence, the final probability of data point X belonging to class j can be expressed as:

$$P^j(X) = \Pi_{i=1}^{M} P_i^j$$

Finally, the data point is classified into the class that exhibits the highest probability P^j among the N possible classes.

III. FEDERIFY'S SCHEME

In the proposed scheme, a set of model owners (MO) sponsor the training of a global model using private local data from a set of data owners (DO). A smart contract (deployed by the smart contract moderator) in a public blockchain acts as a coordinator (Figure 2).

The execution of our proposed scheme consists of four phases:

1) A setup phase is required to set up all the parameters of the scheme and deploy the smart contract on the blockchain.
2) During the registration phase, model owners register to the smart contract, providing their public keys and some capital.

TABLE 1. Description of the notation used throughout the paper.

Notation	Description
m	Number of model owners
MO_k	k^{th} Model Owner, i.e. $k \in \{1, \cdots, m\}$
t	Threshold for the decryption of ElGamal encryption scheme
G	Generator point of the curve used in the ElGamal encryption
(P_k, s_k)	P_k is the public key of MO_k corresponding to its secret key s_k, i.e. $P_k = s_k G$
P	Aggregated public key used for the encryption of the local models: $P = \sum_{k=1}^{m} P_k$
$(R, C,)$	ElGamal cipher-text as provided by $\mathsf{Enc}_P()$ function in Section II-C
r	Random integer corresponding to the random point R, $R = rG$
DO	Data owner
f	Number of features of the data set
f_i	the i-th feature of the feature set
$\mathcal{D}_j$	Data provided by the DO_j
β	Batch size of the data for every update made to the global model
μ_i	Mean of feature values for feature f_i computed over a local data set.
σ_i^2	Variance of feature values for feature f_i computed over a local data set.
μ_i^*	Mean for feature f_i obtained in the global model.
$(\sigma_i^2)^*$	Variance for feature f_i obtained in the global model.
$\overline{\mu_i}$	Encrypted version of μ_i, i.e. $\overline{\mu_i} = E_P(r, \mu_i)$
$\overline{\sigma_i^2}$	Encrypted version of σ_i^2, i.e. $\overline{\sigma_i^2} = E_P(r, \sigma_i^2)$
$\mathcal{S}$	Set of the local model parameters of a DO for a specific class of the data set, $\mathcal{S} = \{(\mu_i, \sigma_i^2)\}_{i=1}^{f}$
$\overline{\mathcal{S}}$	Encrypted version of $\mathcal{S}$, $\overline{\mathcal{S}} = \{(\overline{\mu_i}, \overline{\sigma_i^2})\}_{i=1}^{f}$
DM_i	The divergence metric corresponding feature i of a DO
DM	The divergence metric for the whole model submission of a DO, $DM = \sum_{i=1}^{f} DM_i$

3) Learning is an iterative process composed of several rounds. Each round consists of four steps: training the local models and aggregation of the global model, decryption of the global model, payment of incentives, and decision-making.
   a) Data owners train their models with local data. To preserve the privacy of such local data, data owners encrypt their model before sending it to the smart contract together with a zkSNARK proof attesting to the correct computation of the encrypted model. If the proof is valid, the smart contract aggregates the local model into the global model.
   b) Model owners collaborate in the decryption of the global model: each model owner partially decrypts the global model and submits its partial decryption (and a zkSNARK proof attesting its correct computation) to the smart contract. The participation of at least t honest model owners ensures the global model decryption is obtained.
   c) Data owners are rewarded for their contribution to the global model. Data owners submit a divergence value quantifying their contribution (together with a zkSNARK proof attesting its correct computation) to the smart contract, and they are rewarded using funding that MOs have paid during registration.
   d) Once the global model is decrypted and MOs incentives are paid, model owners can assess its performance and vote on whether to continue training or stop.
4) Upon finishing training, at the ending phase, the smart contract will distribute the remaining capital to the owners.

The following subsections will thoroughly explain the process of the scheme in each of the phases. The notations used throughout the paper can be found in the Table 1.

An overall diagram of the sequence flow of Federify is presented in Figure 3.

A. SETUP

The moderator does the initial zkSNARK setup described in Section II-D, obtaining the `Verification key` and the `Proof key` for all of the verification processes further referred to in the following sections. Upon completing this step, the program, the obtained `Verification keys`, and the `Proof keys` are shared in a shared file system. This is so each entity can verify that the setup phase is done truthfully.

The moderator also has to deploy the scheme's federated learning smart contract (FLSC) to the blockchain. FLSC aggregates model updates and stores deposits and funds, as well as scheme variables. As a result of conducting *Create Verifier()* for each of the verification steps, a smart contract is created which contains the *Verification Key*. These smart contracts are used for the verifications explained later, such as the model verification smart contract (MVSC) in Section III-C, the decryption verification smart contract (DVSC) from Section III-D, and the incentive verification smart contract (IVSC) in Section III-E.

Since a smart contract cannot change after being deployed, the moderator first configures the scheme variables and then deploys the contract to the blockchain. These variables are batch size (β), number of model owners (m), number of features (f), initial deposit, and minimum submissions (p) for each training phase.

B. REGISTRATION

Each Model Owner (MO_k) registers in the smart contract during the registration phase. As part of this phase, *MO*s need to send the initial deposit, funding, and a local public key, P_k.

On the one hand, the initial deposit serves as an incentive for *MO*s to act honestly, that is, to honestly participate in the decryption process. *MO*s will recover this deposit if they submit correct partial decryptions of the model parameters.

On the other hand, funding will be used to reward Data Owners (*DO*s) for their contributions to the model.

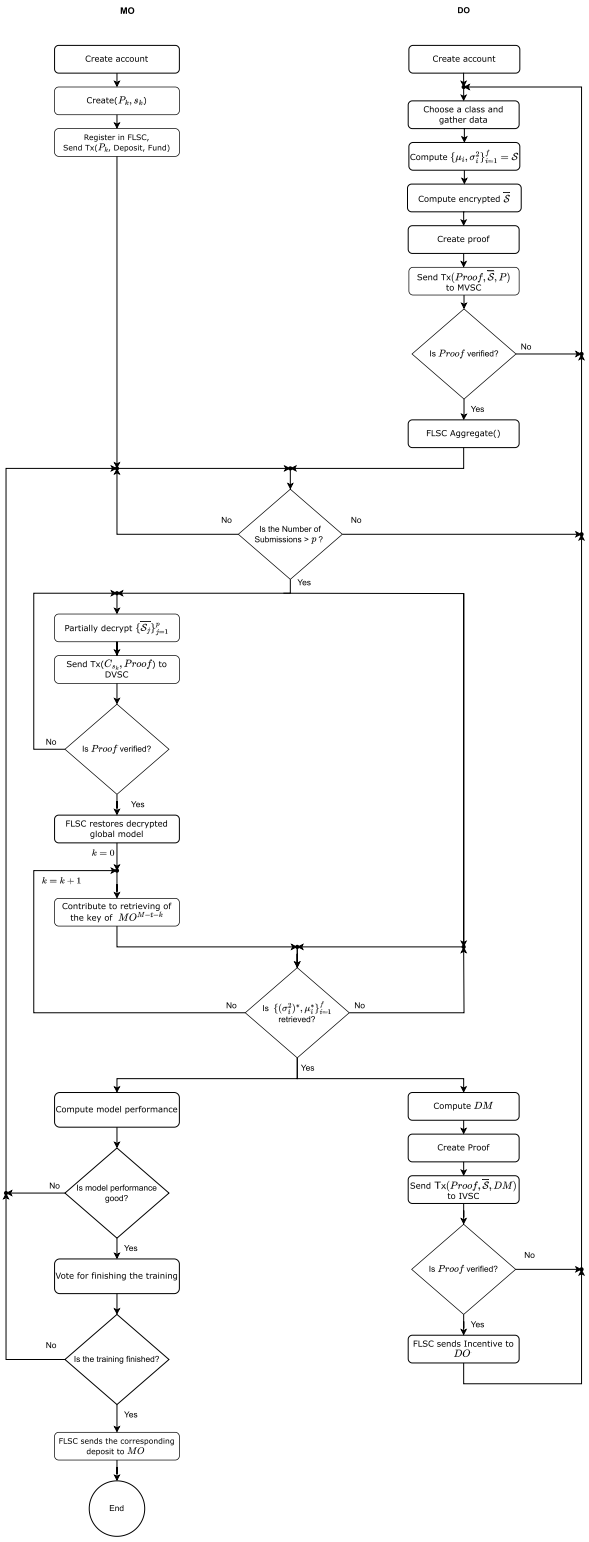

FIGURE 3. Flowchart of the Federify flow. The *DO* and *MO* represent correspondingly the flow of one data owner and the model owner participating in the scheme. Whenever there is a yes/no decision, lines going to the right (respectively, left) correspond to the behavior of the DO (respectively, MO). Each time there is a dot joining different arrows, the direction can be inferred by the missing arrow (i.e. the line without an arrow indicates the outgoing direction).

Finally, the local public key P_k will be used within the threshold encryption scheme, as described in Section II-C. Combining the local public keys sent by the *MO*s will provide the global public key P used for encryption:

$$P = \sum_{1}^{m} P_k$$

Training for the global model can begin once all model owners have registered.

C. LOCAL MODEL TRAINING AND GLOBAL MODEL AGGREGATION

Data owners train a Gaussian naive Bayes (Section II-E) model by updating it repeatedly. The model updates occur in a sequence of classes; whenever data in a specific class reaches the batch size β the *DO* trains its model and sends an update to the global model.

*DO*s will encrypt their local model parameters with ElGamal's threshold encryption scheme using the global public key P computed in the registration phase. The encrypted model each *DO* sends to the smart contract in each round is the set of encrypted parameters for all of the features $\{1, 2, \cdots, f\}$, denoted as $\overline{\mathcal{S}} = \{\overline{\mu_i}, \overline{(\sigma_i)^2}\}_{i=1}^f$, where $\overline{\mu_i} = \mathsf{E}_\mathsf{P}(\mu_i, r_i)$ and $\overline{\sigma_i^2} = \mathsf{E}_\mathsf{P}(\sigma_i^2, r_i')$.

Each *DO* should then create a proof for multiple aspects of its model computations. First, the zkSNARK program *ValidateModel()* would check if the local model parameters are correctly computed from the local data $\mathcal{D}_j$.

Algorithm 1 ValidateModel

Input: $\mathcal{D}_j = \{x_{1i}, \cdots, x_{\beta i}\}, i = \{1, \cdots, f\}, \{(\mu_i, \sigma_i^2)\}_{i=1}^f$
Output: verify
  verify $= true$
  for $i = \{1, \cdots, f\}$ **do**
    $\mu_i' \leftarrow \frac{\Sigma_{k=1}^{\beta} x_{ki}}{\beta}$
    $(\sigma_i^2)' \leftarrow \frac{\Sigma_{k=1}^{\beta} (x_{ki} - \mu_i')^2}{\beta}$
    if $\mu_i' \neq \mu_i$ or $(\sigma_i^2)' \neq \sigma_i^2$ **then**
      verify $= false$
      break
    end if
  end for

After proving that the passed local model parameters are correct, it has to ensure that the encryption process is also done correctly. zkSNARK program *ValidateEncryption()* receives the local model parameters, the encrypted local model parameters, and the global public key P as the input and checks if the encrypted variables are done with the correct public key.

In order to generate the witness, the *DO* must apply *Compute Witness()* on the public inputs, private inputs, and the output of the *Compile()* process. The inputs passed to the

Algorithm 2 ValidateEncryption

Input: $\{(\mu_i, \sigma_i^2)\}_{i=1}^{f}$, $\{(\overline{\mu_i}, \overline{\sigma_i^2})\}_{i=1}^{f}$, $\{r_i\}_{i=1}^{2f}$, P
Output: verify
  verify = true
  for $i = \{1, \cdots, f\}$ **do**
    $\overline{\mu_i'} \leftarrow \mathsf{EP}(\mu_i, r_i)$
    $\overline{(\sigma_i^2)'} \leftarrow \mathsf{EP}(\sigma_i^2, r_{f+i})$
    if $\overline{(\sigma_i^2)'} \neq \overline{\sigma_i^2}$ or $\overline{\mu_i'} \neq \overline{\mu_i}$ **then**
      verify = *false*
      break
    end if
  end for

Compute Witness() are shown below, where underlined items correspond to private inputs.

$$\mathsf{Compute\ Witness}(\underline{\mathcal{S}}, \overline{\mathcal{S}}, \underline{\mathcal{D}_j}, P, \underline{\{r_i\}_{i=1}^{2f}}, \textit{Compile}(\textit{Algorithm 1, 2}))$$

After generating the witness with the inputs necessary for the program, *DO* will apply the *Generate Proof()* on the witness and the `Proof key` which results in the proof.

After creating the proof, the *DO* sends a transaction containing the proof and the public variables of the program such as $Tx(Proof, \overline{\mathcal{S}}, P)$ to the corresponding function of the smart contract to submit an update. The contract aggregates the model upon validating the received proof by adding the encrypted parameters to the class parameters.

Let $\overline{\mathcal{S}_j}$ be the encrypted values of μ_i and σ_i^2 for all features $i = 1, \cdots, f$ provided by DO_j, the encryption aggregation for the first and second DO is defined as $\{\overline{\mathcal{S}_j}\}_{j=1}^{2} = \mathsf{HAdd}(\overline{\mathcal{S}_1}, \overline{\mathcal{S}_2})$. In a more general way, the final aggregated value, $\{\overline{\mathcal{S}_j}\}_{j=1}^{p}$ when the last DO, D_p, merges his encrypted values is computed as

$$\{\overline{\mathcal{S}_j}\}_{j=1}^{p} = \mathsf{HAdd}(\{\overline{\mathcal{S}_1}\}_{j=1}^{p-1}, \overline{\mathcal{S}_p})$$

The training phase can continue until the minimum number of submissions p are made in each class. Each time the scheme reaches this limit, *MO*s decrypt the global model stored in $\{\overline{\mathcal{S}_j}\}_{j=1}^{p}$ to continue with the scheme.

D. DECRYPTION

The more submissions made in a specific class, the more accurate the classification becomes. After the minimum submissions p are made in all classes, *MO*s apply threshold decryption to the global encrypted model $\{\overline{\mathcal{S}_j}\}_{j=1}^{p}$ to retrieve the decrypted global model $\{(\mu_i^*, (\sigma_i^2)^*)\}_{i=1}^{f}$.

Each MO_k would first retrieve the current model from the smart contract and decrypt it using his own private key s_k by performing a partial decryption $\mathsf{PartDec}(s_k, (R, C))$ (as explained in Section II-C). In order to completely decrypt the parameters of the model, each MO_k will need to partially decrypt the model and submit his partial decryption C_{s_k} to the smart contract.[2]

To ensure the submitted partially decrypted value C_{s_k} is correct, the *MO* also computes a proof that will be submitted together with C_{s_k}.

Each *MO* has to prove their decryption computations for all model parameters, as depicted by *ValidateDecryption()*. The function shows the verification for a single parameter, where C is the retrieved partially decrypted ciphertext by other *MO*s until the time of submission.

Algorithm 3 ValidateDecryption

Input: (P_k, s_k), C_{s_k}, (C, R), G
Output: verify
  $C'_{s_k} \leftarrow C - s_k R$
  verify = $(P_k == s_k G$ and $C'_{s_k} == C_{s_k})$

Here also the *MO* computing a proof must first compute a witness.

$$\mathsf{Compute\ Witness}(\underline{s_k}, C, C_{s_k}, P_k, R, \textit{Compile}(\textit{Algorithm 3}))$$

Upon receiving a partially decrypted model, the smart contract will verify the proof. As soon as the decryption is verified, the smart contract replaces the already decrypted model with the new one. The protocol will continue until the model is decrypted.

The parameters retrieved after the decryption are the sums of all local model parameters. The global model is thus the division of these sums by the number of submissions in their respective classes.

E. INCENTIVE DISTRIBUTION

In a distributed architecture, it is vital to encourage participation. This scheme provides an incentive mechanism for this purpose. To measure participation, a metric computes the local model's contribution to the global model in each iteration.

After each decryption phase, the *DO*s who have contributed to that iteration compute a metric using their model parameters and the global model parameters, $\mathcal{F}(\sigma_i, \mu_i, \sigma_i^*, \mu_i^*)$, with the help of Kullback-Leibler [20] or the Bhattacharyya distance [3] that indicates how close their submitted model is to the global model in each for the features.

$$DM_i = \mathcal{F}(\sigma_i, \mu_i, \sigma_i^*, \mu_i^*)$$

where σ_i^* and μ_i^* are, respectively, the variance and the mean for the feature i obtained from the decrypted global model.

Note that the model parameters are all in plain-text format.The value of the computed *DM* quantifies how

[2]Notice that only t number of the MOs need to collaborate in the decryption process since that threshold is enough to compute the secret key of the other $m - t$. For simplicity, the secret key retrieving equations for the possible $m - t$ no-cooperation *MO*s are not mentioned here.

diverged the models are from each other. For a specific model update, the lower the *DM* value for that update, the closer that model is to the global model, and thus the more effective it was in shaping the global model.

When sending their *DM* to the smart contract, each *DO* has to prove that their metric corresponds to the model they have submitted to the scheme in the same iteration. To create the proof, *ValidateEncryption()* will check if the passed encrypted version of the model is the same as the encryption of the plain text input of the model. Then *ValidateFeatureMetric()* will check each of the computations of DM_i for all features in the submitted class.

Algorithm 4 ValidateFeatureMetric

Input: $\{DM_i\}_{i=1}^{f}, \{\mu_i, \sigma_i^2\}_{i=1}^{f}, \{\mu_i^*, (\sigma_i^2)^*\}$
Output: verify
  verify = true
  for $i = \{1, \cdots, f\}$ **do**
    $(DM_i)' \leftarrow \mathcal{F}(\sigma_i, \mu_i, \sigma_i^*, \mu_i^*)$
    if $(DM_i)' \neq DM_i$ **then**
      verify = *false*
      break
    end if
  end for

The overall metric sent to the smart contract by each *DO* is the mean of the DM_i for all of the features existing in the data set. The Algorithm 5 will in the end check if the sent *DM* is correctly computed from the set $\{DM_i\}_{i=1}^{f}$.

Algorithm 5 ValidateMetric

Input: $\{DM_i\}_{i=1}^{f}, DM$
Output: verify $M \leftarrow 0$
  verify = true
  for $i = \{1, \cdots, f\}$ **do**
    $M \leftarrow M + DM_i$
  end for
  $DM' \leftarrow \frac{M}{f}$
  verify = $(DM' == DM)$

After compiling all the programs for the incentive computation proof, the *DO* wanting to submit their *DM* must generate the witness corresponding to their values as below.

$$\text{Compute Witness}(\underline{\mathcal{S}}, \overline{\mathcal{S}}, \underline{\{DM_i\}_{i=1}^{f}}, DM, \mathit{Compile}(\mathit{Algorithm}\ 2, 4, 5))$$

The *DO* wanting to receive its incentive sends a transaction containing the proof, and the public variables in the format of $\mathsf{Tx}(\mathit{Proof}, \overline{\mathcal{S}}, DM)$ to the smart contract. *DO*s will receive an incentive if their proof is correct. A malicious *DO* will receive less incentive if most of the *DO* are acting honestly. This is due to the fact that the parameters of their model will diverge from those of the global model. After the distribution of incentive values, the scheme undergoes either another iteration of training or finishes.

F. DECISION MAKING

The *MO*s will test the model after each decryption to determine whether it has reached a certain performance. In the event that they are satisfied with the model's performance, they will vote to finish the scheme and end the training process; if not, the training process will continue.

G. ENDING

When the training is completed, each *MO* receives his deposit back based on his contribution. The amount of money they get back depends on how many times they contributed to the decryption process and how honest they were. Through the use of deposits, *MO*s are thus incentivized to contribute to decryption. Therefore, the *MO*s using the decrypted models but not contributing cannot claim back their deposits and the computation overhead is fairly distributed among the *MO*s.

IV. IMPLEMENTATION

The purpose of this section is to describe what the implementation consists of and how results affect the scheme. Among the phases described in Section 3, the local model training and aggregation performed by the smart contract are more computationally intensive. As a result, these two are the ones implemented and deployed on the Ethereum public blockchain to prove that Federify is feasible. In the Ethereum blockchain, a block is generated every 12 seconds using the proof of stake consensus protocol.[3] The number of transactions in a block varies as Ethereum uses the notion of gas to adjust the size of each block.[4] Whereas the gas consumption of a transaction varies with respect to its resource consumption. By deploying the smart contracts implemented in Solidity and testing the gas consumption of carrying out the scheme, we prove Federify's practicality.

The implementation consists of two parts and is available for the public on GitHub.[5] Each *DO* gathers data and trains a model corresponding to the scheme, then updates the global model with the trained local model. Each *MO* registers in the smart contract with a public key and later takes part in decryption. Local computations done by the *MO* and *DO* are only trusted when verified via zkSNARK proofs. In this implementation, ZoKrates toolset it used for implementing the zkSNARK circuits of Federify [8]. All the verifications needed for the scheme in ZoKrates are as described in Algorithms 1, 2, 3, 4, and 5. Among these, only Algorithms 1, 2, and 3 are implemented since these

[3] Ethereum started with a proof of work consensus mechanism, but in order to reduce the energy consumption switched to proof of stake on September 2022.

[4] At the time of publishing this paper, Ethereum roughly generates 12 transactions per second.

[5] https://github.com/qatkk/FederatedNaiveBayesZKP

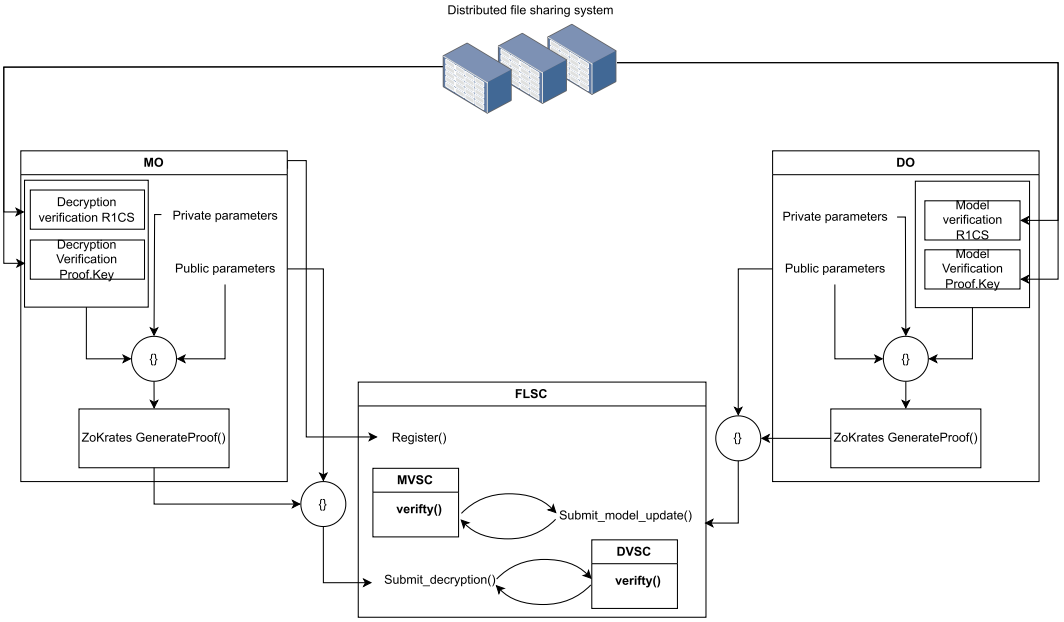

FIGURE 4. Overall interaction between different implemented entities.

are more computationally heavy and crucial when testing feasibility.

The *DO* and *MO* interact with the main smart contract (FLSC) which coordinates the scheme and handles on-chain aggregation and decryption. Model verification smart contract (MVSC) as explained in Section III-C, the decryption verification smart contract (DVSC) from Section III-D, and the incentive verification contract (IVSC) as Section III-E are the three smart contracts used for the on-chain verification process. The uses of these smart contracts and the connection between them are depicted in Fig. 3. As shown, FLSC includes and calls all other verification contracts, but the only ones corresponding to the implemented verifications, MVSC and DVSC, are implemented and described here.

A. DATA OWNER (DO)

As described in Section III-C the *DO* gathers data for a specific class that is to be trained. When the data reaches β, the program computes the model parameters. After training the model parameters *DO* then retrieves the public key from FLSC for encryption. The implementation uses the Baby Jub Jub [39] curve parameters to encrypt the model.

The *DO* must then continue to create a proof using the computed model parameters. Input parameters to ZoKrates need to be prepared before being given as input for witness generation. As part of this preparation, the *DO* must preprocess the data and model parameters due to multiple computation shortfalls in ZoKrates. The first drawback is that ZoKrates cannot handle floating points and negative values. It is necessary to add a specific positive predefined value to all the data points before computing model parameters to remove negative and outlier points. Additionally, ZoKrates' computation requires a fixed-size input for a fixed-compiled program. We solve this problem using β for the data. A model parameter is then multiplied by the accuracy parameter in the form of 10^d, being d the accuracy parameter, in order to have better precision when computing integers.

The data and local model parameters are passed as private parameters, while accuracy, encryption, and public parameters are passed as public to the ZoKrates for witness generation. The only verification applied by *DO* is described in Algorithms 1 and 2. This part is implemented in Javascript to interact both with the smart contract and ZoKrates.

As shown in Fig. 3, the *DO* sends its proof for the model verification to the MVSC. The Verifytx() function of the MVFC is responsible for verifying the proof. Whenever a *DO* sends an update to the UpdateModel() function of FLSC, this function calls Verifytx() from the MVSC passing the proof along with the public parameters as input. If the proof is verified, the UpdateModel() will then homomorphically add the received parameter update to the existing model parameters of the class and increment the number of submissions sent to that class, as described in Section III-C.

B. MODEL OWNER (MO)

The *MO* only interacts with the scheme in two phases through the scheme sequence. At the registration phase, each MO_k registers in the FLSC by sending P_k, deposit, and funds using an Ethereum account. The FLSC contract then saves the address of that specific MO_k along with their deposit and adds the received P_k to P. At the end of the registration phase, FLSC has the P in the format of $P = \Sigma_{i=1}^{m} P_k$ for later use in encryption. The second interaction is when the

TABLE 2. Proof.key and R1CS file size with respect to batch size and number of the features for the model verification process.

	R1CS					Proof.Key				
f	β									
|	100	500	1000	1500	2000	100	500	1000	1500	2000
5	241 MB	1.12 GB	2.22 GB	3.32 GB	4.42 GB	229.8 MB	996.6 MB	1.98 GB	3.23 GB	3.94 GB
8	385.4 MB	1.79 GB	3.55 GB	5.31 GB	7.07 GB	394.4 MB	1.7 GB	3.38 GB	4.52 GB	6.73 GB
10	481.6 MB	2.24 GB	4.44 GB	6.64 GB	8.84 GB	459.4 MB	1.99 GB	3.95 GB	6.45 GB	7.88 GB
13	626 MB	2.91 GB	5.77 GB	8.64 GB	11.5 GB	556.8 MB	2.43 GB	4.81 GB	7.74 GB	9.6 GB
15	717 MB	3.2 GB	6.3 GB	9.4 GB	13.29 GB	612 MB	3.1 GB	6.1 GB	8.1 GB	12.91 GB

TABLE 3. R1CS and Proof.Key size for the whole scheme with respect to batch size and number of the features for the partial decryption process.

f	R1CS	Proof.Key
5	22.8 MB	23.9 MB
8	36.1 MB	41.1 MB
10	45 MB	47.1 MB
13	58.3 MB	55.9 MB
15	67.2 MB	61.8 MB

MO takes part in decryption which consists of multiple steps. First, the *MO* retrieves the partially decrypted global model from the specified function of the FLSC for a specific class of the model.

At this point, the *MO* will then decrypt the retrieved partially decrypted model. *MO* applies verification for decryption as in Algorithm 3. In the verification process, the local public key, the partially decrypted model, the random point, and the decrypted global model until that decryption round are passed as public input. Where the only private input passed is the secret key corresponding to the local public key. In the end, the *MO* interacts with FLSC for decryption verification through the same Ethereum account registered on FLSC. The *MO* sends its decrypted model parameters with proof to the DecryptModel() to submit the new partially decrypted model. This function of the FLSC then calls Verifytx() from the DVSC passing the proof along with the public parameters as input. As a result of verifying the received decrypted model, FLSC then replaces the partially decrypted model with the newly submitted one and increments the number of participations for that specific *MO*. The implementation for the *MO* is in Javascript to interact with both ZoKrates and the blockchain.

V. SCHEME ANALYSIS

Using the implementation, we run experiments to test two valuable resources in this scheme. Use of zkSNARKs is computationally heavy which results in more gas cost on the blockchain and more storage usage when storing the output files. A diagram illustrating how the various implementation parts interact can be found in Fig 4. Furthermore, we also review the security requirements that have been outlined in Section II-B.

A. STORAGE REQUIREMENTS

The storage required for proof generation and verification changes with respect to the scheme parameters and the program that an entity is verifying. The more computationally heavy a program is the more its constraints will be and the bigger the size of its setup files. The R1CS is the main file needed for ComputeWitness() and Setup(). An R1CS is generated before the setup phase and is stored off the blockchain on a distributed file-sharing system for each of the verifications.

The files used directly or indirectly for zkSNARK proof generation are the R1CS and the Proof.Key. The size of these files changes with respect to scheme parameters. The computations done in ZoKrates are converted into constraints after compilation.

Table 2 shows the file size of Proof.Key and R1CS for different values of the number of features (f) and batch sizes (β) for a *DO*.

Decryption verification depends only on f, where Proof.Key and R1CS size for this verification can be found in Table 3 with respect to this parameter. The size of Verify.Key and Proof files are not affected by changes in scheme parameters, with their sizes remaining 22KB for the model verification and 15KB and 7KB for the decryption verification. According to Table 2, in order to generate proof for verifying the trained local model on an average computer, the *DO* only requires 6.62 GB of storage when training a model with 15 different features and 500 batches. As mentioned in Table 3, an *MO* requires only 128 MB of storage in order to verify its decryption for the global model for the mentioned training configuration.

These results are obtained by executing ZoKrates with Groth16 for proof generation and the alt bn128 curve for the key generation and bindings. Proof generation and setup phase for the results reported in this section were obtained by running ZoKrates commands on a system with 16 GB

TABLE 4. Gas cost of the implementation of the scheme with respect to the number of features.

f	FLSC Deployment	Model update for a new class of data	Model update for a submitted class	Partial decryption
5	7879358	1772714	939021	796472
8	8659076	2448946	1302010	1073924
10	9177941	2899821	1545249	1259064
13	9956539	3575964	1909266	1537070
15	10551459	4027268	2152839	1722832

of RAM.[6] The only training configuration exceeding this memory bound is for 15 features and a batch size of 1500 and 2000. Reported results for the mentioned configuration were carried out by a system with 64GB of memory.

B. GAS COST

The second valuable resource is the cost of interacting with the blockchain. When interacting with a smart contract, each transaction costs a certain amount of gas based on the computations performed to execute a function. In order to test the utility of our design, the Goerli test network[7] is used to deploy FLSC and measure the costs of interacting with it. Each *DO* undergoes costs when updating the global model as part of the experiment. Such costs directly influence participation and model quality.

These costs are mainly caused when carrying out the zkSNARK verification on the blockchain. As described in the original paper of ZoKrates [8], the on-chain verification cost of a ZoKrates program only depends on the number of public inputs passed to the program. That means that the complexity of the ZoKrates program doesn't affect the gas costs on the blockchain and is the same for the same number of public inputs to the verification smart contract. In the mentioned public and private set of inputs to ZoKrates, every point on the curve for every encrypted model parameter requires two inputs, the (x, y) coordinate of the point on the curve. Verification costs on the blockchain can be reduced by passing only the x value of each point as public input. This reduces the cost but imposes a new security risk if the verifier passes wrong coordinates. To prevent such actions, the program accepts the y value along with each x and checks if the point is a valid point on the curve.

The gas costs corresponding to model update, contract deployment, and partial decryption can be found in Table 4. Note that the gas cost only changes with respect to f and is independent of β. Among these costs, the contract moderator[8] pays for the contract deployment, *DOs* pay for the model update, and the *MO* pays for each decryption verification they participate in.

The reported values are far from the current limit of the Ethereum network, which is 30,000,000 units, and show that the proposed plan is feasible.

[6]zkSNARKs can only be used when the system has sufficient memory, since when computing heavy programs are run, the number of constraints increases, therefore the amount of memory required increases.

[7]https://goerli.net

[8]This entity can be anyone of the *MOs*.

C. SECURITY DISCUSSION

We have seen in Section II-B an outline of the main problems found in FL frameworks. In our proposal, given our use of a public blockchain and smart contracts, we do not need to place trust in the entity aggregating the local models, something that is usually assumed in FL. This being the major characteristic of our proposal would not suffice by itself without proper mechanism to prevent security and privacy problems, but will bring availability, accessibility and fault tolerance with respect to network failures since a public blockchain is maintained among distributed nodes or miners. We consider that even without the use of a central aggregation server, most of the server-side attacks considered in the literature can also be possible. The aggregation is performed in a smart contract over a public blockchain, which means that the information and computations used in the aggregation are public. In some sense, the lack of a central server requires stronger security mechanisms, something to be expected when we do not require placing trust in such a process.

Security problems in FL are commonly related to clients providing malicious data, either using bad data to train their local model (data poisoning) or directly sending maliciously crafted local model parameters (model poisoning). In order to minimize these problems, our proposal relies on the use of zkSNARKs, which could allow verifying the local model parameters from the local data. Model poisoning attacks are thus prevented, but it is noteworthy to consider that data poisoning is still possible, as it is in most FL frameworks. In general, having control over the data collected by each client goes against the spirit of FL. In our case, the use of the zkSNARKs can also provide some accountability. It makes it possible to perform a posteriori analysis of a given client data. That is, verify, only if there is a problem, the data used by a given client.

From a privacy perspective, we have seen that our proposal relies on homomorphic encryption to preserve the confidentiality of the local models and the aggregation process. The threshold scheme ensures that even data owners are not able to decrypt local models. The only potential inference can be done by a client from the partial global model. This kind of client-side privacy attacks is limited, specially as the number of clients increases [25].

It is noteworthy to highlight that the scheme provides transparency of both the learning process and the incentive mechanism by delegating coordination into a smart contract in a public blockchain. In addition to the use of a

public blockchain, using a threshold encryption mechanism, and the distributed structure of the scheme, results in Federify being resilient to the single point of failure attacks.

In the following sections, we will provide a deeper discussion of these issues comparing them to related work.

VI. RELATED WORKS AND COMPARISON

In this section, an in-depth description and comparison of the existing literature is provided. Many methods for federated learning have been proposed, but their designs are limited by the following issues: (i) security threats to the privacy of local data, (ii) verifiability of the results, and (iii) lack of reward mechanisms.

Our first discussion will focus on the different hiding protocols they may use to protect local data. Afterward, we will discuss the verifiability of the obtained results from two different perspectives. As part of a federated learning structure, the correctness of the global model is dependent on the correct aggregation of the local models and the correct computation of the local models. We present a discussion of different aggregation methods in the field and then discuss how different works examine the correctness of the local models within the analysis.

A. HIDING PROTOCOL

Sending raw model parameters may reveal critical information about the data that was used to train those models [1]. Therefore, existing FL schemes conceal or mask their local models, and several approaches have been followed with this goal. In [22], [26], and [35], model parameters are hidden to some extent by adding random noise, in order to provide differential privacy. Based on the added noise's randomness and value, this method provides varying degrees of security to each scheme. To resolve attacks on the randomness of the noise, Rückel et al. [33] and Zhang et al. [44] verify the randomness of the added noise by using verifiable random functions used in Algorand [7].

Several works, including our proposal, have used encryption to secure model parameters. First, [29] employs a multi-party computation encryption scheme (MPC) to encrypt model parameters. Following a round of model training, these parameters are sent to a third-party server that decrypts and aggregates the model updates in a trusted execution environment (TEE). The TEE then provides participants with the results of aggregation as well as its verification. The use of TEEs introduces a single point of failure and trust into the design, which can be inconvenient.

Toyoda et al. use encryption in [37] and [38] alongside incentive mechanisms to ensure better behavior of the participants. In both of these schemes, a task publisher is responsible for sending out encryption keys to participants through private channels. This can risk the privacy of the scheme in case of eavesdropping. A similar study was conducted by Feng et al. [9] where a conductor sends participants the keys for secret sharing. With this scheme's homomorphic masking protocol, there is no need for decrypting each model update, and aggregation is done by a chosen participant at the end of each training round. A potential disadvantage of this work is that the keys are distributed over a private channel, and it will require an extra setup step in the event that participants drop out.

Other works such as [16] and [41] also use encryption as a hiding protocol for model updates which is also done by a central aggregator server.

Federify only requires an initial setup phase for MPC encryption, allowing participants to drop out at no cost to communications or computation. With the help of cumulative decryption performed at the end of every phase, *MO* privacy is also preserved through homomorphic aggregation. Also, our scheme guarantees no faults in the aggregation process due to the automatic aggregation method done by the smart contract.

B. AGGREGATION METHOD

When talking about aggregation in an FL design, the first idea that comes to mind is to rely on a third-party server. In [29], the trust is removed from the aggregator with the help of a TEE (Trusted Execution Environment) to safely decrypt model updates and compute the aggregated global model. In order to provide fairness to the aggregation process, [41] introduces a verification method. In this work, a cloud service conducts aggregation and then presents the result along with a verification to prove that the aggregation is fair and secure.

To enhance the security of the previous work, [16] adds a key exchange technology, hence improving the problem of a trusted third party. Other works such as [10], [15], and [9] also use an aggregation server and remove trust from the aggregator by using different hiding methods as described in the previous section and conduct fair aggregation by verifying the aggregated results. As an example, [16] leverages homomorphic hash functions and pseudorandom technologies to provide proof for aggregation. Nevertheless, these works rely on a third party for aggregation, which can expose them to single-point-of-failure attacks.

The aggregator server can be removed by distributing the aggregation task. In [37] and [38], multiple entities take the role of an aggregator. A group of randomly chosen entities apply aggregation to the collected updates from each round. In their first work [37], Toyoda et al. ensure the honest behavior of the entities with the help of competitive model updates, where only desired updates are accepted. They implemented an incentive mechanism through which clients would only profit by acting as the scheme demands. Although the aggregation is distributed among participants, the fairness of the aggregation is only guaranteed by an incentive mechanism. Further, they enhanced their previous work in [38] by implementing aggregation verification with the help of Lagrange interpolation.

A different method of distributing the aggregation is to integrate FL with blockchain, where participants are miners as in [22] and [35]. In these works, each block contains all the updates for a training round and the aggregated results. In Biscotti, proof of federation (PoF) is proposed as a blockchain consensus protocol to withstand attacks on the aggregation of FL structures. In this scheme, verifiable random functions and hashing algorithms determine the roles of entities as aggregators of the structure. In [22], *DO*s are the nodes of the blockchain in the sense that the nodes do the verification. Each node verifies the quality of uploaded local models to the blockchain by using their local data as a test set and aggregating the models. When receiving a block, each miner can verify if the aggregated result is true based on the updates in the block. These works verify the aggregation and distribute the trust bringing fairness and correctness to the aggregation process in an FL design but need their own blockchain and cryptocurrencies which can decrease their usability.

To increase the usability and have a trustless distributed aggregation process, multiple works such as [17], [31], [33], and [44] use smart contracts. Since functions in a smart contract run automatically and publicly, using them for aggregation removes the need for trusting a server and verifying the aggregation process. Since the computations are done publicly, these works need to use hiding methods to preserve the privacy of the individual updates in each training round.

C. A COMPARISON OF FEDERIFY WITH THE STATE OF THE ART

In this section, we describe more in detail the works that are closely related to the proposed scheme and compare Federify to the existing state of the art. In [44], the design provides fairness through verifiable aggregation and preserves data privacy by adding random noise to the local model parameters. This scheme takes advantage of a verifiable random function (VRF) to implement fully verifiable differential privacy. In this scheme, at each training phase, a new transaction is created and used as the seed for the VRF function. After VRF creates random values, each client would retrieve the random value, create their own random parameter, and use it to hide their local model parameters. In such a scheme, all random values added to the local parameters will be verified on the blockchain with the help of zkSNARKS. Although this scheme verifies the randomness of the local model parameters and preserves local data privacy, the computation of the gradients is unverified and can put the global model at risk of model poisoning.

Heiss et al. In [17] present a design where all data owners' model computations are verified with zkSNARK. The clients compute a model locally and send their gradients to the smart contract for verification and aggregation. Although the design prevents model poisoning, it cannot guarantee the privacy of the data since all gradients are sent to the blockchain without hiding them.

Ruckel et al. presented a scheme using Blockchain and zkSNARKs in [33] to prevent the model and data poisoning attack. To validate the model owners' computation and process, a simple linear regression circuit is implemented using zkSNAKRs. Each data owner then submits their local model with added noise to the smart contract for aggregation. This scheme takes advantage of a public test data set to check the performance of the submitted models. Although in this work the computation for local model parameters is verified, the added noise is still not verified and won't fully protect the privacy of data owners' data.

A majority of the existing literature carries out aggregation through a third-party server, which can be verified, whereas Federify uses smart contracts to automatically and transparently do the model aggregation, eliminating the need for verification. In comparison to the few works mentioned in this section providing verifiability for the model parameters, Federify's advantage is through the use of homomorphic encryption which provides full privacy for model parameters yet allows the continued training of the global model. As a result, compared to the existing state of the art, Federify ensures full privacy of the local data and model parameters while demonstrating feasibility and verifiability through the provided proof-of-concept.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have designed, implemented, and evaluated Federify, a distributed federated learning framework based on blockchain and cryptography. By using a blockchain, our framework is able to remove the need for a central server that coordinates the learning process. Furthermore, the usage of a public blockchain not only introduces transparency to the learning process and the incentive distribution mechanism but also reinforces the security of the system. The usage of encryption ensures the confidentiality of local models, and the incorporation of zkSNARKs provides security guarantees to the scheme without compromising privacy.

Moreover, our proof-of-concept implementation successfully demonstrated the practical viability of the proposed framework. The proposed smart contracts are deployable on the Ethereum blockchain and the size of the zkSNARKs circuits and keys is manageable even for an average laptop (a 16GB RAM laptop is able to run our proof-of-concept for models up to 15 features with a batch size of 1000).

Moving forward, future research endeavors will focus on enhancing the capabilities of our framework by incorporating other (more complex) learning models and ensuring the scalability of the scheme (for instance, by studying its viability over layer two solutions or sidechains). Additionally, we will explore diverse incentive functions to gauge their impact on the computational cost of running the scheme and the learned models. These extensions will facilitate

a deeper understanding of the potential and limitations of our proposed framework, ultimately contributing to the advancement of federated learning in a distributed and secure manner.

REFERENCES

[1] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 5, pp. 1333–1345, May 2018.

[2] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," in *Proc. 23rd Int. Conf. Artif. Intell. Statist.*, Jun. 2020, pp. 2938–2948.

[3] A. Bhattacharyya, "On a measure of divergence between two multinomial populations," *Sankhyā, Indian J. Statist.*, vol. 7, no. 4, pp. 401–406, Jul. 1946.

[4] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in *Proc. 29th Int. Conf. Mach. Learn. (ICML)*, Jun. 2012, pp. 1467–1474.

[5] P. Blanchard, E. M. El Mhamdi, R. Guerraoui, and J. Stainer, "Machine learning with adversaries: Byzantine tolerant gradient descent," in *Advances in Neural Information Processing Systems*, vol. 30. Red Hook, NY, USA: Curran Associates, 2017.

[6] V. Buterin, "Ethereum white paper: A next generation smart contract & decentralized application platform," Ethereum Found., Tech. Rep., 2013.

[7] J. Chen and S. Micali, "Algorand," 2016, *arXiv:1607.01341*.

[8] J. Eberhardt and S. Tai, "ZoKrates–Scalable privacy-preserving off-chain computations," in *Proc. IEEE Int. Conf. Internet Things (iThings), IEEE Green Comput. Commun. (GreenCom), IEEE Cyber, Phys. Social Comput. (CPSCom), IEEE Smart Data (SmartData)*, Jul. 2018, pp. 1084–1091.

[9] C. Fang, Y. Guo, J. Ma, H. Xie, and Y. Wang, "A privacy-preserving and verifiable federated learning method based on blockchain," *Comput. Commun.*, vol. 186, pp. 1–11, Mar. 2022.

[10] A. Fu, X. Zhang, N. Xiong, Y. Gao, H. Wang, and J. Zhang, "VFL: A verifiable federated learning with privacy-preserving for big data in industrial IoT," *IEEE Trans. Ind. Informat.*, vol. 18, no. 5, pp. 3316–3326, May 2022.

[11] C. Fung, C. J. M. Yoon, and I. Beschastnikh, "Mitigating Sybils in federated learning poisoning," Jul. 2020, *arXiv:1808.04866*.

[12] K. Ganju, Q. Wang, W. Yang, C. A. Gunter, and N. Borisov, "Property inference attacks on fully connected neural networks using permutation invariant representations," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 619–633.

[13] J. Geiping, H. Bauermeister, H. Dröge, and M. Moeller, "Inverting gradients—How easy is it to break privacy in federated learning?" in *Advances in Neural Information Processing Systems*, vol. 33. Red Hook, NY, USA: Curran Associates, 2020, pp. 16937–16947.

[14] Y. Gu, Y. Bai, and S. Xu, "CS-MIA: Membership inference attack based on prediction confidence series in federated learning," *J. Inf. Secur. Appl.*, vol. 67, Jun. 2022, Art. no. 103201.

[15] X. Guo, Z. Liu, J. Li, J. Gao, B. Hou, C. Dong, and T. Baker, "VeriFL: Communication-efficient and fast verifiable aggregation for federated learning," *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 1736–1751, 2021.

[16] G. Han, T. Zhang, Y. Zhang, G. Xu, J. Sun, and J. Cao, "Verifiable and privacy preserving federated learning without fully trusted centers," *J. Ambient Intell. Humanized Comput.*, vol. 13, no. 3, pp. 1431–1441, Mar. 2022.

[17] J. Heiss, E. Grünewald, S. Tai, N. Haimerl, and S. Schulte, "Advancing blockchain-based federated learning through verifiable off-chain computations," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Aug. 2022, pp. 194–201.

[18] B. Hitaj, G. Ateniese, and F. Perez-Cruz, "Deep models under the GAN: Information leakage from collaborative deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 603–618.

[19] N. M. Jebreel, J. Domingo-Ferrer, A. Blanco-Justicia, and D. Sánchez, "Enhanced security and privacy via fragmented federated learning," *IEEE Trans. Neural Netw. Learn. Syst.*, pp. 1–15, Oct. 2022, doi: 10.1109/TNNLS.2022.3212627.

[20] J. M. Joyce, "Kullback–Leibler divergence," in *International Encyclopedia of Statistical Science*. Berlin, Germany: Springer, 2011, pp. 720–722. [Online]. Available: https://doi.org/10.1007/978-3-642-04898-2_327

[21] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: Challenges, methods, and future directions," *IEEE Signal Process. Mag.*, vol. 37, no. 3, pp. 50–60, May 2020.

[22] C. Ma, J. Li, L. Shi, M. Ding, T. Wang, Z. Han, and H. V. Poor, "When federated learning meets blockchain: A new distributed learning paradigm," *IEEE Comput. Intell. Mag.*, vol. 17, no. 3, pp. 26–33, Aug. 2022.

[23] X. Ma, X. Sun, Y. Wu, Z. Liu, X. Chen, and C. Dong, "Differentially private Byzantine-robust federated learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3690–3701, Dec. 2022.

[24] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. Artif. Intell. Statist.*, 2017, pp. 1273–1282.

[25] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov, "Exploiting unintended feature leakage in collaborative learning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 691–706.

[26] A. Mondal, H. Virk, and D. Gupta, "BEAS: Blockchain enabled asynchronous & secure federated machine learning," 2022, *arXiv:2202.02817*.

[27] S. Nakamoto and A. Bitcoin. (Apr. 2008). *A Peer-to-Peer Electronic Cash System*. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[28] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 739–753.

[29] J. Passerat-Palmbach, T. Farnan, R. Miller, M. S. Gross, H. Leigh Flannery, and B. Gleim, "A blockchain-orchestrated federated learning architecture for healthcare consortia," 2019, *arXiv:1910.12603*.

[30] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in *Advances in Cryptology—EUROCRYPT*, vol. 547. Berlin, Germany: Springer, 1991. [Online]. Available: https://doi.org/10.1007/3-540-46416-6_47

[31] P. Ramanan and K. Nakayama, "BAFFLE: Blockchain based aggregator free federated learning," in *Proc. IEEE Int. Conf. Blockchain (Blockchain)*, Nov. 2020, pp. 72–81.

[32] H. Ren, J. Deng, and X. Xie, "GRNN: Generative regression neural network—A data leakage attack for federated learning," *ACM Trans. Intell. Syst. Technol.*, vol. 13, no. 4, pp. 65:1–65:24, May 2022.

[33] T. Rückel, J. Sedlmeir, and P. Hofmann, "Fairness, integrity, and privacy in a scalable blockchain-based federated learning system," *Comput. Netw.*, vol. 202, Jan. 2022, Art. no. 108621.

[34] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 459–474.

[35] M. Shayan, C. Fung, C. J. M. Yoon, and I. Beschastnikh, "Biscotti: A blockchain system for private and secure federated learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1513–1525, Jul. 2021.

[36] N. Szabo, "Smart contracts: Building blocks for digital markets," *EXTROPY, J. Transhumanist Thought*, vol. 18, no. 2, p. 28, 1996.

[37] K. Toyoda and A. N. Zhang, "Mechanism design for an incentive-aware blockchain-enabled federated learning platform," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2019, pp. 395–403.

[38] K. Toyoda, J. Zhao, A. N. S. Zhang, and P. T. Mathiopoulos, "Blockchain-enabled federated learning with mechanism design," *IEEE Access*, vol. 8, pp. 219744–219756, 2020.

[39] B. WhiteHat, J. Baylina, and M. Bellés, "Baby Jubjub elliptic curve," Ethereum Improvement Proposal, Ethereum Found., Tech. Rep. EIP-2494, 2020, vol. 29.

[40] Z. Wu, Q. Ling, T. Chen, and G. B. Giannakis, "Federated variance-reduced stochastic gradient descent with robustness to Byzantine attacks," *IEEE Trans. Signal Process.*, vol. 68, pp. 4583–4596, 2020.

[41] G. Xu, H. Li, S. Liu, K. Yang, and X. Lin, "VerifyNet: Secure and verifiable federated learning," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 911–926, 2020.

[42] H. Yang, Y. Wang, and B. Li, "Individual property inference over collaborative learning in deep feature space," in *Proc. IEEE Int. Conf. Multimedia Expo (ICME)*, Jul. 2022, pp. 1–6.

[43] J. Zhang, J. Zhang, J. Chen, and S. Yu, "GAN enhanced membership inference: A passive local attack in federated learning," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2020, pp. 1–6.

[44] Y. Zhang, Y. Tang, Z. Zhang, M. Li, Z. Li, S. Khan, H. Chen, and G. Cheng, "Blockchain-based practical and privacy-preserving federated learning with verifiable fairness," *Mathematics*, vol. 11, no. 5, p. 1091, Feb. 2023.

[45] B. Zhao, K. R. Mopuri, and H. Bilen, "iDLG: Improved deep leakage from gradients," Jan. 2020, *arXiv:2001.02610*.

[46] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 14774–14784.

GHAZALEH KESHAVARZKALHORI received the bachelor's degree in electrical engineering major in communication engineering from the University of Tehran. Her research conducted primarily in the fields of blockchain technology and privacy-preserving protocols with the Secure Communication Laboratory (SCL), University of Tehran. After graduating, she has been a Researcher with the SENDA Research Group, Autonomous University of Barcelona. She is also researching on the combination of federated learning, blockchain, and privacy.

CRISTINA PÉREZ-SOLÀ received the first Ph.D. degree in computer science from the Autonomous University of Barcelona (UAB) and the second Ph.D. degree in engineering science (electrical engineering) from Katholieke Universiteit Leuven (KULeuven). She is currently an Associate Professor with the Department of Information and Communications Engineering, UAB. Since 2014, her main research interests include Bitcoin and blockchain-based cryptocurrencies, mainly with respect to security and privacy. She is also interested in machine learning, cryptography, and graph theory.

GUILLERMO NAVARRO-ARRIBAS is currently an Associate Professor with the Department of Information and Communications Engineering, Autonomous University of Barcelona (UAB), where he is also a member of the SENDA Research Group. His main research interests include data privacy, privacy enhancing technologies, and blockchain technology.

JORDI HERRERA-JOANCOMARTÍ received the degree in mathematics from the Autonomous University of Barcelona (UAB), in 1994, and the Ph.D. degree from Universitat Politècnica de Catalunya, in 2000. In 2000, he joined Universitat Oberta de Catalunya and founded the KISON Research Group. He is currently an Associate Professor with the Department of Information and Communications Engineering, UAB, where he is also a member of the SENDA Research Group. His research interests include privacy, computer security, and cryptography, with special dedication to blockchain technology and cryptocurrencies.

HABIB YAJAM received the Ph.D. degree in secure communication from the University of Tehran, Iran, and the M.Sc. degree in telecommunication engineering-cryptography from the Sharif University of Technology. His research, conducted primarily at the Secure Communication Laboratory, has been extensive in the fields of blockchain technology, consensus algorithms, privacy-preserving protocols, and anonymous communications. He has a distinguished background in research, with a focus on developing innovative solutions to enhance security and privacy in digital communications.