arXiv:2002.08423v1 [cs.LG] 19 Feb 2020

PRIVACYFL: A SIMULATOR FOR PRIVACY-PRESERVING AND SECURE FEDERATED LEARNING

Vaikkunth Mugunthan
CSAIL
Massachusetts Institute of Technology
vaik@mit.edu

Anton Peraire-Bueno
CSAIL
Massachusetts Institute of Technology
aperaire@mit.edu

Lalana Kagal
CSAIL
Massachusetts Institute of Technology
lkagal@mit.edu

ABSTRACT

Federated learning is a technique that enables distributed clients to collaboratively learn a shared machine learning model while keeping their training data localized. This reduces data privacy risks, however, privacy concerns still exist since it is possible to leak information about the training dataset from the trained model's weights or parameters. Therefore, it is important to develop federated learning algorithms that train highly accurate models in a privacy-preserving manner. Setting up a federated learning environment, especially with security and privacy guarantees, is a time-consuming process with numerous configurations and parameters that can be manipulated. In order to help clients ensure that collaboration is feasible and to check that it improves their model accuracy, a real-world simulator for privacy-preserving and secure federated learning is required.

In this paper, we introduce PrivacyFL, which is an extensible, easily configurable and scalable simulator for federated learning environments. Its key features include latency simulation, robustness to client departure/failure, support for both centralized (with one or more servers) and decentralized (serverless) learning, and configurable privacy and security mechanisms based on differential privacy and secure multiparty computation (MPC).

In this paper, we motivate our research, describe the architecture of the simulator and associated protocols, and discuss its evaluation in numerous scenarios that highlight its wide range of functionality and its advantages. Our paper addresses a significant real-world problem: checking the feasibility of participating in a federated learning environment under a variety of circumstances. It also has a strong practical impact because organizations such as hospitals, banks, and research institutes, which have large amounts of sensitive data and would like to collaborate, would greatly benefit from having a system that enables them to do so in a privacy-preserving and secure manner.

1 Introduction

Federated learning has become an important area of research as it provides an optimal solution to train a collaborative model when data is dispersed across different organizations. Its primary advantage is that it allows statistical models to be trained over remote entities while keeping data localized. For example, individual hospitals can collaboratively train a machine learning model in order to improve their predictive accuracy for a given task. However, most clients are unwilling to share their data or inadvertently leak information due to internal privacy policies or regulation. Therefore, it is important to develop federated learning frameworks that train highly accurate models in a privacy-preserving manner.

Let us assume that there are several hospitals that want to develop a shared machine learning model. Figure 1 depicts the basic workflow of a federated learning environment. First, each hospital trains a local model, ΔW_i, rather than sending raw data to a centralized server. Hospitals act as remote clients and communicate with a central server at regular

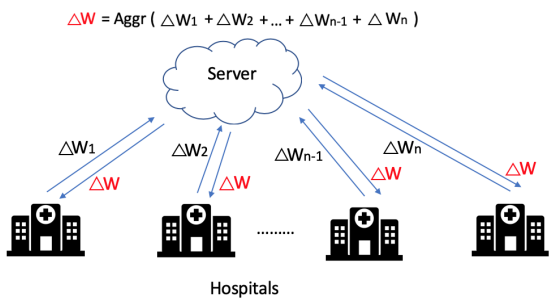

Figure 1: Federated Learning Workflow

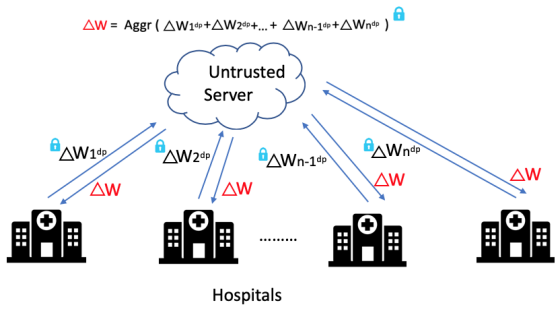

Figure 2: Secure and Privacy-Preserving Federated Learning

intervals to learn a global model. For every iteration, the hospitals send their local model to the server. The server computes and sends back the global model to individual hospitals. This process repeats until convergence occurs or some sort of stopping criterion is achieved. There are two main drawbacks of the protocol implemented in Figure 1: (i) communication is not secure, and (ii) collusion of $n-1$ clients in a n client system reveals the model of the remaining client. The use of MPC helps with the first issue and the second one requires a privacy-preserving mechanism such as differential privacy. This is illustrated in Figure 2. The differentially private inputs, ΔW_i^{dp}, are encrypted and sent to the server. The server aggregates the inputs received from the clients and sends them back to the client.

Some of the core challenges of federated learning include:

- **Communication cost:** This is a huge bottleneck as it increases as the number of iterations increases and becomes an issue as the number of clients increases. Using the simulation results, a system designer can decide whether it is feasible to participate in the federated learning environment.
- **Privacy and security guarantees:** In a federated learning environment, a malicious client can compromise the privacy of individuals in the training data by inferring details of a training dataset from the model's weights [22, 18]. These privacy issues can be addressed through the use of differential privacy and MPC. Our simulator allows the system designer to evaluate the performance with differential privacy and security mechanisms.

While designing a federated learning environment, it is important to understand the trade-offs between privacy, efficiency, and accuracy. System designers should be able to choose different parameters and compare multiple scenarios in order to find the right trade-offs for their environment. Simulators are essential for federated learning environments for the following reasons:

- To evaluate accuracy: It should be possible to simulate federated models and compare their accuracy with local models.
- To evaluate total time taken: Communication between distant clients can become expensive. Simulations are useful for evaluating if client-client and client-server communications are beneficial.
- To evaluate approximate bounds on convergence and time take for convergence.
- To simulate real-time dropouts: Clients in a federated environment may drop out at any time.

Our simulator, PrivacyFL, is designed to specifically address these issues. It is extremely efficient in terms of speed and execution time as client and server operations happen in a multi-threaded manner. It provides numerous differentially

private mechanisms including the Laplacian mechanism [8], the Gaussian mechanism [8], and the Staircase mechanism [10]. System designers can choose from any of these mechanisms to ensure that the learning is privacy-preserving. Our system also provides the option for clients to use a Diffie-Hellman key exchange for inbuilt security. The simulator can also easily support the addition of new privacy-preserving algorithms. It is composed of generalized classes and functions that make it easy for developers to add/modify new algorithms and protocols. Along with simulating federated environments that use one or more central servers, our simulator is also able to simulate a completely decentralized environment, i.e., a federated learning system without a centralized server. In the real world, environments are dynamic with clients failing, dropping off, or being added at run-time. Being able to model this is an important requirement of a simulator for federated learning and PrivacyFL is able to handle such dynamic cases. It is able to simulate computation and communication times to evaluate how long a client would have to wait for each step of the protocol to complete in a real world setting. The simulator also has multi-threading capabilities and uses PySpark to speed up computation so that it can more easily be deployed as a production system.

The remainder of this paper is organized as follows. In Section 2, we discuss existing frameworks in the field of privacy-preserving secure federated learning. Section 3 describes background work and foundational concepts. We provide a detailed explanation of our architecture, classes, and life-cycle in Section 4. In Section 5, we discuss privacy-preserving and secure aggregation algorithms. Experiments and evaluations under different scenarios are described in Section 6, and Section 7 discusses the reproducibility of our experiments. We conclude with a summary and a discussion of our future work in Section 8.

2 Related Work

Federated learning is an approach to train machine learning models on distributed data. The federated averaging algorithm proposed by [15] follows a synchronous training approach in which each client trains a local model and sends it to a centralized server, which computes the average of all models. The server sends the average model back to the clients. Numerous approaches have been proposed to enhance privacy and security in a federated learning environment. [16] and [9] applied differential privacy to provide the necessary privacy guarantees whereas [5] proposed a secure aggregation protocol for privacy-preserving machine learning in a federated environment.

[4] built a production system, the first of its kind, for federated learning in the field of mobile devices using TensorFlow. The authors use the Federated Averaging algorithm proposed by [15]. However, their system has the following limitations, (i) though it does support privacy, it does not allow for new mechanisms to be added easily, (ii) it is dependent on TensorFlow so you cannot use it with other machine learning libraries, (iii) it does not focus on simulating virtual environments with multiple servers and multiple clients at different locations, and (iv) it is not very efficient because it does not support the parallel execution of training sessions.

Our simulator, PrivacyFL, was specifically developed to overcome these issues by providing a highly configurable system. For more details, please see Section 4.

There are a myriad of challenges pertaining to privacy in federated learning. In addition to guaranteeing privacy, it is important to make sure that the communication cost is cheap and efficient. There are numerous privacy definitions for federated learning [11][3] [23]. We can classify them into 2 categories: local privacy and global privacy. In local privacy, each client sends a differentially private value that is secure/encrypted to the server. In the global model, the server adds differentially private noise to the final output.

MPC and differential privacy are the most common techniques to guarantee security and privacy in a federated environment setup [3] [16][5].

In a federated learning setup, instead of sharing data, clients share models. Though this seems to provide increased privacy, there have been a multitude of privacy attacks, including reconstruction and membership inference attacks [22][19][17], demonstrating that additional privacy is required in the form of protecting model parameters. Cryptographic techniques such as secure multiparty computation (MPC) [24, 6, 12, 2] guarantee that clients don't learn anything except the final cumulative model weight. Also, these techniques do not compromise the accuracy of models. MPC allows clients to collaborate in order to compute a function without clients revealing their individual (and possibly sensitive) inputs to other clients.

Though MPC reveals only the final result and not the individual inputs of individual clients, sometimes, this information is sufficient to infer the individual input of a client. For example, let us consider the scenario where $n - 1$ clients collude against a honest client in a n client federated environment and the clients want to compute the overall sum. Once the sum is known, the clients can identify the private input of the non-colluding client. In order to overcome this, differential privacy [8] can be used in addition to MPC. This solution is resistant to collusion attacks and Figure 2 represents a privacy-preserving secure federated learning environment.

The above-mentioned methods are not dependent on each other and can exist independently in a federated learning environment. Our simulator allows the system designer to choose and configure different privacy and security mechanisms according to their needs and provides a flexible architecture.

Our system provides necessary guarantees for the "honest-but-curious" threat model. In this threat model, all the clients in the system follow the protocol but try their best to identify the data/model of other clients. As part of our future work, we plan to add solutions to the "malicious" threat model where clients need not follow the protocol and may conduct adversarial attacks.

3 Background

3.1 PySpark

PySpark is the state of the art framework for high performance cluster computing. PySpark uses a Java Virtual Machine (JVM) to allow Python users to use Spark [21]. PySpark is particularly beneficial when running the simulation on large datasets or with several iterations. Though the clients are prompted to train their models in a multi-threaded manner, they all have access to the same PySpark environment. PySpark makes use of a Directed Acyclic Graph (DAG) execution engine. This facilitates in-memory computation and improves run time. PySpark also guarantees high processing speed of data as it reduces the number of read and writes to disk. PySpark is also fault tolerant as it makes use of the underlying Spark abstraction-RDD.

3.2 Differential Privacy

PrivacyFL provides several differential privacy mechanisms that can be used and configured by the system designer. Differential privacy [9] is a mathematical privacy framework that captures the notion of risk when an individual is included in a dataset. Differentially private methods are capable of learning statistics of a dataset as a whole while protecting the sensitive information of individuals in the group. Essentially, differential privacy guarantees that the released results reveal little or no new information about an individual in the dataset.

Definition 1. *((ϵ, δ)-Differential Privacy) A randomized mechanism $\mathcal{M}$ satisfies (ϵ, δ)-differential privacy ((ϵ, δ)-DP) when there exists $\epsilon > 0$, $\delta > 0$, such that*

$$Pr\,[\mathcal{M}(D_1) \in S] \leq e^{\epsilon} Pr\,[\mathcal{M}(D_2) \in S] + \delta$$

holds for every $S \subseteq$ Range($\mathcal{M}$) and for all datasets D_1 and D_2 differing on at most one element.

When $\delta = 0$, we say that the randomized mechanism $\mathcal{M}$ satisfies $(\epsilon, 0)$-differential privacy

Definition 2. *(Global Sensitivity) For any real-valued query function $t : \mathcal{D} \rightarrow \mathbb{R}$, where $\mathcal{D}$ denotes the set of all possible datasets, the global sensitivity Δ of t is defined as*

$$\Delta = \max_{\mathcal{D}_1 \sim \mathcal{D}_2} |t(\mathcal{D}_1) - t(\mathcal{D}_2)|,$$

for all $\mathcal{D}_1 \in \mathcal{D}$ and $\mathcal{D}_2 \in \mathcal{D}$.

3.2.1 Laplacian Mechanism

The Laplacian mechanism preserves ϵ-differential privacy [8]. We make use of the random noise X drawn from the symmetric Laplacian distribution. The zero-mean Laplacian distribution has a probability density function $f(x)$.

$$f(x) = \frac{1}{2\lambda} e^{-\frac{|x|}{\lambda}},$$

where λ is the scale parameter. Given Δ of the query function t, and the privacy loss parameter ϵ, the *Laplacian mechanism* $\mathcal{M}$ uses random noise X drawn from the Laplacian distribution with scale $\lambda = \frac{\Delta}{\epsilon}$.

3.2.2 Distributing Differentially Private Noise

In addition to the basic differential privacy mechanisms, where each client adds differentially private noise, our framework also supports distributed differentially private noise addition. In distributed differential privacy, each client contributes a portion of the total differentially private noise. The sum of all these individual contributions result in a differentially private noise value.

As the Laplace mechanism satisfies differential privacy, we show how a Laplace Random variable can be generated from Gamma random variables.

$$L(\mu, \lambda) = \frac{\mu}{n} + \sum_{p=1}^{n} \gamma_p - \gamma'_p,$$

where μ and λ are the mean and scale parameters of the Laplacnian mechanism.

γ_p and γ'_p are Gamma distributed random variables with probability density functions defined as :

$$\frac{(1/s)^{1/n}}{\Gamma(1/n)} x^{1/n-1} e^{-x/s},$$

where $1/n$ is the shape parameter, s is the scale parameter and $\Gamma(p) = \int_0^\infty x^{p-1} e^{-x} dx$

We make use of this technique where each client adds $\gamma_p - \gamma'_p$ in our distributed Algorithm 1.

3.3 Differentially Private Federated Logistic Regression

We use differentially private logistic regression for our experiments.

In a local differentially private output perturbation setting, each client adds differentially private noise to their model parameters. In this setting, each client adds Laplacian noise to their model weights and participates in federated learning. Once clients train private models, they coordinate with the server to compute the global model. This client-server communication is repeated till convergence or a predefined number of rounds.

Let $\hat{w_i}^{dp}$, for $i \in 1$ to n, depict the local model constructed by each client after minimizing the objective function and adding differentially private noise to their weights. Here w represents the vector of weights and n represents the number of clients. $\hat{W} = \frac{1}{n} \sum_{i=1}^{n} \hat{w_i}^{dp}$is the differentially-private cumulative model.

In a trusted server setting, every client computes its model and sends it to the server. The server aggregates the model inputs it receives from clients and adds differentially private noise to the average. Here, $\hat{W} = \frac{1}{n} \sum_{i=1}^{n} \hat{w_i} + \eta$, η is the differentially private noise added to the cumulative model.

We also consider the case where each client contributes to a portion of the differentially private noise in the global setting. That is each client adds $\hat{w_i}^{dpPartial}$ to its model parameters, for $i \in 1$ to n. The server does the following, $\hat{W} = \frac{1}{n} \sum_{i=1}^{n} \hat{w_i}^{dpPartial}$.

For logistic regression in the federated multi-client setting, [13] proposed the following: For 1-$Lipschitz$, $\Delta = \frac{2}{n*k*\alpha}$ for a federated multi-client setting. Here k is the size of the smallest dataset amongst the n clients and α is the regularization parameter. Hence, $\eta = Laplace(\frac{2}{n*k*\alpha*\epsilon})$, where ϵ is the privacy loss parameter.

4 Architecture

The purpose of our simulator is to provide a light-weight but efficient Python framework that enables clients to simulate privacy-preserving secure federated learning. In a simulation, a client corresponds to an instance of the **ClientAgent** class. Each client agent is capable of interacting with the other client agents to establish common keys before the simulation. Client agents also communicate with an instance of the **ServerAgent** class if configured to be a centralized system. The server agent is an agent who is not training its models but instead serves to run the federated learning algorithm. It is responsible for requesting weights from the clients, averaging the weights, and returning to the clients the federated weights. Many applications will only require one server agent, but it is worth noting that with some modifications the framework can handle any number of server agents.

To run the simulation, an instance of the **Initializer** class must be created. The initializer will create the agents in the simulation and also invoke the client agents to perform any offline-stage logic such as key exchanges. The initializer also gives each agent a directory, which contains a mapping of agent names to agent instances so that agents can invoke each others methods, thereby enabling all agents to communicate. The initializer also gives each client agent an instance of the **ModelEvaluator** class, which it uses to evaluate the federated learning on a test dataset.

4.1 Simulation Lifecycle

This section provides an overview of the steps carried out in a simulation.

1. System designer wishing to simulate a federated learning environment specifies simulation parameters in `config.py` and runs `run_simulation.py`.

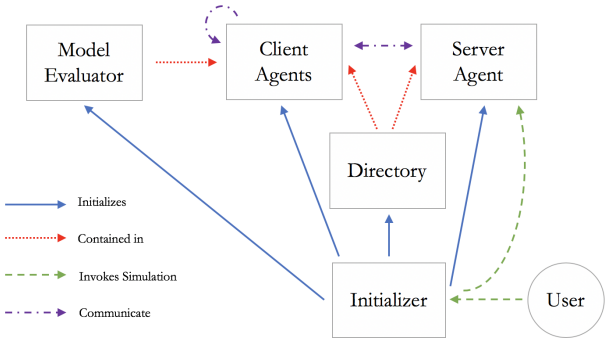

Figure 3: System diagram containing relationships between the important classes in PrivacyFL

2. `run_simulation.py` creates an instance of Initializer, which instantiates the client agents and server agent. It also creates a mapping of agent names to instances and passes those to each client.
3. `run_simulation.py` invokes the initializer's `run_simulation()`, which subsequently calls the server agent's
`request_values()` method.
4. Repeat for $i = 1, 2, 3, \ldots$ num_iterations :
   4.1 Server agent requests weights from clients in parallel by invoking each client's `produce_weights()` method.
   4.2 In `produce_weights()`, each client trains its machine learning model on D_i, the dataset for that iteration. Each client saves the weights locally, and then creates a copy of its weights to which it adds a security offset and differentially private noise. It returns the modified weights to the server.
   4.3 Upon receiving the weights from each client, the server averages the weights and returns the federated weights to each client in parallel through their `receive_weights()`.
   4.4 Upon receiving the federated weights for iteration i from the server, each client computes the accuracy of the federated model vs its local model on the test-set. Each client then computes whether its weights have converged and tells the server whether it is dropping out after this iteration.
   4.5 To conclude the iteration, the server tells all the clients which other clients have dropped out of the simulation.

4.2 Classes

There are 6 main classes in the simulator namely **Message**, **Agent**, **ClientAgent**, **ServerAgent**, **Initializer**, and **Directory**.

4.2.1 The Message Class

All agent-agent communications occur by one agent invoking another agent's method with a message as the sole argument. The message class contains metadata about the communication as well as a `body` attribute, which is a dictionary containing all the values an agent is sending.

4.2.2 The Agent Class

The Agent class is not meant to be initialized directly. Rather, its intended usage is to be sub-classed to create agents with more specific behavior. It is the base class for the ClientAgent and ServerAgent classes we provide, but can be sub-classed to create a different kind of agent if ClientAgent and ServerAgent are not easily modified to fit the needs of your simulation.

4.2.3 The ClientAgent Class

An instance of the ClientAgent class is an entity that is training a machine learning model on the same task as the other client agents. Client agents are assigned an `agent_number`, which is then appended to the string `client_agent` to create their name. For example, if there are three clients they are named client_agent0, client_agent1, and client_agent2.

There are two important public methods of ClientAgent that are invoked by the ServerAgent in the online portion of the simulation.

- `produce_weights(self,message)` is called every iteration and prompts the client to train its machine learning model on its dataset for that iteration. The message contains the iteration, a mutex lock used for multi-threading, and information about the simulated time.
- `receive_weights(self,message)` is called every iteration when the server has federated weights to return to the client. The message contains the iteration, weights, and the simulated time this message is received. This method returns `True` if the client agent's weights have converged with the federated weights and `False` otherwise.
- `remove_active_clients(self,message)` is called at the end of the iteration. The message contains the iteration, list of clients that have dropped out, and the simulated time the message is received. This method is only used if the simulation is configured with client drop out.

In addition, client agents have some methods for the client-client communication that are used to establish shared common keys with the other clients.

- The `send_pubkeys(self)` of each client is invoked by the initializer once all client agents have been initialized. In this method, each client computes secrets $a_{i,1}, \ldots, a_{i,n}$ and sends $a_{i,j}$ to client j by invoking that client's `receive_pubkey()` method.
- In `receive_pubkey(message)`, each client saves the public key sent to it by the other clients.
- `initialize_common_keys(self)` is called once all clients have exchanged public keys using the above two methods. In this method, the common key list is initialized. This common key list will be used to produce security offsets $r_{i,j}$ in the online portion of the simulation.

4.2.4 The ServerAgent Class

An instance of the ServerAgent class represents a third-party entity that coordinates the online portion of the simulation. ServerAgent has one method:

- `request_values(num_iterations)` is called by an instance of Initializer to signal the server agent to start requesting values from clients. `request_values()` first requests weights in parallel from the clients by calling their `produce_weights()`. It then averages these weights and returns them to the clients in parallel by calling their `receive_weights()` method. Finally, if the simulation is configured to allow client dropout, the server invokes the clients' `remove_active_clients()` to indicate which clients have dropped out at the end of this iteration.

.

4.2.5 The Initializer Class

An instance of the Initializer class is used to initialize the agents and model evaluator. In addition any offline stage logic, such as prompting the clients to perform a Diffie-Hellman key exchange, should occur in this class. In our example, it loads the MNIST dataset [14] and processes it into PySpark data-frames for easy usage by the client agents. To commence the simulation, one creates an instance of the Initializer class and invokes its `run_simulation()` method, which then invokes the server agent's `request_values()` to commence the online portion of the simulation.

4.2.6 The Directory Class

An instance of the Directory class contains a mapping of agent names to agent instances that allows agents to invoke other agents' methods by only having their name. An instance of Directory is created in the `__init__` method of the Initializer class after all the agents have been created. It is then passed on to all the agents using their `set_directory()` method.

4.3 Configurations and Features

PrivacyFL can be configured to simulate a variety of scenarios with the configuration parameters available in `config.py`. These configuration parameters are described in Table 1.

In addition, config.py allows system designers to set several other parameters such as the number of clients, custom ϵ and dataset sizes for each client, and thread-safe random seed for reproducibility.

USE_SECURITY	If True, the clients perform a Diffie-Hellman key exchange in the offline portion of the simulation to establish common keys for encryption. Has no effect on federated accuracy.
USE_DP_PRIVACY	If True, clients will add differentially private noise with parameters specified in the config file.
DP_ALGORITHM	If USE_DP_PRIVACY is True, DP_ALGORITHM can be set to 'Gaussian' or 'Laplace' to specify what kind of differentially private noise to use. More options can easily be configured by modifying the source code.
SUBTRACT_DP_NOISE	If True, clients will subtract the noise they added to the federated model upon receiving it from the server. When False, clients use the federated model computed by the server.
CLIENT_DROPOUT	If True, a client drops out of simulation when each weight in the federated model is within config.tolerance of the client's weight. The simulation continues without that client
SIMULATE_LATENCIES	If True, the system simulates how long it would take for each step in the protocol to complete using the user-defined communication latencies in config.LATENCY_DICT. If False, this information is not displayed.
USING_CUMULATIVE	If using our data partitioning module, this flag is useful for experimenting between dataset options. If False, the dataset for each iteration includes only the new data available that iteration. This makes sense for Algorithm 1. If True, the size of the dataset grows each iteration by the amount of new data. This configuration makes more sense for Algorithm 2, where the weights of the ith iteration are not used in producing the weights of the $i+1$th iteration.
USING_PySpark	The machine learning problem preformed by the clients need not be done in PySpark, but if so this flag should be set to True so that a PySpark session will be initialized at the start of the simulation.

Table 1: Configuration Parameters

5 Use case

To evaluate our simulator, we developed a secure and differentially private federated logistic regression algorithm. We used the Laplacian differential privacy mechanism available in the simulation in conjunction with a Diffie-Hellman key exchange for security.

5.1 Differentially Private Federated Averaging

The federated averaging algorithm is one of the most popular approaches of federated learning. It is a customized version of parallel Stochastic Gradient Descent. In our approach, Algorithm 1, each client runs U rounds of local SGD and returns the trained weights. Clients make use of new data every round. Here, each client adds the difference of gamma random variables to their unperturbed weights. In the aggregation phase, the server adds the perturbed weights it receives from all values. The server aggregates the weights into the final model, which are differentially private because $L(\mu, \lambda) = \frac{\mu}{n} + \sum_{k=1}^{n} \gamma_k - \gamma'_k$, where γ_k and γ'_k are Gamma distributed random variables. The client-server communication happens R times unless a client drops out.

We propose Algorithm 3, where each client contributes differentially private weights to the server, and Algorithm 2, where clients reuse old data used in previous rounds and also make use of any new data available.

The main difference between Algorithm 1 and Algorithm 2 is that, in Algorithm 2 clients retrain their model from scratch on their entire local dataset, which may or may not get updated every round, and don't use the weights received from the server while running local gradient descent. In Algorithm 1, clients compare their weights from the previous iteration with that of the global weights they receive from the server. With all algorithms, if local convergence is achieved then the client can drop out of the system.

In all the algorithms, when clients receive the federated weights from the server they are able to subtract the differentially private noise that they contributed to the federated weights.

Algorithm 1: Differentially Private Federated Averaging - Each client contributes a portion of the total differentially private noise

Server Implements:
  Initialize w_0
  for $r \leftarrow 1$ to R **do**
    for every client $k \in n$ **in parallel do**
      $w_{r+1}^k \leftarrow ClientProcedure(k, w_r)$
    end for
    $w_{r+1} \leftarrow \frac{1}{n} \sum_{i=1}^{n} w_{r+1}^i$
  end for
ClientProcedure(k,w):
  for local gradient update $u \leftarrow 1$ to U **do**
    $w \leftarrow w - \eta \nabla g(w)$
  end for
  Return $w + \gamma - \gamma'$

Algorithm 2: Differentially Private Weighted Averaging

Server Implements:
  for $r \leftarrow 1$ to R **do**
    for every client $k \in n$ **in parallel do**
      $w_{r+1}^k \leftarrow ClientProcedure(k, w_r)$
    end for
    $w_{r+1} \leftarrow \frac{1}{n} \sum_{i=1}^{n} w_{r+1}^i$
  end for
ClientProcedure(k, w_s)**:**
  if Round 1 **then**
    Initialize $w_c^{dp} = 0$
  end if
  if $w_s < w_c^{dp}$ **then**
    Initialize w
    for local gradient update $u \leftarrow 1$ to U **do**
      $w \leftarrow w - \eta \nabla g(w)$
    end for
    $w_c^{dp} \leftarrow w + \gamma - \gamma'$
  end if
  Return w_c^{dp}

5.2 Secure Aggregation

To provide the necessary security guarantees, we use Diffie-Hellman key exchange to ensure a secure aggregation protocol.

In the offline phase, every pair of clients P_i and P_j share n bit secret random values $r_{i,j} = r_{j,i}$ which is known only to them. The clients generate the common random values by running the Diffie-Hellman Key exchange protocol [7].

Every client splits their secret random values into a k bit offset $r'_{i,j}$ and a $n-k$ bit seed $s_{i,j}$. client P_i sends its weight w_i masked with the random offsets. For every $j > i$, clients add $r'_{i,j}$ and for every $j < i$ clients subtract $r'_{i,j}$. In order to generate new n bit secret random values for the next round, each client makes use of the $n-k$ bit seed of the current round. This process of generating random offsets and seeds continue as long as clients are sending weights to the server, i.e., until clients drop out. Hence, clients only need to communicate with each other once at the start of the simulation to ensure security for all iterations, thereby reducing the amount of client-client communications and overall communication time. If desired, the clients can perform the Diffie-Helman key exchange again at any time in the simulation.

Algorithm 3: Differentially Private Federated Averaging - Each client adds differentially private Laplace noise

Server Implements:
  Initialize w_0
  for $r \leftarrow 1$ to R **do**
    for every client $k \in n$ **in parallel do**
      $w_{r+1}^k \leftarrow ClientProcedure(k, w_r)$
    end for
    $w_{r+1} \leftarrow \frac{1}{n} \sum_{i=1}^{n} w_{r+1}^i$
  end for
ClientProcedure(k,w):
  for local gradient update $u \leftarrow 1$ to U **do**
    $w \leftarrow w - \eta \nabla g(w)$
  end for
  Return $w + Laplace(\frac{2}{n*k*\alpha*epsilon})$

6 Experiments

For our evaluation, we configured the clients to train on UCI's Machine Learning MNIST dataset [14] [20] for eight iterations. This dataset is easily loaded using scikit-learn's `datasets.load_digits()` module. We ran the experiments on a MacBook Pro 2.3 GHz Intel Core i5, using Python's multithreading module for parallelization. All experiments were initialized with the same thread-safe random seeds, ensuring fair comparisons where possible.

6.1 Experiment 1: Accuracy vs Privacy vs Number of Clients Trade-offs

One of the principal uses of the simulator is to enable clients to assess whether the accuracy of their model would increase if they participated in the federated learning. For that reason, we compare the mean accuracy of the clients' model with the accuracy of the federated/global model for each iteration. We used Algorithm 2. When computing a client's accuracy on iteration i, we consider its weights with no differential privacy since this scenario corresponds to clients not participating in the federated learning, thus having no need to add differentially private noise. Figure 4 compares the mean of three clients' accuracy with the federated accuracy for different values of ϵ, where a smaller value of ϵ corresponds to less noise. As expected, the federated model's accuracy increases as ϵ increases, but there is not a significant difference between $\epsilon = 1$ and $\epsilon = 8$. It is also important to note that the federated accuracy for the $\epsilon = 0.1$ line increases more rapidly than the mean client accuracy. This is expected behavior since the amount of noise added per client is smaller as the size of each client's dataset increases by thirty samples each iteration.

In the following example, we simulate a larger number of clients. We used the KDDCup99 dataset [1] and Algorithm 1. In Figure 5, one can also see the result of varying the number of clients and ϵ. As expected, the accuracy of the federated model increases with ϵ since less noise is added. Also, accuracy increases as the number of clients increases. This behavior is expected since more clients contributing to the federated model should increase the accuracy of the federated model.

6.2 Experiment 2: Comparison of Noise Adding Mechanisms

Table 2 shows the accuracy increases that can be achieved by using some of the existing configuration flags in the simulation. Laplacian noise is commonly used for differential privacy and is available as one of our differentially private algorithms. Algorithm 2, however, uses distributed differentially privacy by having each client add noise from the difference of Gamma random variables. In addition, our simulation also provides the option for each client to subtract its own noise upon receiving the federated model from the server. In Table 2 one can see how each of these improvements corresponds to an increase in accuracy of the federated model.

6.3 Experiment 3: Privacy Constraints

This experiment illustrates a use case where each client can have different amounts of data and privacy requirements. Suppose Clients 1 and 2 each have 150 data points and $\epsilon = 1.0$. However, Client 3 has 250 data points but also a more stringent privacy requirement, $\epsilon = 0.1$. Clients 1 and 2 may want to decide whether they should only collaborate among themselves, include Client 3, or participate in federated learning at all.

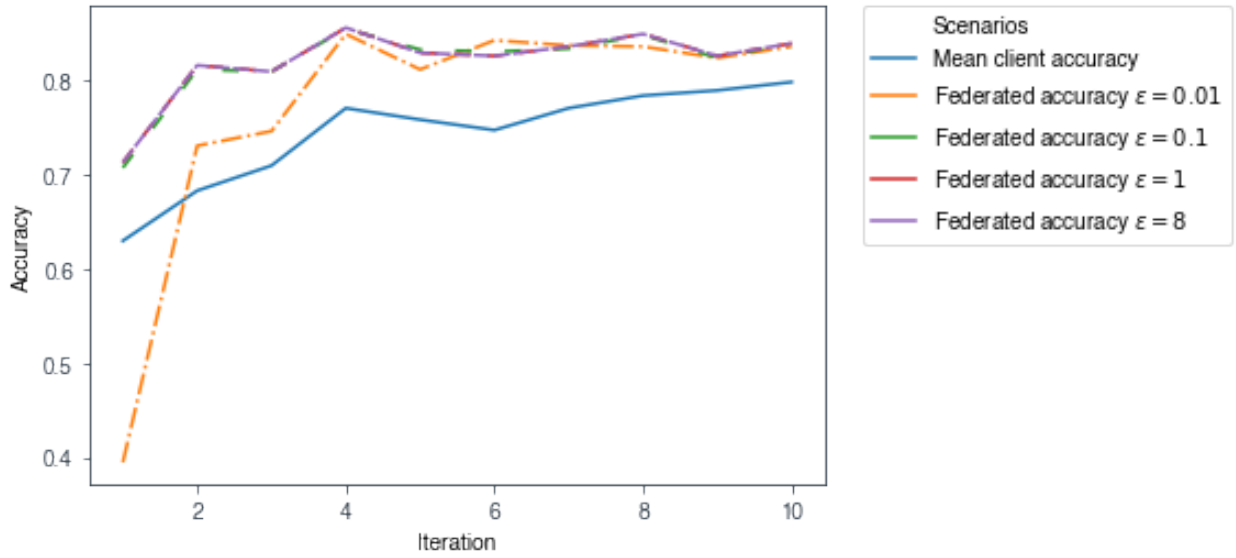

Figure 4: Mean client accuracy on test dataset versus federated model accuracy for different ϵ. Clients follow Algorithm 2.

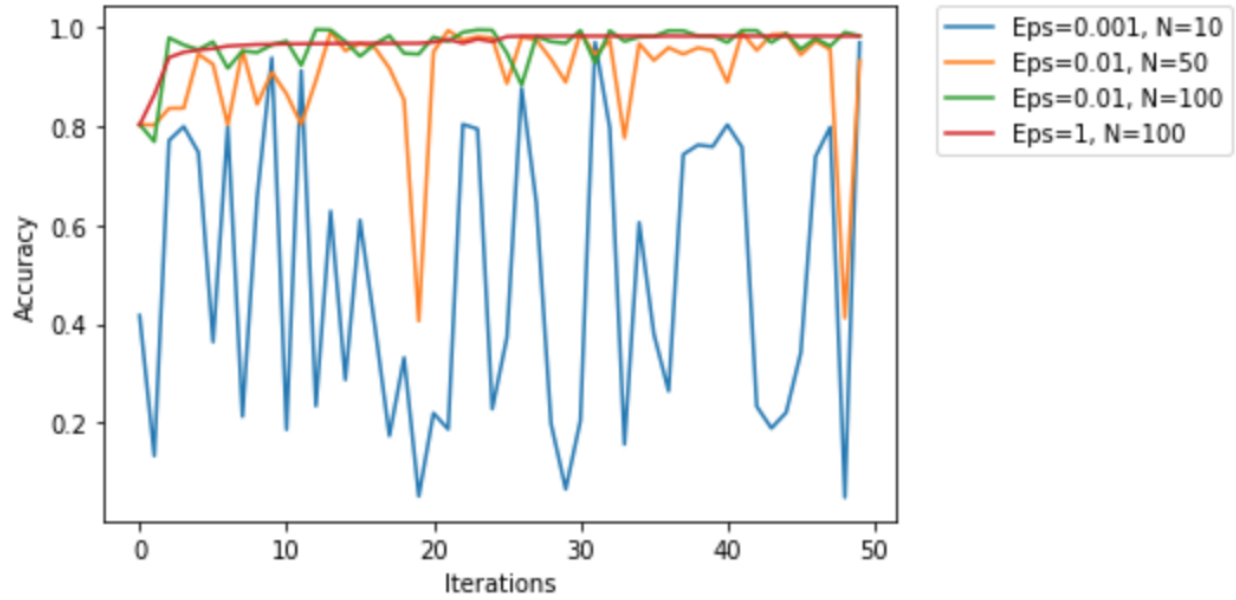

Figure 5: Accuracy vs Iterations for different values of ϵ and number of parties

Noise mechanism	Accuracy
Laplacian Noise (Algorithm3)	0.40
Difference of Gamma random variables (Algorithm1)	0.63
Difference of Gamma random variables and subtracting personal contribution	0.70

Table 2: A comparison of the different protocols described in this paper with $\epsilon = 0.01$

- Scenario 1. No clients participate in federated learning so their accuracy is their own model's accuracy.

- Scenario 2. Clients 1 and 2 participate in the federated learning but do not include Client 3 because Client 3 has $\epsilon = 0.1$ while Clients 1 and 2 have $\epsilon = 1.0$.
- Scenario 3. All three clients participate in the federated learning and use Client 3's privacy requirement, $\epsilon = 0.1$.
- Scenario 4. All three clients participate in federated learning, but Clients 1 and 2 use their privacy requirement $\epsilon = 1.0$, and Client 3 uses its privacy requirement, $\epsilon = 0.1$

Scenarios	Client 1	Client 2	Client 3
Scenario 1	0.750	0.793	0.823
Scenario 2	0.806	0.810	-
Scenario 3	0.767	0.800	0.813
Scenario 4	0.827	0.830	0.850

Table 3: Each client's accuracy using Algorithm 2 in the four different scenarios

Table 3 shows the clients' accuracy in each scenario. These values are obtained by using the configuration options that correspond to Algorithm 2. As is apparent from the table, Clients 1 and 2 do not benefit from including Client 3 in the simulation unless each client can use its own value of ϵ. In that case (Scenario 4), each client benefits from participating in the federated learning.

6.4 Experiment 5: Decentralized (Serverless) Federated Learning

The simulation can also be easily modified beyond the configuration parameters. In the following example, we subclassed the Agent class to create a new kind of Agent that is able to perform federated machine learning without a server. In this case, clients send their masked weights directly to the other clients. Once a client has received the weights from all the other clients, it is able to average them to create a federated model. Figure 6 shows the results of three clients each training according to Algorithm 1, with thirty data points per iteration for seven iterations. In this example, clients use the security protocol but no differential privacy. This behavior is achieved by setting the USE_DP_PRIVACY flag to False and the USE_SECURITY flag to True. As one can see from Figure 6, all clients clearly benefit from participating in the federated learning in this situation.

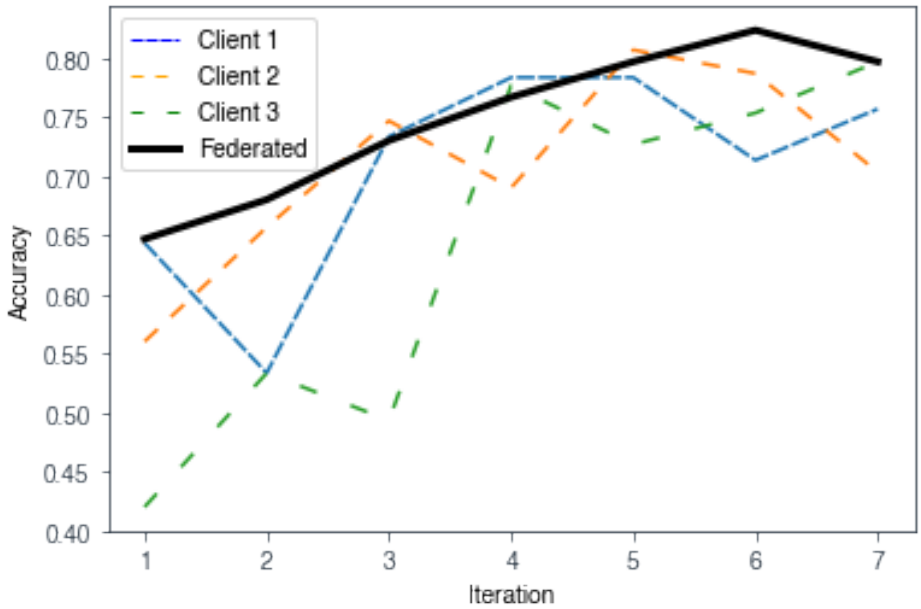

Figure 6: Simulation with Algorithm 2 but modified so that no server is required.

Iteration	Simulated Time to Receive Federated Weights (seconds)		
	Boston	Singapore	NYC
Iteration 1	4.310	6.010	4.110
Iteration 2	4.306	6.006	4.106
Iteration 3	0.909	Dropped Out	0.709

Table 4: Simulated time to receive federated weights by iteration in an example where Singapore, the farthest client, drops out after the second iteration.

Iteration	Personal Computation Time to Produce Weights (seconds)		
	Boston	Singapore	NYC
Iteration 1	0.0128	0.0101	0.0102
Iteration 2	0.0106	0.0059	0.0072
Iteration 3	0.0086	Dropped out	0.0083

Table 5: Time taken per client to produce weights using 1 on 100 data points per iteration.

6.5 Experiment 6: Real-World Latency Simulation

For this experiment, we simulated the situation where the clients are in different geographic locations. We configured three clients as if they were in the Boston, Singapore, NYC, with the server located closest to the NYC client. As such, we designated the following communication latencies: (i) Boston-Server: 0.3 seconds, (ii) Singapore-Server: 2.0 seconds, (iii) NYC-Server: 0.1 seconds.

One can also set client-client latencies. In this example, those only come into effect for the offline portion of the simulation since there is no client-client communication during the online portion of the simulation.

Table 4 shows the simulated time, which measures the time from the start of each iteration to the time when clients would receive the federated weights from the server. The simulated time is available for any step in the protocol since it is included in every message between clients. Upon receiving a message, an agent computes how long its logic has taken. The agent then adds the time that its logic as taken as well as the simulated communication time for whomever the client is sending the message to. This produces a new simulated time for the receiving agent. In Table 4 one can also see that the simulated times for Boston and NYC are significantly lower on the third iteration. This is because clients were permitted to drop out since the CLIENT_DROPOUT flag was set to True. The Singapore-Server latency is the highest, which means that once Singapore drops out at the end of the second simulation the other clients receive the federated weights faster since the server does not need to wait on the Singaporean client's weights.

Table 5 shows the time taken by each client to compute its weights on each iteration. Since the USE_CUMULATIVE flag is set to False for 1, each iteration contains 100 new data points for each client which explains why the computation times are fairly small and constant. This same figure for Algorithm 2 would have personal computation times increasing since the datasets are cumulative for Algorithm 2.

7 Reproducibility

PrivacyFL is available on GitHub at `https://github.com/vaikkunth/PrivacyFL/` under the MIT License.

To run the simulator, the repository should be cloned locally and a conda environment with the necessary dependencies should be created as shown in Figure 7.

The config.py file contains a variety of parameters as described in Section 4.3 that allow the simulation to be customized. `utils/data_formatting.py` provides an an example of how a dataset, in this case the MNIST dataset, can be pre-processed for PySpark. In addition, the `config.py` file also has random seeds for each client that ensure that randomness is handled in a reproducible thread-safe manner.

```
conda env create -f environment.yml -n YourEnvironmentName
cd src
python run_simulation.py
```

Figure 7: Steps to execute the code

8 Conclusion and Future Work

In this paper, we presented PrivacyFL, a simulator for privacy-preserving and secure federated learning. The primary features of our system include latency simulation, robustness to client departure/failure, support for both server-based and serverless federated learning, and tunable parameters for differential privacy and MPC.

We evaluated our simulator by running multiple experiments on federated logistic regression to demonstrate the flexibility of our framework. Our simulator and protocols can be easily applied to other federated machine learning algorithms as well. PrivacyFL is highly customizable and system designers can easily configure parameters such as setting different latencies/delays, choosing appropriate differential privacy mechanisms, and simulating clients dropping out or being added dynamically.

As part of our future work, we plan to explore solutions to the "malicious" threat model where clients need not follow the protocol and may conduct adversarial attacks. We plan to extend our simulator to real-time cluster environments and perform experiments on larger datasets. Once our simulator is capable of being deployed in a cloud computing platform, we plan to work with pilot partners for the real-time testing of our system and protocols.

References

[1] Stephen D Bay, Dennis Kibler, Michael J Pazzani, and Padhraic Smyth. The uci kdd archive of large data sets for data mining research and experimentation. *ACM SIGKDD explorations newsletter*, 2(2):81–85, 2000.

[2] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[3] Abhishek Bhowmick, John Duchi, Julien Freudiger, Gaurav Kapoor, and Ryan Rogers. Protection against reconstruction and its applications in private federated learning. *arXiv preprint arXiv:1812.00984*, 2018.

[4] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecny, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[5] 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. ACM, 2017.

[6] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract). In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, page 462, 1987.

[7] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.

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

[9] Cynthia Dwork and Adam Smith. Differential privacy for statistics: What we know and what we want to learn. *Journal of Privacy and Confidentiality*, 1(2):2, 2010.

[10] Quan Geng, Peter Kairouz, Sewoong Oh, and Pramod Viswanath. The staircase mechanism in differential privacy. *IEEE Journal of Selected Topics in Signal Processing*, 9(7):1176–1184, 2015.

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

[12] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 78, 1998.

[13] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. Distributed learning without distress: Privacy-preserving empirical risk minimization. In *Advances in Neural Information Processing Systems*, pages 6343–6354, 2018.

[14] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998. *URL http://yann. lecun. com/exdb/mnist*, 10:34, 1998.

[15] Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data. *Google Research Blog*, 3, 2017.

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

[17] 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.

[18] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Stand-alone and federated learning under passive and active white-box inference attacks. *arXiv preprint arXiv:1812.00910*, 2018.

[19] Milad Nasr, Reza Shokri, and Amir Houmansadr. Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 739–753. IEEE, 2019.

[20] DJ Newman, S Hettich, CL Blake, and CJ Merz. Uci repository of machine learning databases. dept. information and computer sciences, univ. california, irvine, 1998.

[21] Joshua Rosen and Sean Owen. Pyspark internals. *Disponıvel em: https://cwiki. apache. org/confluence/display/S-PARK/PySpark+ Internals. Acessado em*, 14(12):2017, 2016.

[22] 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.

[23] Om Thakkar, Galen Andrew, and H Brendan McMahan. Differentially private learning with adaptive clipping. *arXiv preprint arXiv:1905.03871*, 2019.

[24] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.