Secure Analytics: Federated Learning and Secure Aggregation

Jack Sullivan

October 15, 2018

1 Introduction

Analytics such as training machine learning models on sensitive real-world data promise improvements to everything from medical screening to disease outbreak discovery. Such sensitive data is becoming increasingly available through the widespread use of mobile devices for example. However, large-scale collection of sensitive data entails risks to users' sensitive information.

The general setting for secure analytics is to consider n parties that have sensitive data (such as temporal location data) and are not willing to share such data individually, but are willing to share an aggregate computation over joint data. Computation in this case can span from analytics such as SQL queries to learning.

2 Federated Learning

An example of a secure analytics technique that handles our general setting in the learning case is federated learning. Federated learning is a machine learning setting where the goal is to train a high quality, centralized model while training data remains distributed over a large number of clients each with unreliable and relatively slow network connections [2]. For each iteration of learning algorithms, each client independently computes an update to the current model based on its local data, and communicates this update to a central server, where the client-side updates are aggregated to compute a new global model. The typical clients in this setting are mobile phones and communication efficiency is of the utmost importance.

2.1 Example: Google's Gboard

Google's Gboard enables users to search Google and share results from their keyboard, anything from flight times and weather to news [3]. Gboard also predicts possible searches that may be relevant to the user via federated learning. When Gboard shows a suggested query, their phone locally stores information about the current context and whether they clicked the suggestion. Each smart phone device has lots of data and rather than sending all of its data to the server to update the centralized model, users instead send a focused update of their local data. All the training data remains on user devices and no individual updates are stored in the cloud.

2.1.1 Threat Model

Our threat model is the same for Gboard and the below secure aggregation example. For users, we assume there are very large numbers of them (in the millions), can dropout at any time, and have limited latency and throughput. Both users and the server are assumed to be honest-but-curious, meaning they follow the protocol but can piece together data that goes through them to potentially learn sensitive data. Also as discussed in class, users can realistically be malicious as someone can take their phone and perform bad actions.

2.1.2 Protocol and Model Diagram

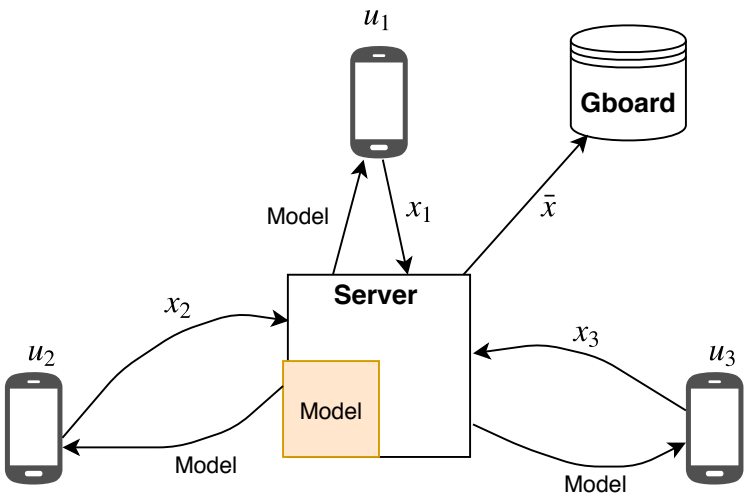

Figure 1: Gboard Federated Learning Model

Figure 1 outlines Gboard's federated learning protocol listed below:

1. The server sends the model to each user's device.
2. Users send updates x_u's to server once they collect new data.
3. The server averages all user updates, updates model, and stores the averaged update ($\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$) on the Gboard database.

Advantages of this scheme are that it supports any number users with limited latencies/throughputs and is efficient. The disadvantage of this scheme is that the server sees plaintext individual user updates and based on how updates are defined, the server may be able to infer what was typed by that user. An example of the server being able to infer what was typed is if an update included a word dependent indicator update that returns 1 if a user typed "restaurant" and a 0 otherwise. Secure aggregation in the next section will help address this problem by the server only being able to see the aggregated result of the data values.

3 Secure Aggregation

The problem of computing a multiparty weighted sum where no party reveals its update in the clear (even to the aggregator) is referred to as secure aggregation. The secure aggregation primitive can be used to privately combine the outputs of local machine learning on user devices in order to update a global model. Training models in this way ensures a user that they can share an update knowing that the service provider will only see that update after it has been averaged with those of other users. Google's paper *Practical Secure Aggregation for Privacy-Preserving Machine Learning* [1], aims to provide secure aggregation in the above federated learning case.

3.1 Problem Definition, Goals, and Assumptions

A reminder of the threat model is we assume that both the server and users are honest-but-curious. If users are malicious, they have the power to bias aggregate results without being detected. An example is calculating the average speed on a road and a malicious user can submit -100 mph, which greatly decreases

the average and because the server can only see the aggregated result and not the individual values, the malicious user's impact is not traceable.

This paper is particularly focused on the setting of mobile devices, where communication is extremely expensive, and dropouts are common. The goal is to invoke MPC with the server, but general MPC such as garbled circuits are expensive, have large overhead, and will not work with dropouts.

We also assume there exists a public key infrastructure such that every user knows the public keys of legitimate users (to prevent Sybil attacks as a user can create t fake users such that they can construct secret s with their t shares). Specifically, users can communicate their public keys through secure channels that are setup via Diffie-Hellman key exchange.

The server uses federated learning to get update vectors taken over a random subset of users. Using a secure aggregation protocol to compute these weighted averages would ensure that the server may learn only that one or more users in this randomly selected subset wrote a given word, but not which users. Federated learning motivates a need for a secure aggregation protocol that:

1. Operates on high-dimensional vectors.
2. Is highly communication efficient regardless of the subset of users involved in the computation.
3. Is robust to users dropping out.
4. Provides the strongest possible security under the constraints of a server-mediated, unauthenticated network model.

As is common in this space, the paper aims to design a tailored secure aggregation protocol instead of using generic MPC libraries.

3.2 First Attempt: Masking with One-Time Pads

The first attempt of securely aggregating user data vectors involves each pair of users u, v with a total order on users $u < v$ agreeing on a secret denoted as $s_{u,v}$. Each user blinds their input value x_u with the secrets they constructed with all other users. If u adds $s_{u,v}$ to x_u and v subtracts it from x_v, then the mask will be canceled when their vectors are added, but their actual inputs will not be revealed. Formally, each user's blinded value is

$$y_u = x_u + \sum_{u<v} s_{u,v} - \sum_{u>v} s_{v,u} \bmod R$$

where R is a public large prime number. Figure 2 below shows an example with three users.

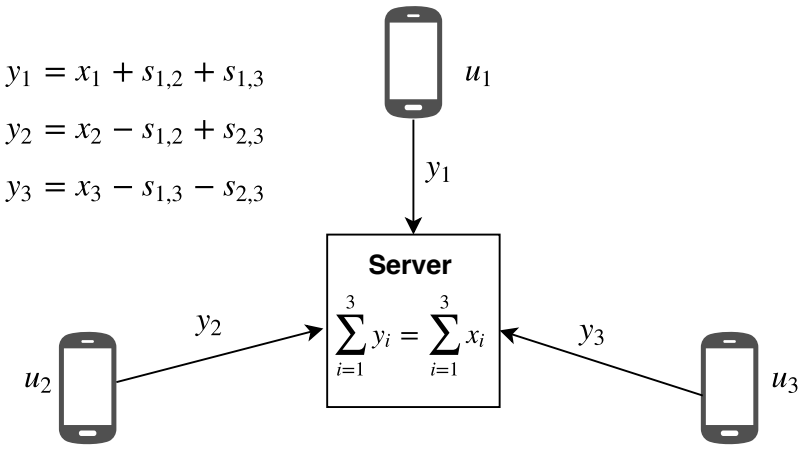

Figure 2: Aggregate via One-Time Pads

The proof of $\sum_{i=1}^{3} y_i = \sum_{i=1}^{3} x_i$ is simple as the blinding terms additively cancel:

$$\sum_{i=1}^{3} y_i = x_1 + s_{1,2} + s_{1,3} + x_2 - s_{1,2} + s_{2,3} + x_3 - s_{1,3} - s_{2,3} = \sum_{i=1}^{3} x_i$$

The scheme's shortcomings are:

- Quadratic communication: Every user pair u and v has to produce their secret $s_{u,v}$.
- If any person drops, the $\sum_{\forall i} y_i$ becomes junk as all blinding terms will not be canceled out.

3.2.1 Improving Quadratic Communication with Diffie-Hellman Key Exchange

We can improve users having to establish secure connections with all other users directly before every query. With an agreed upon large prime modulus p and generator g, users can instead send their public key $g^{a_u} \mod p$ where a_u is their secret key to the server. The server then broadcasts a user's public key to all other users, where they raise the public key of to the power of their secret key to create their shared secret. We can assume that we can route messages through the server as we assume the server is honest-but-curious. An example of this can be seen with u_1 and u_2:

$$u_1 \text{ computes } (g^{a_2})^{a_1} \mod p = g^{a_1 a_2} \mod p = s_{1,2}$$

$$u_2 \text{ computes } (g^{a_1})^{a_2} \mod p = g^{a_1 a_2} \mod p = s_{1,2}$$

There is still quadratic traffic and total data sent, but this method is much more simple and efficient for users by using the server.

3.3 Second Attempt: Masking with One-Time Pads with Recovery

Similar to above, users send their blinded vector y_u to the server and to handle dropped-out users, the server would notify the surviving users of the drop-out and to have them each reply with the common secret they computed with the dropped user. Secrets of the form $s_{\text{in,out}}$ and $s_{\text{out,in}}$ need to be recovered by the surviving users and server doesn't have to worry about secrets $s_{\text{in,in}}$ and $s_{\text{out,out}}$ as they are already not blinding the sum.

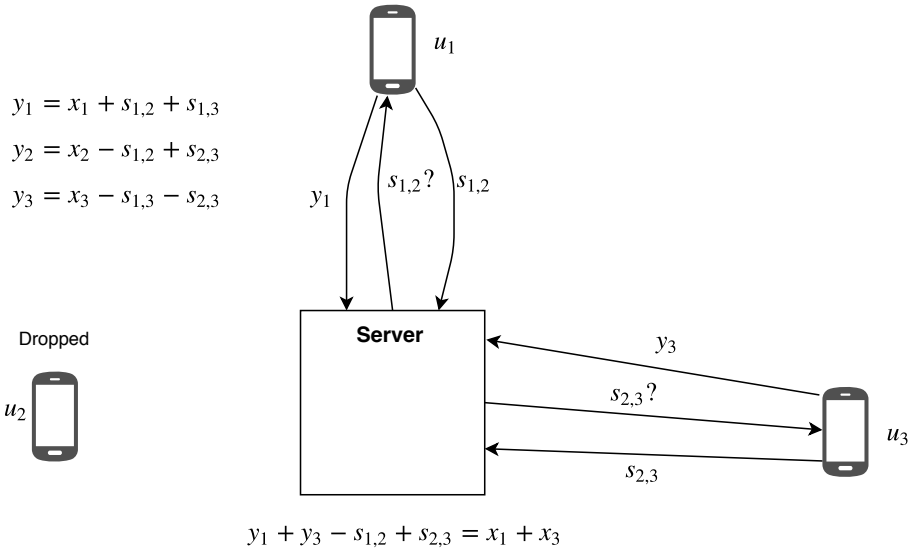

Figure 3: Aggregate via Secret Recovery

Figure 3 shows an example of this protocol with u_2 dropped-out and u_1 and u_3 online. Since the other users do not know u_2 is dropped, they include the secrets with u_2 anyways and the server asks the surviving users their secrets with u_2 to cancel out the blinding terms.

Approach Problems:

- Additional users may drop out in the recovery phase before replying with the secrets, which would thus require an additional recovery phase for the newly dropped users' secrets to be reported, and so on, leading the number of rounds up to at most the number of users.
- Delayed communication can lead to the server assuming a user is dropped and thus can recover their secrets from all other users. This leads to the server being able to decrypt their individual x_u after receiving the user's secrets.

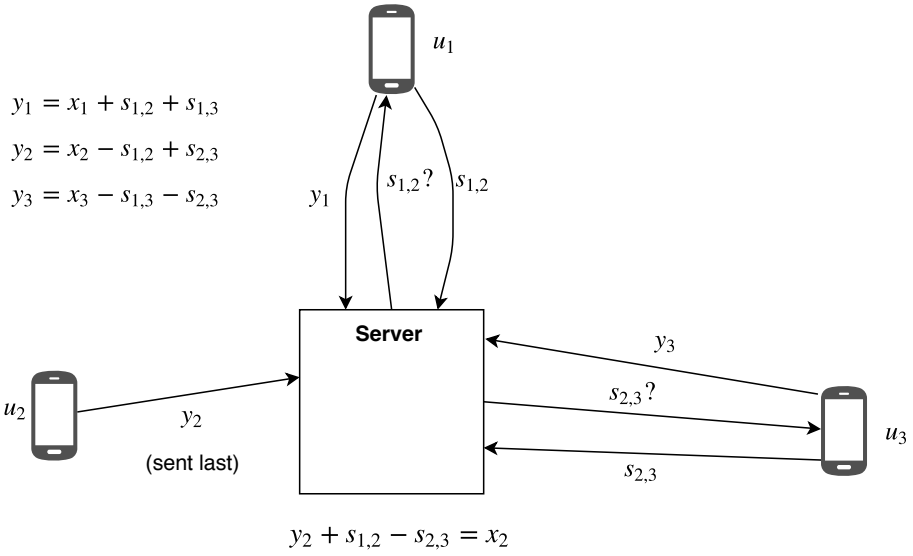

Figure 4: Server Decrypting User Data Vector through Delayed Communication

An example of delayed communication is depicted in Figure 4. If the server doesn't hear from u_2, it asks other users for u_2's secrets. If y_2 eventually is delivered, an honest-but-curious server can decrypt (unblind) u_2's data.

3.3.1 Malicious Server Seeing User Inputs

If the server is malicious and untrusted, it can lie about which users dropped and can thus see a user's individual vector x_u in plaintext. See Figure 4 to see how a server unblinds a user's input.

3.4 Shamir's Secret Sharing

The next suggested scheme relies on a crypto building block known as Shamir's t-out-of-n Secret Sharing [4], which allows a user to split a secret s into n shares, such that any t shares can be used to reconstruct secret s (Property 1), but any set of at most $t-1$ shares gives no information about s (Property 2).

- Given $f(x) = ax + b$:
  - Knowing 2 or more points of the function $f(x)$ will reveal $f(x)$, therefore revealing $s = f(0)$.
  - Knowing less than 2 points of the function $f(x)$ will not reveal $f(x)$, therefore not revealing $s = f(0)$.
- More generally, given $f(x) = a_{k-1}x^{k-1} + \cdots + a_2x^2 + a_1x + a_0$:
  - Knowing k or more points of the function $f(x)$ will reveal $f(x)$, therefore revealing $s = f(0)$.
  - Knowing less than k points of the function $f(x)$ will not reveal $f(x)$, therefore not revealing $s = f(0)$.

Consider an example of Shamir's Secret Sharing with 2-out-of-3 ($t = 2$, $n = 3$) where the secret s splits into $n = 3$ pieces s_1, s_2, s_3. In this case, we can uncover the secret s by constructing a one dimensional line $f(x) = ax + b$ with two or more secrets s_i's where $s = f(0)$. Thus Property 1 is satisfied as at least $t = 2$ secrets can be combined to reconstruct secret s. Property 2 is also satisfied as having strictly less than $t = 2$ points does not enable an individual to learn anything about the secret s. Figure 5 demonstrates that having all three shares of secret s is enough to reconstruct the linear function and can thus recover s by evaluating $f(0)$.

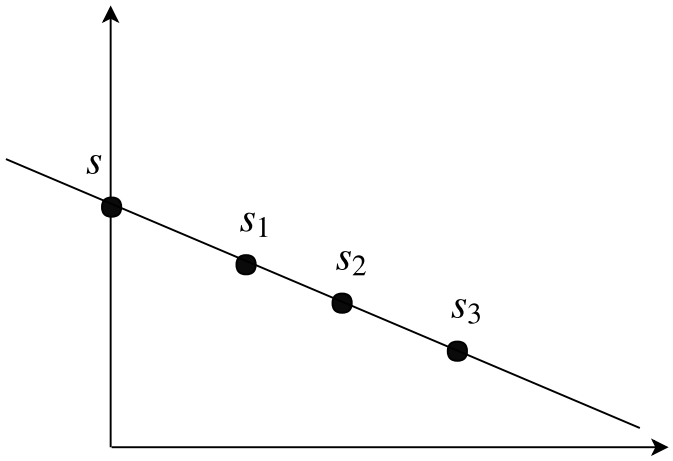

Figure 5: Sufficient Number of Points to Recover the Secret

An example of having one s_i and knowing nothing about the secret s is shown below in Figure 6, as there can be infinite linear lines that fit that one point.

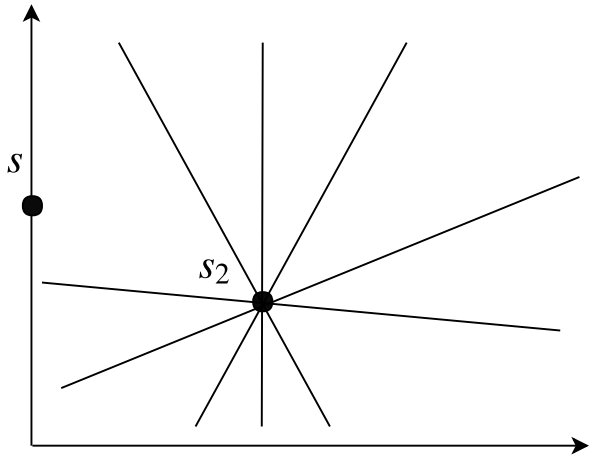

Figure 6: Insufficient Number of Points to Recover the Secret

3.4.1 Shamir Secret Sharing to Form Virtual Certificate Authorities

By using Shamir's secret sharing, the system secret or private key of a certificate authority is divided into n parts such that any k parts can perform as a virtual certificate authority. Each part is given to a node in the network. A node's properties can be summarized as below:

- A node can be initialized securely with its share of the secret.
- A node knows the public keys of all nodes in the network including ones that will join the network after initiation.

Now consider an example when Alice needs to find out the public key of Bob. Alice sends out a broadcast message to its neighbors requesting a certificate for Bob. Each node hears this message and generates a partial certificate with its partial system secret and sends it to a combiner. A combiner is just a node which combines partial certificates and generates a complete certificate to send to Alice. Any node can take on the role of a combiner. Conversely, a node does not gain any extra information about the system secret by being a combiner.

3.5 Third Attempt: Handling Dropped Users

To combat the additional recovery phases for newly dropped users between rounds of communications, we can use a threshold Shamir secret sharing scheme and have each user send shares of their Diffie-Hellman secret to all other users. This allows pairwise secrets to be recovered even if additional parties drop out during the recovery, as long as some minimum number of parties (equal to the threshold) remain alive and respond with the shares of the dropped users keys. The protocol is concisely:

1. Each user u splits their Diffie-Hellman secret a_u into $n-1$ parts $a_{u1}, a_{u2}, \ldots, a_{u(n-1)}$ and sends the shares to all other $n-1$ users.
2. Server receives y_u (same as above) from all online users in the first communication round $U_{\text{online,round 1}}$.
3. Server computes the set of users dropped, denoted as $U_{\text{dropped,round 1}}$.
4. Server contacts $U_{\text{online,round 1}}$ to ask for shares for $U_{\text{dropped,round 1}}$.
   (a) Assumption is that at least t of $U_{\text{online,round 1}}$ are still online in round 2.
5. Server recovers shares for $U_{\text{dropped,round 1}}$ and removes their randomness from the final aggregate.
   (a) Specifically, for a reconstructed one time secret a_u, because the server knows the public keys of all users ($g^{a_j}\ \forall j \in U_{\text{online,round 1}}$), raising a_u to those public keys yields each pairwise secret that the server can use to eliminate the randomness.

This approach solves the problem of unbounded recovery rounds, but still has the issue of user data being accidentally leaked to the server with delayed communication of their y_u.

3.6 Final Attempt: Double-Masking

The goal of double-masking is to protect each user's x_u even when the server reconstructs u's masks. First, each user u samples an additional random secret α_u and distributes its shares to each of the other users. When generating y_u, users also add this secondary mask:

$$y_u = x_u + \alpha_u + \sum_{u<v} s_{u,v} - \sum_{u>v} s_{v,u} \bmod R$$

During the recovery round, the server must make an explicit choice with respect to each user u. From each surviving member v, the server can request either a share of the common secret $s_{u,v}$ associated with u or a share of the α_u for u; an honest user v will never reveal both kinds of shares for the same user. After gathering at least t shares of $s_{u,v}$ for all dropped users and t shares of α_u for all surviving users, the server can subtract off the remaining masks to reveal the sum.

Overall computation and communication of this protocol is still on the order of n^2 where n is the number of users. One new problem is that if $t < \frac{n}{2}$, the server can ask two sets of users for the Diffie-Hellman secret α and secret s separately, which can lead to the server being able to decrypt a user's data vector x_u.

References

[1] K. Bonawitz. "Practical Secure Aggregation for Privacy-Preserving Machine Learning". 2017.

[2] J. Konecny. "Federated Learning: Strategies for Improving Communication Efficiency". 2017.

[3] H. B. McMahan. "Communication-Efficient Learning of Deep Networks from Decentralized Data". 2016.

[4] A. Shamir. "How to Share a Secret". 1979.