Secure Aggregation for Federated Computation

David Goetze and Mark Bissell
Williams College

1 Introduction

Massive quantities of data are generated every day across a variety of device types and for a variety of purposes. Consumer devices such as phones, laptops, wearables, and appliances track personal data and device metrics, while businesses and other large institutions amass enormous databases of proprietary and industry-specific information. Importantly, much of this data is decentralized; it remains separated from other data silos, often for reasons related to privacy and legal restrictions around data sharing. For example, a smartphone producer might wish to have each client's phone send its local data to a centralized server for analysis (e.g. to improve the phone keyboard's text recommendation algorithm), but it may not be able to do so without jeopardizing customers' privacy. Similarly, a network of hospitals might wish to aggregate their anonymized patient data to perform observational research studies, but privacy laws might forbid such sharing due to restrictions around sending patient-level data.

These examples represent a similar technical problem: how can an entity compute an aggregate across a decentralized dataset while preserving the privacy of each individual data owner? Secure aggregation is a strategy that addresses this challenge using cryptographic protocols.

In this paper, we present two different implementations of the secure aggregation protocol proposed by Bonawitz et al. [1]. The first implementation simulates *cross-device* federated computation — situations similar to the smartphone example mentioned above, in which private data is aggregated across a large number of distinct devices under the coordination of a central server. Our second implementation is motivated by the aforementioned hospital data-sharing situation, in which private data is aggregated across a smaller consortium of separate administrative units. We refer to this as *cross-silo* federated computation.

Developing effective methods of secure aggregation is essential for preserving the privacy of individual data owners while simultaneously allowing organizations to leverage insights that can only be computed using large quantities of data. Secure aggregation is especially useful in the context of federated learning — a process for training a machine learning model across private, decentralized data by having each data owner train the model locally and then send their local model weights to a central entity for aggregation. Federated learning can be used in the cross-device setting to develop models that improve user experience for all device owners, and in the cross-silo setting to collaboratively develop models across organizations. For example, federated learning has been proposed as a way for large banks to develop shared models for identifying fraud while preserving the privacy of each institution's dataset.

Secure aggregation plays an important role in federated learning since researchers have been able to reconstruct individual training examples using only the parameters of a trained neural network [2]. Sending model updates to a central entity therefore risks exposing each data owner's private information unless secure aggregation is used.

2 Background

A foundational 2016 paper from Bonawitz et al. [1] outlined several methods for securely aggregating data for federated learning. Our paper presents two implementations of the first cryptographic protocol provided in the paper, which we now describe.

Given a collection of n nodes $X_1, X_2, \ldots, X_n$ each with some secret value $x_1, x_2, \ldots, x_n$, we aim to compute the sum $\sum_{i=1}^{n} x_i$ without any party learning anything about other data owner's private values, except what can be inferred from the aggregate value itself. We assume that each node has the ability to communicate with each other node (which we implement using different strategies for our two implementations, as discussed in section 3). Each node is also aware of some shared public parameter b, which is an integer we refer to as the cryptographic base. Finally, we assume that all nodes are "honest but curious" [1], meaning each participant (including the aggregator) will honestly follow the protocol but will attempt to learn information about other nodes' secret values.

Secure aggregation is achieved by having each node X_i compute a masked version of its private value such that when all masked values are summed, the result is the same as if the values themselves were summed. To compute these masked values, each node first generates a set of random integers between 0 and b which we call *perturbations*. Each node X_i computes a perturbation S_{ij} for each other node X_j and sends the perturbation to X_j.

Next, node X_i uses the perturbations it has received from all other nodes to compute its masked value. It does so by first computing values $P_{i,j} = S_{i,j} - S_{j,i} \pmod b$ using both the perturbation it sent to each other node X_j and the perturbation it received from each X_j. The masked value for node X_i is then the sum of its private value and each of these "sub-masks," modulo the cryptographic base. Formally, the masked value it sends the server is $x_i + \sum_{j \neq i} P_{i,j} \pmod b$.

The server then simply computes the sum of these masked values modulo b. The perturbations present in each masked value cancel out and the result is equal to the sum of the node's private values, thanks to the fact that $P_{i,j} = -P_{j,i} \pmod b$.

3 Design and Implementation

The high-level architectures of our two implementations of the above protocol are shown in Figure 1. The first, a cross-device implementation, utilizes a typical client-server architecture, and the second, a cross-silo implementation, uses a peer-to-peer architecture.

Both of these implementations share the same design goals, beginning with a focus on privacy above all else. Since the sole purpose of secure aggregation is to preserve data privacy, our systems should not sacrifice this paramount goal for improvement in any other area. While we would also like our systems to achieve reasonable scalability and availability, we only attempted to optimize these metrics after achieving our privacy guarantees.

Finally, we chose to accept a certain sacrifice in the reliability of our systems in order to accommodate our goal of privacy. We are aware that our systems can fail when nodes drop out midway through the protocol since we require all participants to complete all rounds of the protocol in order for the totality of their perturbations to be present in the aggregator's final sum. Significantly more complex protocols exist to handle node dropout and are mentioned in Section 5, but we found this to not be an issue in our relatively simple protocol which typically completes within a matter of seconds. With such a short duration, client dropout is less likely, and re-running the protocol is not very expensive.

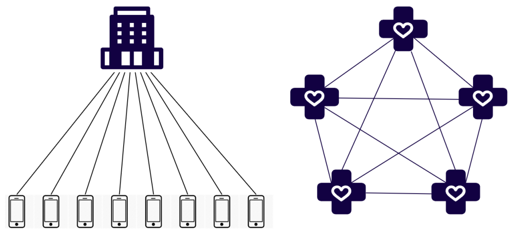
Figure 1: Left: architecture of a client-server system for cross-device federated computation. Right: architecture of a peer-to-peer system for cross-silo federated computation.

3.1 Client-Server System

The client-server system models a cross-device setting in which many clients (such as a set of smartphones running the same OS) participate in a multi-round protocol that allows a central server (such as the smartphone producer's data center) to securely compute an aggregate of the clients' local data.

To begin the protocol, we assume that each client has a public/private key pair. It is not strictly necessary for the other clients or the server to know each other's public keys since the protocol handles public key distribution, but a more robust public key infrastructure (PKI) must be assumed to avoid man-in-the-middle attacks and other issues that arise when identities cannot be linked to public keys. We also assume that the server has picked the shared cryptographic base b (which is sufficiently large relative to client data to avoid overflow issues when computing sums modulo b) and that the server has decided on a fixed number of clients n that will participate in the protocol. Note that the server need not pick the particular clients that will participate, only the count n.

Figure 2 illustrates a simplified version of the first two stages of the protocol. First, shared parameters are established when each client sends the server its public key and the server sends the shared cryptographic base to each client. After receiving all public keys, the server sends them to every client (although this step is not necessary if a complete PKI already exists).

Each client then generates its random perturbations and, as shown in Figure 2, shares these with all other clients by way of sending them to the server and having the server route them to the appropriate recipient client. Leveraging the server as a middleman allows clients to avoid setting up direct connections with one another and limits the total number of messages that must be sent. It is also useful for more complex variations of the protocol for the server to know which clients participated in this protocol round in case some dropped out. Each client encrypts its perturbations using the other clients' public keys to maintain confidentiality since knowledge of the perturbations would allow the server to compute each client's private value.

Once each client has received all perturbation messages from other clients, it can complete the protocol by computing its masked value and sending this value to the server. The server then computes the aggregate and, optionally, sends the aggregate to the clients to inform them of the result.

The initial client-server system only supported the aggregation of single integer values for each client, but it was expanded to allow for the element-wise aggregation of private client vectors. Modifying the protocol to support vector aggregation was straightforward, with the largest change being that each client computes a vector of perturbations to send to all other clients rather than a single value. These perturbations are applied element-wise to each client's vector in order to compute a vector of masked values. The total number of stages and messages sent within the protocol does not change.

The client-server system is implemented using Python, and all communication occurs over the WebSockets protocol using the websockets library. The asyncio library is used to enable concurrency, and PyCryptodome is used to encrypt messages sent by clients (in particular, asymmetric encryption using an AES session key). WebSockets was chosen in order to support a web-based demo of the system. It is a protocol built on top of TCP to allow a client (often a web browser) to interact with a server over a connection that is persistent, low-latency, and full-duplex. These features made it a good choice for a multi-round protocol that involved frequent messaging between clients and a server, often with a delay between each round as clients ran various computations or waited for the next stage to begin. Creating the web demo involved building a second implementation of the client using JavaScript, which could interoperate with the Python server by issuing calls from the browser over WebSockets.

3.2 Peer-to-Peer System

The peer-to-peer implementation models a scenario of cross-silo federated computation. In addition to the assumptions made by the protocol, we also assume the existence of a fully connected peer-to-peer network. Additionally, we assume that one node in the network is interested in aggregating the data of its peers, and once the aggregation has begun, we assume that the network is stable (i.e. no nodes join or leave).

The peer-to-peer implementation also uses Python, and it makes use of the `p2p-network` library. The library defines an abstract Node class and abstracts away how nodes connect with and send messages to each other. The library uses sockets for communication between nodes. Additionally, the library defines events, which happen on nodes. Events include `receive_message`, `inbound_node_connected`, `outbound_node_connected`, and more.

Our implementation assumes the existence of

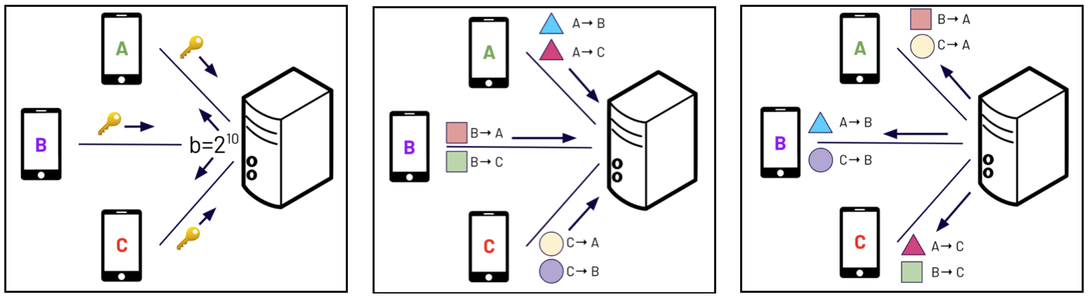

Figure 2: Two stages of the secure aggregation protocol performed by the client-server system. On the left, initial parameters are established. The other two frames show how clients exchange perturbations with one another via the server. Each perturbation is encrypted using the recipient's public key to maintain confidentiality from the server.

some fully connected peer-to-peer system. To accomplish this, we have a strict protocol for nodes joining the system. To join the system, a node must know an address of a peer in the network. The new node sends a connection request to the known network node, which triggers the `inbound_node_connected` event on the network node. This event confirms the connection between the new node and the known network node, and then sends a response to the connecting node. This response contains an acknowledgment of connection and a list of every other node in the network. Since we assume the network is always connected, every node knows about every other node and this message will contain every node in the network. When the new node receives this message, it recursively connects to all the nodes received in the acknowledgment, with the condition that it will not attempt to duplicate an existing connection.

With this method for building out a fully connected peer-to-peer network by simply adding nodes to the network and letting them take care of the rest, we are now ready to handle the aggregation protocol. As noted above, we assume that one of the peers in the network wishes to aggregate the data of its peers. The aggregator begins by sending a message to every node in the connected network. This message contains a request for aggregation and information about the aggregator's node, including its address and node ID.

Once a node receives the aggregator's message, it begins the process of computing its mask. To do so, it sends a perturbation to every node it is connected to, except for the aggregator. Each node maintains a dictionary to record the perturbations it sent to each of its peers. Each node also listens for incoming perturbations which it stores in a dictionary mapping the ID of the sender to the perturbation that it sent. Once a node has sent and received all of its perturbations, it can compute its masked value as described in the protocol specification to send to the aggregator node.

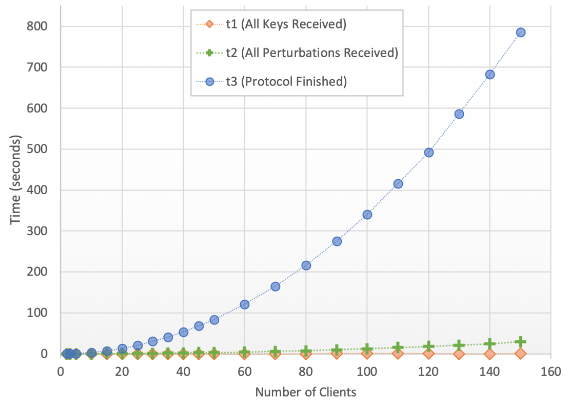

Figure 3: Duration of protocol stages for the client-server system as the number of clients increases. All clients generated a random vector of 5 values in the range [0, 1000], and $b = 10^6$ was used as the cryptographic base.

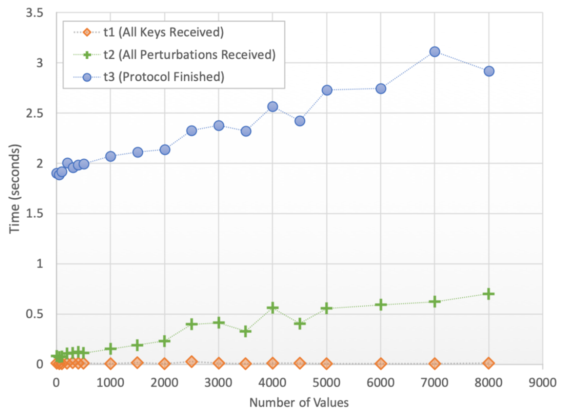

Figure 4: Duration of protocol stages for the client-server system as the number of values sent in each client vector increases. All trials involved 5 clients generating vectors with values in the range [0, 10^6], and $b = 10^{12}$ was used as the cryptographic base.

4 Evaluation

4.1 Client-Server System

The most important aspect of evaluating the client-server system was ensuring the correctness and privacy of the protocol. Both of these are binary results and could be easily confirmed by ensuring that (1) clients never sent messages that could be used to identify their private values and (2) the secure aggregation results computed by the server over the masked values matched the aggregation of the client's private values.

Further evaluation of the system consisted of two experiments for testing its scalability. Both sought to understand how the duration of the protocol increased as the quantity of data being aggregated increased, with the first examining duration as a function of the number of participating clients (holding the quantity and range of values being aggregated constant) and the second examining duration as a function of the number of values sent by each client (holding the total number of clients constant).

The evaluation was performed by running the server on the machine `limia.cs.williams.edu` and having each client run as a separate thread on a laptop connected to the same local network. This configuration was chosen since all messages in the protocol are sent between the clients and the server (i.e. there are no client-to-client messages), so all WebSocket communication was guaranteed to occur over a real network rather than within a single machine's loopback. Although the client threads may have competed for resources on the single laptop, this is actually a desirable property for evaluating the protocol since it simulates heterogeneous clients finishing stages of the protocol at slightly different times. An initial attempt at evaluation had each client generate its own RSA key-pair at the start of the protocol, but this is an extremely expensive operation relative to the rest of the protocol so the results shown represent a setup in which each client simply reads in a pre-generated keypair from a local file.

Both experiments timed the total duration of a single trial of the protocol as the sum of three stages. The first stage (and the protocol itself) was considered to begin when the first client connected to the server, and it lasted until $t1$ when the server received all public keys. The second stage then began immediately and lasted until $t2$ when the server received all perturbation messages from the clients. This stage involved each client generating its perturbation messages and encrypting them with the appropriate public keys for the recipients. $t2$ also marked the beginning of the third stage, which involved clients receiving perturbation messages from the server, decrypting the messages, computing their masked values, and sending these values to the server. The protocol finished at $t3$ when the server received all masked values and computed the aggregate. Note that $t1$, $t2$, and $t3$ are all considered relative to the start of the protocol rather than relative to each other (so, for example, $t3$ represents the entire duration of the protocol trial rather than the time between $t2$ and the end of the trial).

As shown in Figure 3, the total duration of the protocol scales quadratically with the number of clients. This behavior makes sense due to the n^2 number of perturbation messages that must be generated and shared between clients to compute their masked values. Interestingly, the vast majority of time spent in the protocol occurs between $t2$ and $t3$ even though the stage between $t1$ and $t2$ also requires clients to generate and encrypt their n^2 perturbation messages. This is likely due to the fact that

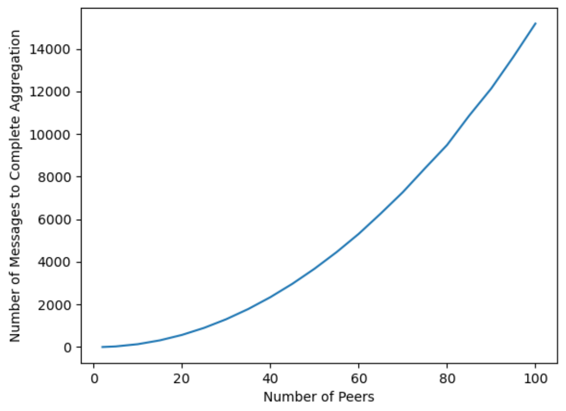

Figure 5: Number of Messages Sent in Peer-to-Peer Implementation

RSA decryption is known to take longer than encryption, and the stage between $t2$ and $t3$ requires clients to not only decrypt their received perturbation messages but also use them to compute their masked values which involves a multi-step equation with lots of modular arithmetic.

Figure 4 shows how the duration of the protocol scales with the number of values included in each client vector being aggregated. Since secure aggregation is frequently used to aggregate machine learning model weights, which could involve large integer representations of floating-point numbers and high-dimensional vectors, each client generated vectors containing values ranging from 0 to 10^6. As expected, the protocol duration appeared to grow roughly linearly with respect to the number of values in each client's vector. Increasing the number of values sent in each vector required clients to generate more perturbations and perform more arithmetic, but with the number of clients held constant, there was no n^2 increase in costly metrics such as the number of messages sent or the number of encryptions/decryptions performed. Once again, the stage between $t2$ and $t3$ took the greatest amount of time.

4.2 Peer-to-Peer System

To evaluate the performance of our peer-to-peer system, we created a network of nodes and had one peer aggregate the values of the rest of the network. We ran tests on a single machine instead of creating n nodes on n different devices since doing so was deemed unfeasible for large n. Each node was associated with the local IP address and a port on the testing machine, and all messages were passed as if they were network requests. However, modern computers can detect that they are sending messages to different ports on their machine and thus avoid routing requests onto the network. They instead handle the message sending internally, which drastically reduces the total network time for each message. Knowing this, we chose to measure the total number of messages sent rather than the wall-clock time of the protocol in order to evaluate our peer-to-peer implementation. Our results are highlighted in Figure 5.

We note that the protocol exhibits $O(n^2)$ behavior. To understand why, we note that securely aggregating across n peers requires computing $O(n)$ masks. Each mask requires a given node to send $O(n)$ messages, one to each non-aggregator peer. In total, this requires $O(n^2)$ messages to compute all masks. There is also $O(n^2)$ behavior during the initial connection process. The total wall-clock time required to perform this aggregation will thus also take roughly $O(n^2)$ time.

5 Future Work

Our two main goals for future work are to increase the reliability and efficiency of our systems. As mentioned previously, our current systems do not implement the more advanced protocols described by Bonawitz et al. [1], which are able to handle clients dropping out after the protocol has begun. In our current systems, if a client drops out, the masked values will not properly cancel and the server cannot compute the correct aggregate. Implementing the more advanced versions of the protocols introduced by Bonawitz et al. [1] would involve adding multiple additional rounds to the protocol that leverage a cryptographic secret sharing scheme, such as Shamir's Secret Sharing (SSS) algorithm. Another protocol outlined by the paper allows us to relax the "honest but curious" assumption.

Further work could also be done to improve the efficiency of our systems. Cutting-edge implementations of secure aggregation such as the one presented by Behnia et al. [3] involve even more complicated protocols than those offered by Bonawitz et al. [1], but there is room for improvement even in our relatively simple systems. For example, we would like to experiment with more efficient serialization and deserialization techniques for our messages, and we are confident that our method of encrypting client messages in the client-server system could be improved. Due to limitations of Python's type system and the underlying C libraries it uses, Python lists cannot be directly encrypted using our chosen library, so client vectors are converted to strings, encrypted, sent as a message, then decrypted and parsed back into lists. This process could certainly be streamlined using more advanced techniques.

Finally, we would like to experiment with complementary technologies such as differential privacy [4] and federated learning to see if our system could be used as a protocol layer to support entire applications or frameworks for privacy-preserving data analytics and machine learning.

6 Conclusion

Building our implementation of a secure aggregation protocol was a useful and rewarding project. We believe privacy is an undervalued and often overlooked design goal that many distributed systems either disregard or sacrifice in favor of greater simplicity, scalability, or latency. Engineering a system that optimized for privacy above all else was therefore an gratifying and instructive process. We also enjoyed the process of taking abstract ideas from a research paper and turning them into tangible code that produced a real output. Translating hypothetical theories and formulas into working programs and systems is a valuable skill, and it was extremely satisfying to see academic text come alive as an interactive demo.

Given the complexity of the math that underlies secure aggregation, we were grateful to be able to use Python for both implementations of our system. With so many open-source libraries to choose from, we were able to pick tools that provided us with enough abstraction to architect our systems without worrying about minor implementation details, yet which also provided enough access to low-level functionality that we were not restricted.

Of course, there were also challenging and frustrating aspects of the project. Implementing peer-to-peer networks can quickly become a headache, and many problems can arise when trying to manage so many connections and interactions between peers. The client-server architecture avoided some of the complications associated with the peer-to-peer system, but introduced issues of its own such as having to encrypt messages that are passed through the central server in order to confidentially communicate between clients.

Above all else, we appreciated this project for the exciting real-world implications of the topic of secure aggregation. There seems to be a tension in our modern digital society between the incredible benefits that can be realized by performing computations on massive quantities of data on the one hand, and the alarming erosion of personal privacy on the other. Secure aggregation offers a "best of both worlds" approach that lets entities perform computations on decentralized data at scale, thus allowing users to maintain their privacy (and letting the aggregating entities adhere to data-sharing laws) while still reaping the benefits of strategies such as federated learning. While these complex protocols have costs, including the added overhead of various cryptographic schemes, secure aggregation appears to be one of the rare cases in which having to make a tradeoff between two crucial goals is not necessary.

References

[1] K. A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016. URL https://arxiv.org/abs/1611.04482.

[2] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC*

conference on computer and communications security, pages 1322–1333, 2015.

[3] Rouzbeh Behnia, Mohammadreza Ebrahimi, Arman Riasi, Balaji Padmanabhan, and Thang Hoang. Efficient secure aggregation for privacy-preserving federated machine learning. *arXiv preprint arXiv:2304.03841*, 2023.

[4] Cynthia Dwork. Differential privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 1–12, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-35908-1.