Implementation of Secure Aggregation Protocol and Efficient Packed Secret Sharing

Yixin Zhang
University of Rochester

(Dated: 11 May 2025)

Packed Secret Sharing shares secrets amongst multiple parties with performance and confidentiality, while Secure Aggregation is one of its applications that enables training ML models without compromising privacy.

I. INTRODUCTION

Data is one of the fundamental building blocks of the modern world. By leveraging the use of data, enterprise, institutions, and private individuals are able to extract valuable information by various means of analysis such as Machine Learning. Hence, there is considerable motivations behind protecting the confidentiality of data while maintain its availability. This is where modern Cryptography come into place to encode data in such a way that stakeholders can still make use of it. Zero Knowledge Proofs and Secret Sharing Protocols are examples of such cryptosystems. The former proves that a user is in possession of some digital asset without revealing the content[1], while the later distributes confidential data to multiple parties such that the distributed pieces do not leak any information about the original content. Our work, which is based on Secret Sharing, is a type of Multiparty Secure Computation (MPC). MPC allows multiple parties with distributed secrets to jointly compute some functions without a mediator that accesses the secrets. Computation is done by all stakeholders together with homomorphic encryption, and thus, both confidentiality and correctness are preserved against either semi-honest or malicious adversary.

When using MPC for Machine Learning, called Federated Learning (FL), enormous amount of training data could be a performance bottleneck due to the limitation of bandwidth and computing power of each. In 2017, research of practical FL was launched in Google AI Lab for its significant application in data collected from mobile device. Google was testing their results in Gboard on Android, the Google Keyboard, which was trained with encrypted, decentralized data stored locally. Google's researcher pointed out that in a FL setting, user devices have significant latency, lower throughput due to limited internet bandwidth and are intermittently unavailable for training[2]. To address these real-world constraints, the first Secure Aggregation protocol is formulated by Bonawitz *et al* with cryptographic preliminaries. Our work, which has a similar round-base, server-client communication model, is robust againt malicious adversary by including a series of verification steps such as the $L_2 normtest$ and leveraging Packed Secret Sharing (PSS).

In this thesis, we explore the implementations of the Secure Aggregation protocol such as the design of a stateful multithreading server, setups of private channel and public broadcasting, as well as mechanism to handle dropouts within security limit. Also, we aims to uplift the bottleneck of PSS by providing algorithms to compute the optimal packing ratio and degrees. In extension of the secret sharing scheme, we also parallelize multiple instances of FFT when calculating shares of secrets, thus the performance of overall protocol is further improved. Not that theoretical justification is not concerned by this thesis; a proof of security is included in the group's publication.

II. PRELIMINARIES AND RELATED WORKS

1. Federated Learning

Our work is a protocol that aggregates data for Federated Learning, which is introduced by McMahan *et al*[3] with a model that aggregates inputs from distributed devices to minimizes global loss. With N Mobile devices and loss function $l_i(\cdot)$ where $i \in \{1,2,\ldots,N\}$, the global loss function is given by summing all local losses with weight w by:

$$L(w) = \sum_{i=1}^{N} l_i(w)$$

$\mathrm{L}_{i,S}(t)$
$\mathrm{Z}_S(t) * v_i/(t-\omega^i)$
$\mathrm{Z}_S(t) = \prod_j (t-\omega^j) = (t^m - 1)$
$\mathrm{v}_i = 1/\prod_{j\neq i}(\omega^i - \omega^j)$
$\mathrm{v}_0 = 1/mandv_{i+1} = \omega * v_i.$

For each round, mobile device i sends an update to the server and all updates are aggregated to train the global model. Upon aggregation, the server sends the updated model back to local devices to initiate the next round.

2. Multi-party Secure Computation

In MPC, parties $p_1, p_2, \ldots, p_n$ each with private inputs $x_1, x_2, \ldots, x_n$ jointly computes a function $F(x_1, x_2, \ldots, x_n)$ without leaking information about their private data. Famous MPC paradigm includes the Garble Circuits introduced by Yao *et al*[4], and GMW protocol introduced by Goldreich *et al*[5] which generalized the secure 2-party computation into MPC.

3. t-out-of-n Secret Sharing

Our work is also based on t-out-of-n secret shring that computes function f with t malicious adversary out of n participants. f tolerates a fraction of malicious as long as $n > 2t + 1$[6]. We include a variant of Shamir's Secrete Sharing called Packed Secrete Sharing which is describe in the later sections.

III. IMPLEMENTATION OF SECURE AGGREGATION PROTOCOL

The protocol runs between one server and n clients. The server acts as a coordinator who manages the state of the protocol, calculates and broadcasts public parameters, and forwards messages amongst pairs of clients. The clients are owner of secrets; they each have an input vectors of messages being converted to 64-bit unsigned integers, and want to jointly compute the sum of every party's input. Following the MPC paradigm, the protocol will secret-share all inputs with homomorphic encryption and dissipates the shares from each one party to all other parties. Dropouts are handled from each sessions of communication, then a number of tests are conducted to identify adversary in the malicious case. Finally, the server signals for aggregation, followed by a reconstruction step that recovers the aggregated result.

To accommodate for the round-base communicational model, we design a stateful server with five different states along with two messaging channels — a private channel that replies to individual client's request and a public channel for broadcast. The `Handshake` state accepts incoming TCP connection with clients, then establish a profile and generate an Elliptic Curve Digital Signature (ECDSA) key pair for each client and sends out the signing keys. The `Key Exchange` state performs Elliptic-curve Diffie–Hellman (ECDH) key exchange by broadcasting the list of all public keys received from clients, so that clients will each establishes pairwise ECDH shared keys. After establishing the key infrastructure, the server will derive the optimal parameters for PSS based on the input size, number of parties, and the security tolerance of dropouts and corruption. In `Input Sharing` state, every participant runs PSS algorithm with these parameters — they will divide their input vector into blocks, then compute shares of blocks for each other peer. They send these shares to the server with AES-GCM encryption using the DH shared keys, then the server will forward these shares to their peers.

Our protocol diverges into the Semi-Honest case and the Malicious case after input sharing. For the Semi-Honest implementation, we only consider clients dropping out from the protocol. The server will enters the `Aggregation` state which broadcasts a list dropout parties, and the clients will aggregated only the shares of those who remains in the session. While for the Malicious case, we include a state of `Error Correction` which identify dropouts and corrupted parties. The server will broadcasts another set of parameters for a number of error correction tests, and requires clients to compute for specify results. By evaluating the results, the server then broadcasts the list of malicious parties in addition to dropouts. Finally, the protocol enter the `Aggregation` state and the server will reconstruct for the summation of all input vectors.

Overall, in order to ensure the security and performance while handling large size input and significant amount of clients, we need a robust architecture with sophisticated implementation design. For example, we consider how to implement a multi-threaded server that handles queries of clients in parallel, how to maintain a public key infrastructure that sustain a seamless encryption-decryption process, how to calculate the optimal sharing parameters to optimize the performance of DFTs, and how to efficiently reconstruct secrets when dropout and corruption occurs.

We further attempt to parallelize the PSS scheme with GPU acceleration, and our results shows that parallelizing will significantly improve the performance with large input size and smaller block size when facing certain constraint with security parameters.

1. Multi-threaded Server Architecture and Messaging Framework

The server consists of a state controller, concurrent workers and an underlying messaging framework powered ZeroMQ. ZeroMQ is an embedded networking library which provides N-to-N with connection patterns such as task distribution, request-reply, and publisher-subscriber[7]; in its concurrent topology, we employed a load balancing broker model with one broker and a set of concurrent workers. The broker and workers are connected with inter-thread communication transport, while clients connects with the broker through one single TCP port. Note that peer-to-peer conversation amongst clients is not allowed, the clients interact with the server in a request-respond basis and receive broadcast by subscribing to the server's publisher channel.

For the stateful server, a typical round of communication proceed as follows:

1. The server enters a new state, accepting incoming message from its frontend TCP port; each message is forwarded by the broker to one of the workers to process;
2. Upon receiving a message, the worker performs sanity checks on the client ID and format of the message; if pass, the worker process the message and replies OK;
3. The worker signals success to broker, while broker counts number of success communications and keeps track of time elapse since the start of current state; it initiates state change as soon as the count of successes reaches some maximum value or timer expires;
4. Broker broadcasts parameters needed for the next round of communication and finally incitement the state.

All data are being kept in the server data structure and protected with mutex. When workers make changes to data, for example inserting new client ID, creating ECDSA key pairs, or deleting profiles from dropouts, the server acts like a concurrent database.

Overall, this architecture provides the flexibility to accept connections from any parties and the security to abort malicious requests. Since we do not maintain any persisting private connections, message-based communication becomes easy to handle, and clients are able to join and dropout at any time. Parties with limited computing power can perform slower computation offline, then sends back the result before the server enters the next state. Enough time will be allowed between states, hence the protocol is asynchronous for computation and synchronous for communication rounds.

2. *Key Infrastructure and Encrypted Channel*

Key Exchange schemes are the fundamental building blocks of today's world wide web. For example, in SSL/TSL, servers who are certified by a Certificate Authority (CA) provide public keys for incoming clients to establish shared session keys[8]. In this implementation, however, clients each needs a public key for digital signature and to create pairwise secure channels. Since it's unpractical to include CA to certify participants, we leave key management to the server entirely.

When accepting a new client, the server will generate a ECDSA key pair and send it to the client. Although the handshake process is susceptible to man-in-the-middle-attack, the vulnerability before key establishment is not of our concern; we will recognize any party by its ID and a valid ECDSA signature that matches the one generated by server. The client then generate a ECDH public key encrypted by ECDSA private key and send to server. At this point, we will perform the classic Deffie-Hellman (DH) key exchange, except that the server will collect all DH public keys and broadcast the entire list to all participants. In that later `Input Sharing` stage, the most important criteria is the authenticity and integrity of messages when clients are sending shares to every other parties through the server. These pairwise DH shared keys will create end-to-end encrypted channels against semi-honest server.

We use the Elliptic Curve P-256 for both ECDSA and ECDH standardized by National Institute of Standards and Technology (NIST)[9]. This curve provides 128-bits of security and it's commonly used in the internet infrastructure including the Border Gateway Protocol (BGP)[10]. NIST prime curves are expressed by the following equation:

$$y^n = x^3 - 3x + b \bmod p$$

where prime p is:

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$

After generation of shared keys, we use AES-GCM create end-to-end encryption between each pairs of clients. The initialization vector of AES-GCM, which usually serves as a nonce or session key in other protocols[11], is set to be the ECDSA public key of sender in encrypted channels.

3. *Dropouts and Error Corrections*

Between rounds of communications, clients may dropout of the protocol due to the instability of real-world network connection. Some parties may be honest-but-curious or even corrupted - they may attempt to attain extra information or disrupt the on-going protocol by deviating from the specified operations. Thus, a practical protocol needs to account for dropouts and corruption.

a. Semi-Honest Setting: adversary attempts to attain extra information. This Protocol is inherently robust against semi-honest clients since t-out-of-n secret sharing distribute secrets to n parties such that information would not be leaked unless t parties join to reveal secrets. This means that, t out of n participants have to be corrupted in order to hijack the secrecy. Thus, when corruptions are under the limit set by the security parameters, we only consider dropouts to ensure that server is able recover the aggregated results. At the end of each states, dropout clients are being identify if their reply is not received before timeout. For example, after the server sends a client its ECDSA private key in `Handshake`, server will enter the `Key Exchange` state, expecting this client to send its ECDH public key within certain time frame. If the client fails to do so, its profile will be erased from the concurrent database.

Once `Key Exchange` is over, the server will determine the secret sharing parameters based on the current number of clients, the input length, as well as the tolerance of dropouts and corruption. No more changes will happen to the profile list because a fixed evaluation point has been assigned for each client. Later on, we will reconstruct the result based on these evaluation point, hence it's necessary to know which corresponding point to skip if a client dropouts. Hence for `Input Sharing`, server will broadcast the sharing parameters and expect clients to send their shares back, if a client fails to do so, its position in the sharing matrix will be left empty. For `Reconstruction`, a final set of dropouts and corrupted parties will be identified and published, all clients will then aggregate only the shares of qualified parties. Performance wise, if we have unwanted data in the sharing matrix due dropouts, leaving the entry blank is faster than resizing the data structure.

b. Malicious Setting: adversary disrupts the protocol. We include an extra `Error Correction` state to identify malicious parties who sends disruptive messages. Eight tests will be perform to assess whether a client has sent to actual shares of its input computed in the desirable manner. These tests are Degree test, Input Bit test, Quadratic test, Input Bound test, L2-Norm Sum test, L2-Norm Bit test, and L2-Norm Bound test which based on the mathematical properties of PSS polynomials and input vector. To perform the test, the server will broadcast a set of tests parameters, and clients will run specified computation with each peer's shares with these parameters, then respond with the results. In general, our stateful implementation provides significant advantage for the server to include or exclude additional verification steps; if more tests need to be performed, we will add them to the `Error Correction` module. It is also simple to switch between the semi-honest mode and malicious mode since the server can disable the `Error Correction` state flexibly.

IV. EFFICIENT PACKED SECRET SHARING AND PARALLELIZATION

Packed Secret Sharing is the building block of this protocol, and an efficient implementation of t-out-of-n PSS is crucial to the overall performance. In our setting, a party needs to share its input vector of length m with n parties including itself, and t parties will have to remain in the protocol for successful reconstruction. This minimum number t is called threshold of the scheme, and PSS is an variant of Adi Shamir's threshold scheme called Shamir's Secret Sharing.

The essential idea of Shamir's Secret Sharing is to define degree t-1 polynomial with t discrete points. For example, we can define a straight line with 2 points, parabola with 3 points, and cubic curve with 4 points in a Cartesian plain. Suppose we have secret S sharing with n parties, we build the polynomial

$$f(x) = s_0 + s_1 x + s_2 x^2 + \cdots + s_{t-1} x^{t-1}$$

$a_0, a_1, \ldots, a_{k-1}$ are elements of finite field $\mathbb{F}_P$ where P is a large prime, and $a_0 = S$ which encodes the secret itself. We then take n evaluations of the polynomial at points $i = 1, 2, \ldots, n$ to retrieve $(i, f(i))$ and distribute them to n parties. In this case, one of the individuals knows the value of $S = a_0$ but if t parties gather, their points can uniquely define the original polynomial, hence uncovering the secret.

While Shamir provides an idealized scheme to share one secret which has to be an element of $\mathbb{F}_P$, we need to distribute the input vector for Federated Learning which will includes up to 10,000 large integers. This is why we adopt PSS to subdivide the input vector into blocks, and transform the blocks into multiple polynomials then take evaluations with DFT. In particular, we have input vector $[s_0, s_1, \ldots, s_m]$ with packing ratio b, then the vector will be divided into the following $b \times d_2$ matrix where d_2 is the degree of some polynomials:

$$\begin{bmatrix} s_{00} & s_{01} & \cdots & s_{0l} \\ s_{10} & s_{11} & \cdots & s_{1l} \\ \cdots \\ s_{b0} & s_{b1} & \cdots & s_{bl} \end{bmatrix}$$

$$\begin{bmatrix} s_{00} & s_{01} & \cdots & s_{0l} & r_{0(1+1)} & r_{0(1+2)} & \cdots & r_{0d_2} \\ s_{10} & s_{11} & \cdots & s_{1l} & r_{1(1+1)} & r_{1(1+2)} & \cdots & r_{1d_2} \\ \cdots \\ s_{b0} & s_{b1} & \cdots & s_{bl} & r_{b(1+1)} & r_{b(1+2)} & \cdots & r_{bd_2} \end{bmatrix}$$

Note that $l = m/b$ is the number of integer in a block and $l < d_2$. Namely, we pick a degree d_2 and extend the blocks with random values r_{ij} to length d_2, and each row are treated as the evaluation of polynomial $p_i(\cdot)$ at points $[w_0, w_1, \ldots, w_{d_2}]$, in which w_j are j-th powers of d_2-th primitive roots of unity in $\mathbb{F}_P$. Now we have a sets of evaluation (w_j, s_{ij}), we can use inverse radix2-DFT to construct polynomial $p_i(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{d_2-1} x^{d_2-1}$. The DFT over finite field $\mathbb{F}_P$ is a generalization of FFT also known as Number Theoretic Transform; it enables us to evaluate polynomial fast with inputs as large as 128-bit unsigned integer.

To generates the shares for n parties, we make evaluations again at n points that are different than the d_2 points at which we define the polynomial. We pick another degree d_3 such that $d_3 > n > d_2$ and pad the polynomial with zeros to length d_3 to get the following matrix:

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0d_2} \\ a_{10} & a_{11} & \cdots & a_{1d_2} \\ \cdots \\ a_{b0} & a_{b1} & \cdots & a_{bd_2} \end{bmatrix}$$

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0d_2} & 0 & \cdots & 0 \\ a_{10} & a_{11} & \cdots & a_{1d_2} & 0 & \cdots & 0 \\ \cdots \\ a_{b0} & a_{b1} & \cdots & a_{bd_2} & 0 & \cdots & 0 \end{bmatrix}$$

We then perform radix-3 DFT to evaluate the polynomial at points $[v_0, v_1, \ldots, v_{d_3}]$, in which w_j are j-th powers of d_3-th primitive roots of unity in $\mathbb{F}_P$. The result of the transformation will be a $b \times d_3$ matrix, however only the first n columns are needed since the scheme aims share with n parties. Hence we return the following $b \times n$ shares:

$$\begin{bmatrix} y_{00} & y_{01} & \cdots & y_{0n} \\ y_{10} & y_{11} & \cdots & y_{1n} \\ \cdots \\ y_{b0} & y_{b1} & \cdots & y_{bn} \end{bmatrix}$$

At this point, each column vector $[y_{0j}, y_{1j}, \ldots, y_{bj}]$ will be send to a party Pj; note that each y_{ij} is indistinguishable from random values, but when $t = d_2 + 1$ parties gather with their shares, they will be able to recover each $p_i(\cdot)$. Therefore, we have successfully computed t-out-of-n PSS.

1. *Radix-2 and Radix-3 DFT with Optimal Parameters*

We use the Cooley–Tukey algorithm to comput Radix-2 and Radix-3 DFTs which uses divide-and-conquer technique. The commonly used radix-2 variant divides a DFT of size N into two interleaved DFTs of size N/2, hence the name "radix-2"; and the radix-3 variant would divide into DFTs of size N/3. Due to the nature of how it divides, radix-2 DFT requires evaluation points at d_2-th primitive roots of unity, where d_2 must be a power of two; on the other hand d_3 must be a power of three. This implementation choice is rather sophisticated because it allow us to construct the polynomials and evalute the polynomials at to sets of distinct points. In general, as long as integer a and b are relative primes, a-th primitive roots and b-th primitive roots will be disjoint.

However, this method involves inserting random values before construction and truncating extra shares after evaluation, which will waste some of the computational power. To satisfied our DFT paradigm while the minimizing computational waste, we must pack our input vector into feasible block length l, and select the optimal value of d_2 and d_3. Let c be the tolerance of corruptions, d be the tolerance of dropouts, and n be number of parties, these parameters first also need to satisfy the following constraints for the sake of security:

$$d_2 \geq l + c \tag{1}$$

$$d_2 \leq n - (2c + d) \tag{2}$$

$$d_3 \geq n \tag{3}$$

Eq.(1) ensures that c corrupted parties togeter cannot recover the degree d_2 polynomial; Eq.(2) allows the secrets to be reconstructed after excluding c corrupted parties and d dropouts; Eq.(3) leave us enough shares to distribute. Let m be the input vector length, we first find the greatest power of two that is smaller than $n - (2c + d)$, then calculate the greatest divider of m that is smaller than $d_2 - c$. For d_3, we find the smallest power of three which is greater than n. In this way, we are inserting the minimum amount of randomness $d_2 - l$, while still make sure that it is enough to protect the polynomial against c corrupted parties; We also have enough shares to distribute without truncating excessive amount of extra values out of radix-3 DFT.

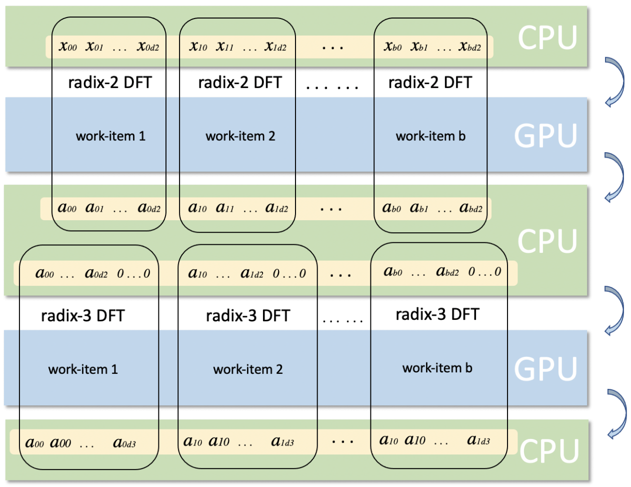
2. *Parallelization of Packed Secret Sharing Scheme on GPU*

As we examine the PSS shceme, we believe that PSS can be parallelized since different blocks of input vector can be transform and evaluated independently, thus we can run multiple instance of DFTs in concurrent threads. We initiate an experimental attempt to adapt the PSS program on GPU by executing the radix-2 DFT and radix-3 DFT with two GPU kernels.

We use OpenCL as the parallel programming language, which characterizes the memory hierarchy of the computing device into four level; In global memory, data are shared by all threads with high latency; in constant memory, available size is smaller but latency is lower; in local memory, only threads in the same work-group can share the access[12]. The work-group could be an abstraction of GPU thread blocks which shares the same L1 Cache that allows more synchronization, thus the size of work-group is important for better performance.

Since PSS runs b instances of radix-2 DFTs without synchronization, we set the group size equal to the Nvidia GPU warp size, and run one DFT per kernel, while the prime P and the powers of roots in finite field $\mathbb{F}_P$ are passed into the global memory. Particularly, the inverse radix-2 DFTs of length d_2 were first calculated in-place, and we transfer the result back to the CPU. We then extend each polynomial with zeros to length d_3, and perform another data transfer to run b instances of radix-3 DFTs (Fig. 1).

We used a NVIDIA Tesla T4 GPU for parallelized PSS and a Intel Cascade Lake CPU for sequential processing. As a result, we were able to accelerate the performance for roughly 15 times.



x_{00} x_{01} ... x_{0d2}
x_{10} x_{11} ... x_{1d2}
x_{b0} x_{b1} ... x_{bd2}
CPU
radix-2 DFT
radix-2 DFT
radix-2 DFT
work-item 1
work-item 2
work-item b
GPU
a_{00} a_{01} ... a_{0d2}
a_{10} a_{11} ... a_{1d2}
a_{b0} a_{b1} ... a_{bd2}
CPU
a_{00} ... a_{0d2} 0...0
a_{10} ... a_{1d2} 0...0
a_{b0} ... a_{bd2} 0...0
radix-3 DFT
radix-3 DFT
radix-3 DFT
work-item 1
work-item 2
work-item b
GPU
a_{00} $a00$ ... a_{0d3}
a_{10} $a10$ ... a_{1d3}
a_{10} $a10$ ... a_{1d3}
CPU

FIG. 1: Demonstration of work division and data transfer of the paralleled PSS. Each work-item executes one instance of DFT, and there are 2 round-trip data transfers in total between the GPU and GPU.

V. RESULTS

1. *Performance and Communication Overhead of Semi-Honest Secure Aggregation Protocol*

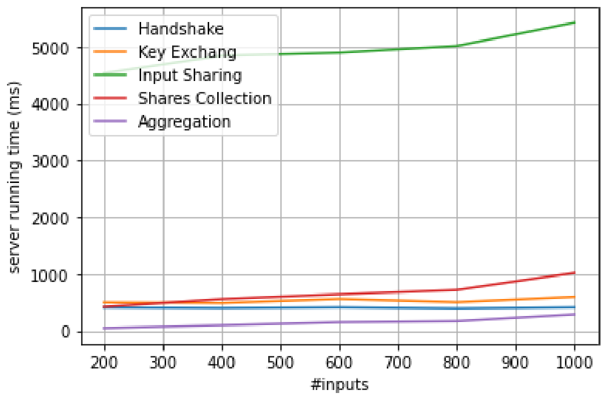
We measure the wall-clock running time and communication overhead for the semi-honest protocol(Table. I). Each state of the server(Fig. 3) and the client (Fig. 3)are measured except for the `Error Correction` since it is necessary only in the malicious setting. The simulation is done with a 2.5 GHz Cascade Lake 24C CPU on a Linux workstation. Unfortunately, due to our limitation of computing resources, all programs are ran simultaneously on the same machine by multiprocessing; This is drastically different from the real-world setting and introduces exponential latency as we increase the instances of client. A more precise performance evaluation will be provided in our group's publication.

2. *Benchmark of Sequential and Parallel Packed Secret Sharing*

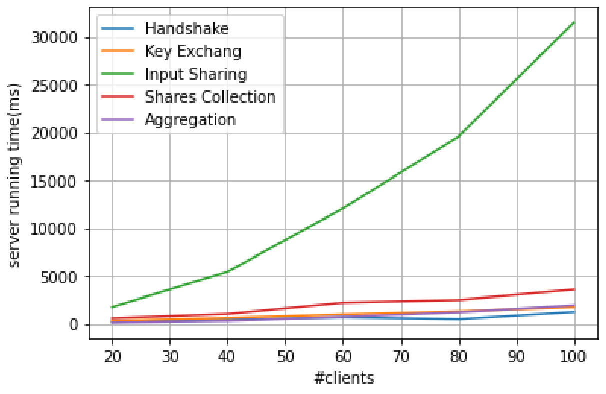
We compare the running time of radix-2 DFT (Fig. 4a) and radix-3 DFT (Fig. 4b) separately on a 2.5 GHz Cascade Lake 24C CPU and a NVIDIA T4 Tensor Core GPU. Our results shows that paralleled PSS has constantly better performance than its sequential counterpart. For the radix-2 DFT, we raise the input size and measure with respect to different different block size, and with input size of 20,000 the paralleled version is on average 11.33 times faster, as we raise the input size to 80,000, we get a improvement of 28.09 times (Table II). For radix-3 DFT the performance difference is roughly 4 times at blocks number of 40, and it widens to 32 times with blocks number equals 1600.

TABLE I: Client and Server running time for each state measured on the same machine. Latency caused by waiting for other clients has been subtracted, and the running time is averaged amongst all clients.

	#Clients	#Inputs	Handshake	Key Exchang	Input Sharing	Shares Collection	Aggregation	Total
Client	40	200	62.525 ms	2785.85 ms	436.575 ms	410.55 ms	43.6 ms	3782.7 ms
		600	55.0 ms	2762.125 ms	790.8 ms	657.2 ms	61.225 ms	4387.57 ms
		1000	56.875 ms	2841.575 ms	1372.825 ms	1022.525 ms	60.55 ms	5414.9 ms
	1000	200	28.79 ms	16446.39 ms	2093.33 ms	1990.78 ms	52.19 ms	20663.67 ms
		600	29.54 ms	16451.39 ms	3111.2 ms	2524.84 ms	57.15 ms	22231.27 ms
		1000	30.17 ms	16372.33 ms	5163.57 ms	3784.2 ms	151.39 ms	25653.05 ms
Server	40	200	410 ms	500 ms	4533 ms	423 ms	46 ms	5912 ms
		600	413 ms	560 ms	4892 ms	642 ms	6658 ms	13165 ms
		1000	413 ms	597 ms	5422 ms	1019 ms	288 ms	7739 ms
	100	200	1224 ms	1705 ms	28173 ms	1869 ms	331 ms	33302 ms
		600	1199 ms	1759 ms	29165 ms	2248 ms	863 ms	35234 ms
		1000	1227 ms	1735 ms	31472 ms	3572 ms	1921 ms	39927 ms

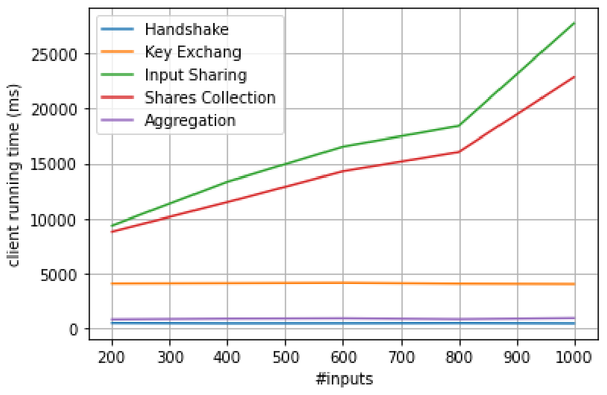


(a) Running time of server, as input size increase

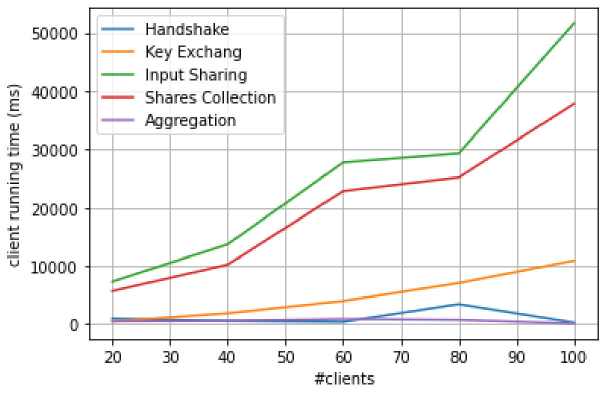


(b) Running time of server, as total number of clients increase

FIG. 2: Server running time in the semi-honest protocol. Measured in a multi-processing setting.

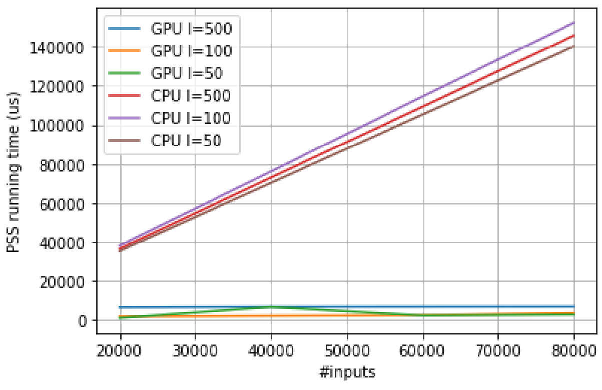


(a) Running time per client, as input size increase

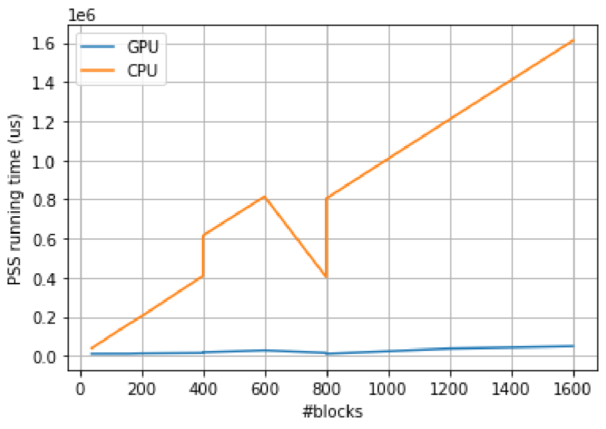


(b) Running time per client, as total number of clients increase

FIG. 3: Clients running time in the semi-honest protocol. Measured in a multi-processing setting.



(a) radix-2 DFT, measured against increasing input size



(b) radix-3 DFT, measured against increasing number of block

FIG. 4: Running time of PSS on GPU and CPU, as input size increases.

TABLE II: Running time of paralleled and sequential PSS on GPU and CPU, as input size increase dividing into different number of blocks.

			GPU		CPU			
#inputs	#blocks	block size	radix-2 DFT	radix-3 DFT	radix-2 DFT	radix-3 DFT	radix-2 Ratio	radix-3 Ratio
	40	500	66.27 ms	116.86 ms	364.11 ms	410.18 ms	5.494	3.51
20000	200	100	19.08 ms	138.75 ms	380.79 ms	2021.93 ms	19.95	14.57
	400	50	11.44 ms	166.68 ms	352.43 ms	4038.77 ms	30.80	24.23
	80	500	68.34 ms	118.9 ms	729.34 ms	818.1 ms	10.67	6.88
40000	400	100	22.61 ms	167.24 ms	760.15 ms	4093.04 ms	33.62	24.47
	800	50	18.52 ms	297.2 ms	702.27 ms	8046.04 ms	37.91	27.07
	120	500	68.93 ms	120.32 ms	1091.81 ms	1239.7 ms	15.83	10.30
60000	600	100	25.69 ms	197.22 ms	1144.19 ms	6155.17 ms	44.53	31.20
	1200	50	24.03 ms	384.44 ms	1051.93 ms	12091.13 ms	43.77	31.45
	160	500	69.36 ms	120.7 ms	1456.47 ms	1635.32 ms	20.99	13.54
80000	800	100	36.78 ms	297.86 ms	1522.24 ms	8151.86 ms	41.38	27.36
	1600	50	28.8 ms	503.12 ms	1400.85 ms	16110.2 ms	48.64	32.02

VI. DISCUSSION

Our results of the Secure Aggregation protocol aligns with our theoretical prediction and demonstrates the effectiveness of the implementation. In the case of client, the running time of `Input Sharing` and `Shares Collection` increase when both number of inputs and clients increase, while the other states that are communication overheads of the protocol remains fairly constant. `Key Exchange` increases as well when there are more number of peer's keys to process. This aligns with our model since larger input size result in larger sharing matrix for PSS, while more parties will result in more number of shares to compute. Overall, the main computation cost for the client is `Input Sharing`; the main communication cost is `Shares Collection` and partially in `Key Exchange`.

In the case of server, all states' running times increase against both input size and number of clients. However, `Input Sharing` grow more rapidly and appears to be exponential as number of clients increase; interestingly, it is not growing as significant while plotting against input size. Hence we suspect that the latency is due to multi-processing overhead. Overall, the main cost of time complexity of the server is `Input Sharing` because the server is forwarding shares between each pair of clients.

Our results of the PSS also indicate that paralelization significantly enhance the performance especially for large input size and more number of blocks. A larger block size l translates to larger instances of DFT, which has a time complexity of $O(llog(l))$. For the sequential PSS, a overall time complexity, considering the number of blocks $b = m/l$, would be $O(b*llog(l)) = O(mlog(l))$, thus the CPU version should perform the best when we have small block size. Nevertheless, block sizes of 50 gives us the best performance, following by block size of 500. With the GPU however, we do see more improvement as number of blocks (i.e. number of treads) increase for the same input size; and the paralleled PSS has a theoretical complexity of only $O(llog(l))$ since b blocks are executed concurrently.

For radix-3 DFT however, we plotted the result against number of blocks since the time complexity is $O(b) = O(m/l)$ with fixed number of shares of 729. And our result does show a roughly linear increase for bothe the CPU and GPU version. For small input size w ith small block size, the GPU is about 4 times faster given that it's under-utilized; For 80,000 inputs with 1600 blocks (threads) however, the performance of GPU is 48 times better. We will work on parallelizing more cryptographic transform algorithms with in our future works.

VII. CONCLUSION

In this thesis, we provided a efficient implementation of a Secure Aggregation protocol as well as the paralleled and sequential PSS scheme. We designed a multi-threaded server architecture and messaging framework which tolerates dropouts and malicious adversary. We handled dropouts and adversary by including multiple states and providing a mechanism that calculates optimal parameters for secret sharing.

With limited computing resource, our work securely aggregates 100 clients' input vector with length 1000 in 39.92 seconds when running all server and client programs on the same machine. We also implemented the t-out-of-n secret sharing scheme that is essential to the protocol. After prallelizing the scheme, we improved its performance for roughly 40 times with large input size.

VIII. REFERENCES

[1]O. Goldreich and Y. Oren, "Definitions and properties of zero-knowledge proof systems," **7** (1994).

[2]B. McMahan and D. Ramages, "Federated learning: Collaborative machine learning without centralized training data," (2017).

[3]J. Konecný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," CoRR **abs/1610.02527** (2016), arXiv:1610.02527.

[4]A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)* (1986) pp. 162–167.

[5]O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game, or a completeness theorem for protocols with honest majority," in *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali* (Association for Computing Machinery, New York, NY, USA, 2019) p. 307–328.

[6]M. Franklin and M. Yung, "Communication complexity of secure computation (extended abstract)," (Association for Computing Machinery, New York, NY, USA, 1992).

[7]. T. Z. authors, "Zeromq: An open-source universal messaging library," (2021).

[8]DigitCert, "What is ssl, tls and https?" (2021).

[9]NIST, "Framework for improving critical infrastructure cybersecurity," (NIST, 2014).

[10]M. Adalier and A. Teknik, "Efficient and secure elliptic curve cryptography implementation of curve p-256," (2015).

[11]J. Viega, "The use of galois/counter mode (gcm) in ipsec encapsulating security payload (esp)," (Cisco Systems, Inc, 2005).

[12]T. K. Group, "The opencl™ specification," (2019).

[13]K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," (2017).