MPC Validation and Aggregation of Unit Vectors

UCLA CSD Technical Report No. 170001

Dylan Gray   Joshua Joy   Mario Gerla

***Abstract*—When dealing with privatized data, it is important to be able to protect against malformed user inputs. This becomes difficult in MPC systems as each server should not contain enough information to know what values any user has submitted. In this paper, we implement an MPC technique to verify blinded user inputs are unit vectors. In addition, we introduce a BGW circuit which can securely aggregate the blinded inputs while only releasing the result when it is above a public threshold. These distributed techniques take as input a unit vector. While this initially seems limiting compared to real number input, it is quite powerful for cases such as selecting from a list of options, indicating a location from a set of possibilities, or any system which uses one-hot encoding.**

I. Introduction

In our system, a user would like to input an N long unit vector; however, no strict subset of the servers should be able to recreate user inputs. The servers should then be able to verify that the user's input was a unit vector. In addition, the servers should then be able to aggregate all user inputs and release the value only if it is above a public threshold. The verification circuit is an implementation of the MPC protocol described in [1]. This scheme takes input vectors, blinds (randomizes) the values, and then splits each value across all verification servers. The result is that no strict subset of the servers which collaborate can distinguish the inputs from random. However, all servers working together can verify that an input is a unit vector without revealing any other information about the input. Finally, the servers can combine all user inputs to perform an aggregate on each element of the inputs by using the BGW circuit shown in Figure 1. In other words, let N be the input length and ℓ be the number of user inputs. This data can be represented as an ℓxN matrix where each row is a user input, and the aggregation is a sum on each column, thus giving N aggregation results. We will now discuss the implementation of the verification and aggregation circuits as well as their performance.

II. Verification

Assume a user wants to submit an N long vector $V = [v_1, \ \ldots, \ v_N]$, and we want to verify that it is a unit vector using P parties without any party learning the user's vector input. The verification scheme is as follows:

1) The user splits V into a PxN matrix ($Split$), such that $\sum_i Split_{ij} = v_j$.

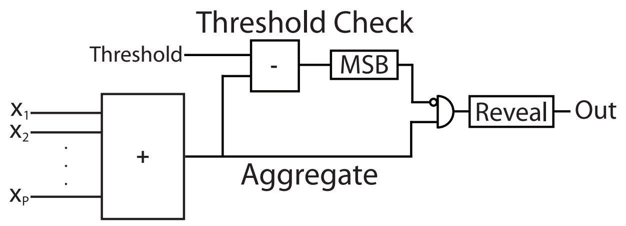

Fig. 1. BGW Multiparty Aggregation Circuit

A multiparty computation circuit which uses BGW and Shamir's Secret Sharing Algorithm to compute the aggreagate of all inputs x_i and only release it if it is above the publicly known threshold.

2) The user blinds *Split* into a PxP matrix *Blind* using one of the three algorithms discussed in Section II-B.

3) The user sends one Px1 column vector

$$share_i = \begin{bmatrix} share_{i,1} \\ \vdots \\ share_{i,P} \end{bmatrix} = \begin{bmatrix} blind_{1,i} \\ \vdots \\ blind_{P,i} \end{bmatrix}$$

from *Blind* to each verifier.

4) Verifiers perform an MPC sum on each row of the shares to get the Px1 vector

$$Sums = \begin{bmatrix} sum_1 \\ \vdots \\ sum_P \end{bmatrix}$$

where

$$sum_j = \sum_i share_{i,j}$$

5) Validation functions are performed on *Sums* to check *V* was was a unit vector.

It is important to note that all operations are done in a finite field with a prime size which is randomly chosen. The recommended size is at least 256 bits, and the performance impacts of differnet field sizes are discussed in Section IV-A.

A. Splitting

The user starts with a unit vector $V = [v_1, \ \ldots, \ v_N]$. For each v_j, $j \in \{1, \ \ldots, \ N\}$ the user generates random values $split_{i,j}$, $i \in \{1, \ \ldots, \ P-1\}$. Then they set

$split_{p,j} = v_j - \sum_i split_{i,j}$. After this is done for all N rows, the result is *Split*, a PxN matrix where $\sum_i Split_{i,j} = v_j$.

B. Blinding

There are 3 functions defined in [1] for blinding which we implemented: $Square$, $Product$, and $Inverse$. Let $\mathcal{L}$ be a PxN matrix used to blind *Split*, and *r* is a PxN matrix of random values. After the $\mathcal{L}$ matrix is generated, we perform *Blind* = $\mathcal{L}$**Split* to calculate our blinded PxP matrix. It is important to note that all calculations are done in a finite field of prime size. The constructions of the $\mathcal{L}$ matrix for the various blinding techniques are below:

$Square$:

$$\mathcal{L} \in \mathbf{F}^{PxN}$$
$$\mathcal{L}_{1,j} = r_{1,j} \ \forall j \text{ s.t. } 0 < j \leq N$$
$$\mathcal{L}_{i,j} = r_{1,j}^{i} \ \forall i,j \text{ s.t. } 1 < i \leq P,\ 0 < j \leq N$$

The result is a matrix

$$\mathcal{L}_{sq} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,N} \\ r_{1,1}^2 & r_{1,2}^2 & \cdots & r_{1,N}^2 \\ \vdots & & & \vdots \\ r_{1,1}^P & r_{1,2}^P & \cdots & r_{1,N}^P \end{bmatrix}$$

$Product$:

$$\mathcal{L} \in \mathbf{F}^{PxN}$$
$$\mathcal{L}_{i,j} = r_{i,j} \ \forall i,j \text{ s.t. } 0 < i < P,\ 0 < j \leq N$$
$$\mathcal{L}_{P,j} = \prod_{i=1}^{P-1} r_{i,j} \ \forall j \text{ s.t. },\ 0 < j \leq N$$

The result is a matrix

$$\mathcal{L}_{prod} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,N} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,N} \\ \vdots & & & \vdots \\ \prod_i r_{i,1} & \prod_i r_{i,2} & \cdots & \prod_i r_{i,N} \end{bmatrix}$$

$Inverse$:

$$\mathcal{L} \in \mathbf{F}^{PxN}$$
$$\mathcal{L}_{i,j} = r_{i,j} \ \forall i,j \text{ s.t. } 0 < i < P,\ 0 < j \leq N$$
$$\mathcal{L}_{P,j} = \left[\prod_{i=1}^{P-1} r_{i,j}\right]^{-1} \ \forall j \text{ s.t. },\ 0 < j \leq N$$

The result is a matrix

$$\mathcal{L}_{inv} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,N} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,N} \\ \vdots & & & \vdots \\ \left[\prod_i r_{i,1}\right]^{-1} & \left[\prod_i r_{i,2}\right]^{-1} & \cdots & \left[\prod_i r_{i,N}\right]^{-1} \end{bmatrix}$$

C. Sharing

After the user calculates *Blind*, they split it into P, Px1 column vectors

$$share_i = \begin{bmatrix} share_{i,1} \\ \vdots \\ share_{i,P} \end{bmatrix} = \begin{bmatrix} blind_{1,i} \\ \vdots \\ blind_{P,i} \end{bmatrix}$$

The client sends one $share_i$ to each verifier. Before the verifiers can check if the input vector is valid, they must first sum their shares into a Px1 vector

$$Sums = \begin{bmatrix} sum_1 \\ \vdots \\ sum_P \end{bmatrix}$$

where

$$sum_j = \sum_i share_{i,j}$$

This is acheived through any MPC summation scheme which does not reveal any information but the final value. Every verifier will calculate the same $Sums$ vector.

D. Validation

Based on the method chosen for blinding, there are 3 ways to verify that the original input vector was a unit vector [1]. $Sums$ is the Px1 vector which is the result of the MPC sum done by the verifiers. To verify, simply check the relation below holds for the chosen blinding method.

$Square$:

$$Sums_1^i = Sums_i \quad \forall i,\ 0 < i \leq P$$

$Product$:

$$Sums_P = \prod_{i=1}^{P-1} Sums_i$$

$Inverse$:

$$\prod_{i=1}^{P} Sums_i = 1$$

E. Security

At no point does any individual have enough information to reconstruct a user's input. In addition, after the user splits their value into P shares, any subset of those P shares appears random (as each share is a randomly generated bit-array). This ensures that as long as there is at least one honest party, security holds. However, for correctness to hold, all parties must act in an honestly-but-curious way.

III. BGW Aggregation

Once all users' inputs are validated, it is convenient to be able to aggregate the initial unit vectors. However, we wish for the servers to never be able to reconstruct any individual's input. We will now describe a BGW circuit which returns a sum per entry in the Nx1 input vectors across all users' inputs, and each entry's result is only released if it is above a publicly known threshold. In other words, all ℓ inputs can be thought of as an ℓxN matrix where each row is a user's input, and we will perform a sum on each column. Our BGW circuit can be seen in Figure 1.

First, each user performs a sum on the blinded values they received from all users. Then, they use Shamir's Secret Sharing Algorithm [2] to create shares and distribute one to each participating server. These are the inputs to the BGW circuit in Figure 1. Next, each server locally computes the sum of shares (sum) and subtracts the publicly known threshold ($thresh$), to get $dif = sum - thresh$. Next, we use the BGW bit conversion algorithm described in [3] to get the encrypted bit representation of dif, called dif_b. We then take the most significant bit of dif_b (called msb), calculate $(1 - msb) * sum$, and reveal the result. If $msb = 0$ then $sum \geq thresh$, so we reveal the aggregate. If $msb = 1$ then $sum < thresh$ so 0 is returned. This ensures that we only release the aggregate value if it is above $thresh$.

The BGW bit conversion described in [3] works as follows. The algorithm takes as input an encrypted share $[a]_p$ and returns an array of encrypted shares $[a]_b$. When decrypted, $[a]_b$ is the bit representation of decrypted $[a]_p$. This is done by generating an array of shares $[b]_b$ which represent random bits. Next, we reveal c = $([a]_p - [b]_b)$ and convert it to bit representation in the clear. $[a]_b = c + [b]_b$ can now be computed. In our implementation, we do not reveal any bits as we only need the encrypted value of the MSB.

This circuit relies on manipulation of Shamir's Secret Shares, which we know is resilient to $M < P/3$ malicious adversaries where P is the number of participating parties [4]. It also relies on the BGW Bit Conversion described in [3] which they prove is secure against an adversary which controls $M < \lfloor (P-1)/2 \rfloor$ honest-but-curious participants.

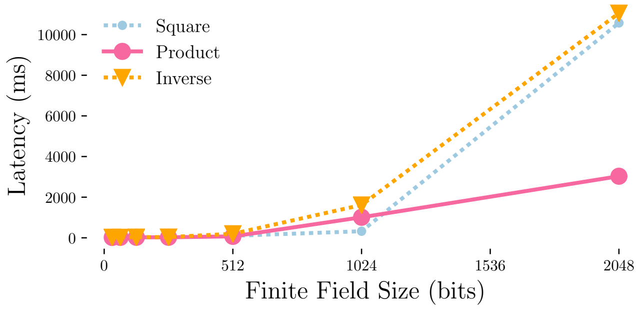

Fig. 2. Algorithm Performance (Client)

Client latency for various sizes of finite field for the 3 blinding algorithms.

IV. Performance

A. Verification

We will now discuss the performance of our implementations for the verification and aggregation schemes described above. Our verification scheme is lightweight and fast. Running on consumer grade hardware, we can run thousands of verifications simultaneously at roughly the same latency as running 100 queries simultaneously. In general, a verifier takes between 350 and 550 ms, and the client takes 20 to 50 ms for normal operating modes per query. There are five parameters which affect execution time which we will analyze in turn: field size, blinding algorithm, number of simultaneous queries, length of input vector, and number of verifiers.

When verifying, a random prime is selected for the finite field size used in all our arithmetic. The length of this field does not drastically affect the verifiers, but the client's performance is drastically impacted by the number of bits in the chosen prime. The client is decently efficient up to 512 bit primes, but it slows significantly beyond that. The graphs of finite field size vs latency for the client and verifiers can be seen in Figures 2 and 3 respectively.

The $Square$ and $Product$ algorithms for verification have similar latencies for small finite field sizes, but $Product$ is significantly faster than $Square$ for large finite field sizes (over 1024 bits). In addition, $Inverse$ is slower for the client at all finite field sizes. This relationship can be seen in Figures 2 and 4. A mojority of the client processing time is spent blinding and transfering data over the newtork. Because of this, we see large changes in performance due to blinding algorithm choice. The $Product$ algorithm is much more efficient than the alternatives because it only needs to execute $(P-1)$ multiplications per value to blind, whereas $Square$ performs $(P-1)$ exponentiation operations, and $Inverse$ performs $(P-1)$ multiplications and one exponentiation. The majority of server time is not spent

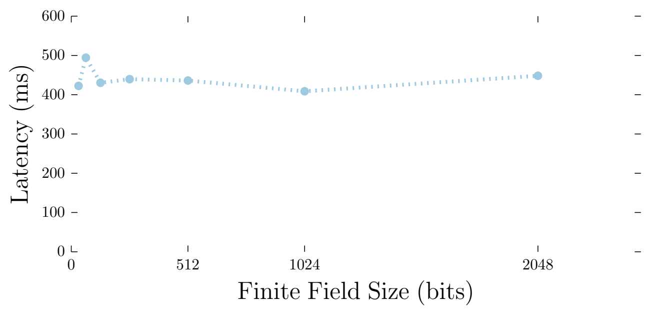

Fig. 3. Finite Field Size vs Latency (Server)

The latency observed by the servers for various finite field size.

Mode	Client (ms)	Server (ms)
Square	29.75.5	436.8968
Product	29.65	562.78
Inverse	45.24	473.94

Fig. 4. Algorithm vs Latency

The latency for client and server with different algorithms with a finite field size of 256 bits, 3 parties, and input vectors 100 values long. All algorithms are efficient at this size of finite field.

doing the final verification step, so we do not see a change in run-time for the servers when we change the algorithm used.

The number of simultaneous queries, unsurprisingly, affects latency. This linear slowdown is observed by both clients and verifiers. However, the slowdown has a gradual slope. Increasing from 1 query to 100 simultaneous queries only causes a slowdown of roughly 4 times. Figures 5 and 6 show the relationship between the number of simultaneous queries and latency for clients and verifiers respectively.

Our verification algorithm is highly parallelizable. Each

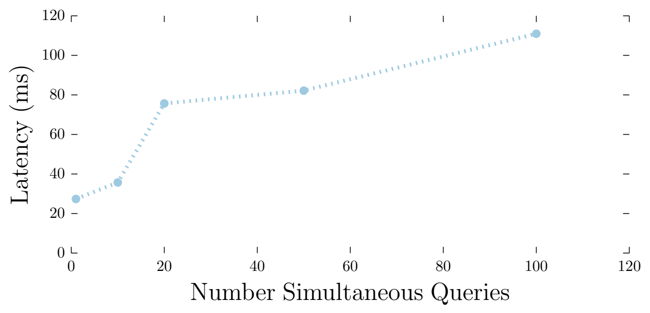

Fig. 5. Number Queries vs Latency (Client)

Latency observed by client while sending many simultaneous queries for verification.

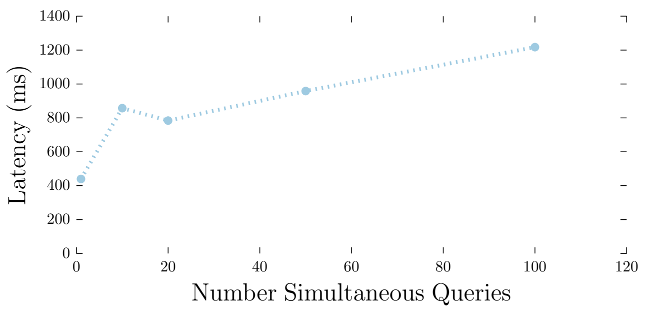

Fig. 6. Number Queries vs Latency (Server)

Server latency when processing multiple simultaneous verification requests.

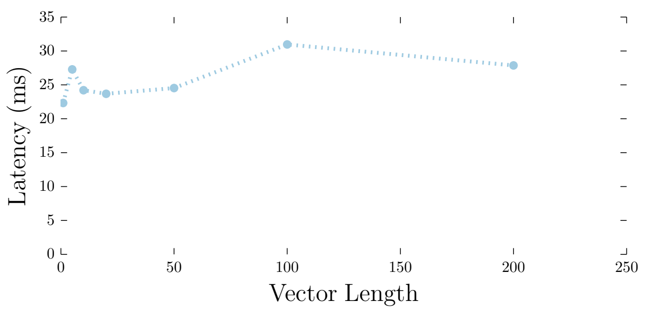

Fig. 7. Vector Length vs Latency (Client)

The latency observed by clients for various sizes on input vector.

element in the original input vector can be handled separately, and the only instance where threads must be joined is on the final check performed in the *validation* step. This leads to vector size not impacting latency for the clients or verifiers. All differences in timings can be attributed to external factors, such as random generation of larger numbers or an increase in network load due to other users on the network. The relationship between vector length and latency for the client and verifiers is shown in Figures 7 and 8.

During our verification process, every verification party communicates with all other servers, meaning our communication complexity is O(P^2) where P is the number of verification parties. This relationship is seen in our experimental results for the verifiers. However, the client's dependency is linear in P because an increased number of parties linearly relates to the number of blinded shares to create. These relationships can be seen in Figures 9 and 10.

B. BGW Aggregation

BGW bit conversion is incredibly expensive. As shown in [3], the BGW bit conversion protocol requires $O(L \log L)$

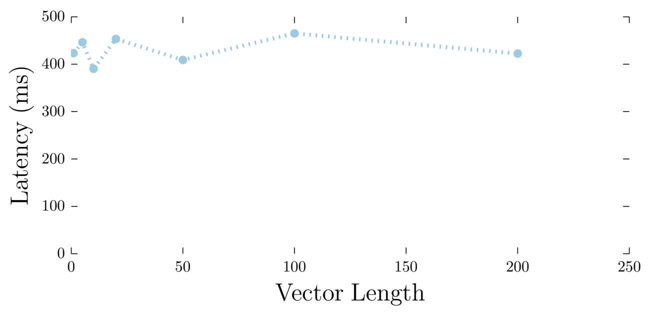

Fig. 8. Vector Length vs Latency (Server)

Server latency for various input vector lengths.

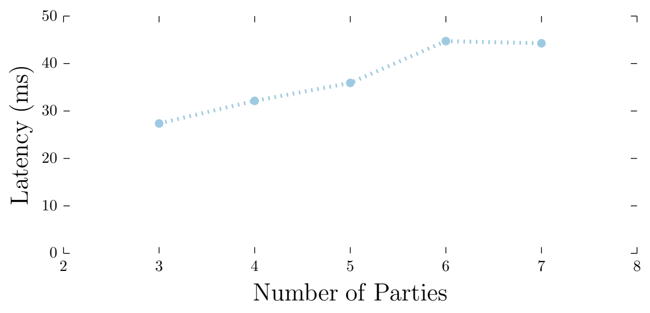

Fig. 9. Number of Parties vs Latency (Client)

Client latency vs number of parties used for verification.

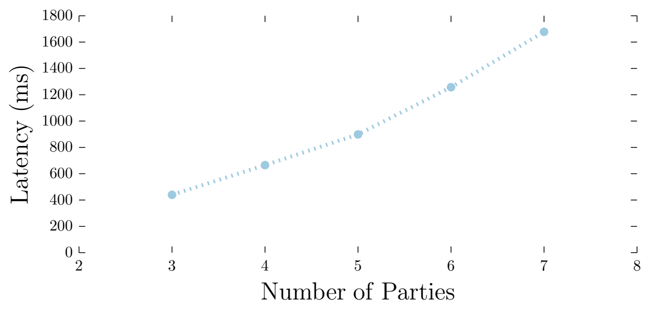

Fig. 10. Number of Parties vs Latency (Server)

Latency seen by servers for various number of parties used in verification process.

BGW multiplications where L is the nubmer of bits we wish to convert to. Each of these multiplications involves every server sending and receiving data to and from every other server, leading to large delays. Our findings mirrored these results. Once parallelized, our consumer-level hardware performed bit conversion in the following times:

L (bits)	Server (s)
8	49.4
32	508
64	1,850

This is incredibly inefficient, even though the algorithms are run in parallel. These times will improve with enterprise-level hardware, however they will still be quite slow. Alternative techniques (such as GMW circuits) are recommended.

V. Conclusion

This paper has described an implementation for the validation and aggregation of secret unit vector inputs in an MPC system. It has provided a fast, scalable system for validation with minimal overhead and computation requirements while retaining privacy as long as one honest server exists. It also provided a (slow) scheme for aggregation on secret inputs which only releases the sum if above a publicly known threshold.

References

[1] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1292–1303. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978429

[2] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359176

[3] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *Proceedings of the Third Conference on Theory of Cryptography*, ser. TCC'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 285–304. [Online]. Available: http://dx.doi.org/10.1007/11681878_15

[4] G. Asharov and Y. Lindell, "A full proof of the BGW protocol for perfectly secure multiparty computation," *J. Cryptology*, vol. 30, no. 1, pp. 58–151, 2017. [Online]. Available: http://dx.doi.org/10.1007/s00145-015-9214-4