Empowering SMPC: Bridging the Gap Between Scalability, Memory Efficiency and Privacy in Neural Network Inference

Ramya Burra
IUDX Program Unit, SID, IISc
ramya.burra@gmail.com

Anshoo Tandon
IUDX Program Unit, SID, IISc
anshoo.tandon@gmail.com

Srishti Mittal
IUDX Program Unit, SID, IISc
srishti.mittal@datakaveri.org

***Abstract*—This paper aims to develop an efficient open-source Secure Multi-Party Computation (SMPC) repository, that addresses the issue of practical and scalable implementation of SMPC protocol on machines with moderate computational resources, while aiming to reduce the execution time. We implement the ABY2.0 protocol for SMPC, providing developers with effective tools for building applications on the ABY 2.0 protocol. This article addresses the limitations of the C++ based MOTION2NX framework for secure neural network inference, including memory constraints and operation compatibility issues. Our enhancements include optimizing the memory usage, reducing execution time using a third-party Helper node, and enhancing efficiency while still preserving data privacy. These optimizations enable MNIST dataset inference in just 32 seconds with only 0.2 GB of RAM for a 5-layer neural network. In contrast, the previous baseline implementation required 8.03 GB of RAM and 200 seconds of execution time.**

***Index Terms*—SMPC, ABY2.0, MOTION2NX**

I. Introduction

In today's interconnected and data-driven world, the ability to perform computations while preserving privacy and security is paramount. Privacy is considered a fundamental human right as it balances the need for transparency and accountability with the protection of individual rights [1]. As technology advances and the digital age evolves, preserving privacy remains a pressing concern that requires ongoing attention and protection. Secure Multi Party Computation (SMPC) serves as a fundamental tool to address these concerns and facilitates secure data sharing and decision-making across various domains. When implementing SMPC for real-life data, it is essential to consider factors such as the nature of the data, the privacy requirements, the computational resources available, and the specific tasks to be performed. In this paper, our objective is to tackle the challenge of implementing the SMPC protocol in real world scenarios and at scale on machines with modest computational resources, all while striving to minimize the execution time. Our main goal is to optimize the code, making it more efficient and accessible to a broader audience, ultimately empowering SMPC. We offer neural network inference solutions with semi-honest security [11] executed on virtual machines with less than 1 GB RAM. Previous baseline implementation required 8.03 GB RAM on virtual machines for executing the same neural network model. In practical real-world setting, this reduction in RAM usage results in significant cost reduction (in dollar terms) [3].

To accomplish this, we modified the C++ based MOTION2NX framework [6] and included additional functionality to provide a *resource-optimized* implementation for secure inferencing tasks. We begin by examining MOTION2NX's limitations, such as memory issues and the lack of interoperability between tensor and non-tensor operations. The proposed enhancements include leveraging efficient tensor operations, overcoming the absence of an argmax function using novel approaches, optimizing memory usage, and reducing execution time with the introduction of a third-party Helper node. These improvements aim to enhance the framework's capabilities and efficiency while maintaining data privacy and integrity (refer to Section III for details).

In our optimized implementation, it is important to highlight that the memory usage of a standard N-layer neural network is determined by its largest layer, i.e., the layer with most parameters. This feature makes our implementation highly scalable as the memory footprint does not grow with the number of layers of the neural network. Moreover, we present an approach to further decrease the memory footprint by splitting the computations for the largest layer without compromising on privacy or accuracy.

We use the data provider framework of SMPC. We consider there are two compute servers (for executing the ABY2.0 SMPC protocol [9]) and two data providers (that possess input data and neural network model, respectively). Data providers provide shares to compute servers for computation in this framework (see Section II-D for details). The ABY2.0 protocol ensures privacy against a semi-honest adversary controlling one of the compute servers. Further, our implementational enhancements, providing RAM usage and execution time optimization, ensure that the compute servers receive no additional information for estimating the input values from the respective data providers. We assume that the readers are familiar with ABY2.0 protocol [9].

A. Related Work

Over the past several years, there has been an increased focus on practical application of SMPC to real-world problems. Here, we elucidate two real-life applications of SMPC.

a) Secure Auction: In Denmark, farmers sell sugar beets to Danisco. The Market Clearing Prices, which represent the price per unit of the commodity that balances total supply and demand in the auction, play a pivotal role in the allocation of contracts among farmers, ensuring a fair and efficient distribution of production rights. To preserve bid privacy in this process, a three-party SMPC system involving representatives from Danisco, and two other organizations was employed [5].

b) Secure Gender Wage Gap Study: Here, a specialized software facilitates data analysis of collaborative compensation for organizations, like the Boston Women's Workforce Council (BWWC) study in Greater Boston [7]. This application seamlessly integrates SMPC techniques to ensure collective computation of aggregate compensation data while preserving individual privacy. This approach empowers organizations to collaborate effectively while upholding data privacy.

We remark that the above two described SMPC applications are not memory or computationally intensive. On the other hand, in this paper, we present the modifications and functional additions to the MOTION2NX framework for practical implementation of secure neural network inference task that is *both* memory intensive and computationally intensive. These modifications and updates are a step towards secure disease prediction (see [2] for secure medical image analysis) where one party provides secret shares of medical images while the other party provides secret shares of a pre-trained neural network model.

B. Our Contributions

The following is the list of our contribution, specifically to MOTION2NX setup.

- Extending MOTION2NX to the setup where data providers and compute servers are separate entities.
- Creating an argmax function for multiple (> 2) inputs.
- Enabling writing of output shares to files and preventing reconstruction of the output at the compute servers. The compute servers send their respective final shares to the output owners.
- Inter-operability of tensor and non-tensor operations.
- A new optimized Helper node algorithm, working in conjunction with the ABY2.0 protocol, for efficient matrix multiplication.
- Our optimized 5-layer neural network inference requires about 0.2 GB of RAM, where as the original non-optimized version requires 8.03 GB of RAM. This implies over $40\times$ reduction in RAM usage.
- For obtaining realistic numbers, we opted to deploy the Compute servers and the Helper node on the cloud. Setting up and operating these systems require considerable amount of time due to installation of necessary dependencies and compilation of binaries in each of the machines. To simplify this process, we offer the docker images which contains all the essential dependencies and compiled binaries of the new enhancements we contributed. The SMPC Compute servers can now be easily brought up as docker containers from these docker images on any machine with docker installation.

II. Preliminaries

In this section, we provide preliminary details of our secure neural network inferencing implementation.

A. Framework for Implementation

We consider MOTION2NX, a C++ framework for generic mixed-protocol secure two-party computation in our paper. The following are the features of baseline MOTION2NX.

- Assumes data providers are a part of compute servers
- No intermediate values are reconstructed
- Assumes either of the compute servers as output owners
- Output is reconstructed in clear and shared with the output owner

Tensor and non-Tensor variants: MOTION2NX offers non-optimized secure functions that compile the descriptions of low-level circuits, which are referred to as non-tensor operations. MOTION2NX also provides optimized building blocks that directly implement common high-level operations, which are referred to as tensor operations. The tensor operations are more computationally efficient than the primitive operations. They use a specialized executor to evaluate the tensor operations sequentially while parallelizing the operations themselves with multi-threading and SIMD operations.

We discuss the details of our proposed enhancements and optimizations in Section III.

B. N-layer Neural Network

Our optimizations in MOTION2NX enable us to implement deep neural networks with relatively large number of layers. For illustrative purposes, we only present the details for 2-layer and 5-layer neural networks on MNIST dataset. A similar procedure can be adapted for any other dataset on their pretrained models with multiple neural network layers.

The input to the neural network is a real-valued vector of size 784×1. The output is a boolean vector of size 10, where only one element is set to 1. Each element in the output vector corresponds to an index from $\{0, 1, 2, \ldots, 9\}$. If the j^{th} element in the vector is 1, it indicates that the predicted label is j. The dimensions of weights and biases used in our implementation are listed below.

1) 2 layer neural network dimensions:

- Layer 1 : Weights (256×784), bias (256×1)
- Layer 2 : Weights (10×256), bias (10×1)

2) 5 layer neural network dimensions:

- Layer 1 : Weights (512×784), bias (512×1)
- Layer 2 : Weights (256×512), bias (256×1)
- Layer 3 : Weights (128×256), bias (128×1)
- Layer 4 : Weights (64×128), bias (64×1)
- Layer 5 : Weights (10×64), bias (10×1)

Algorithm 1 describes a simple two-layer secure neural network inference implementation with ReLU activation. This algorithm takes ABY2.0 shares of input data (image), neural

network weights, and biases as inputs and produces ABY2.0 shares of the predicted label as output.

We recall that ABY2.0 shares consist of a pair comprising a public share and a private share [9]. For instance, the ABY2.0 shares of an input variable y associated with server i are represented as a pair consisting of Δ_y and $[\delta_y]_i$. Note that Δ and δ represent public share and private share respectively.

The secure functions listed in Algorithm 1 were provided by MOTION2NX framework. We used ABY2.0 arithmetic protocol for multiplication and addition. We used Yao protocol to perform the ReLU function, as Yao performs better for comparison operations. Before performing Yao operations we convert ABY2.0 arithmetic shares to Yao shares with the help of MOTION2NX functions. Post the ReLU function we convert the Yao shares to ABY2.0 arithmetic shares with the help of MOTION2NX inbuilt functions. We choose boolean ABY2.0 protocol to implement the argmax function.

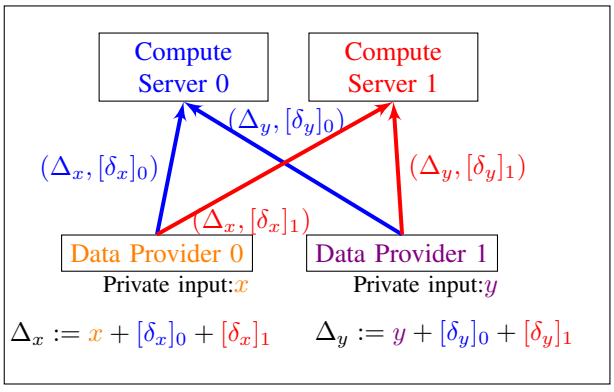
Algorithm 1, unfortunately, couldn't be executed completely in MOTION2NX using its built-in functions. In a single MOTION2NX instance, only one of the tensor variant or non-tensor variant could be utilized. Regrettably, the tensor variant lacked an argmax function, which prevented us from implementing Algorithm 1 using its standard functions. Nevertheless, we were able to address this issue by applying the optimizations outlined in Section III-A, which provides further insights into this matter.

Algorithm 1 Neural Net inferencing task with ReLU activation function at compute server-i, $i \in \{0, 1\}$

Require: Input image shares x^i, weight shares w_1^i, w_2^i, bias shares b_1^i, b_2^i. All the above shares are vectors in the form of ABY2.0 shares

Ensure: Shares of predicted class label $\hat{y}^i$

1: Compute first layer:
2: z_1^i =SecureAdd(SecureMul$(w_1^i, x^i), b_1^i)$
3: h_1^i =SecureReLU$(0, z_1^i)$
4: Compute Second layer:
5: z_2^i =SecureAdd(SecureMul$(w_2^i, h_1^i), b_2^i)$
6: Compute predicted class label:
7: $\hat{y}^i$ =SecureArgmax(z_2)
8: **Return** $\hat{y}^i$



We remark that the steps outlined in Algorithm 1 can be readily extended to provide secure inferencing results for a general fully-connected neural network with $N\,(> 2)$ layers.

C. Reconstructing the Predicted label in clear

Algorithm 2 outlines the process of deriving the predicted label from boolean output shares. With MNIST, the label is a number from 0 to 9. The output here is a boolean vector of 10 elements indexed from 0 to 9, with only one element set to 1; a 1 in the j^{th} element signifies the predicted label as j.

Algorithm 2 Reconstructing the Predicted label in clear

Require: Boolean ABY2.0 shares from both the servers $\hat{y}^0, \hat{y}^1$

Ensure: Predicted class label Y

1: **for** j $= 0$ to 9 **do**
2: $\hat{y}[j] = \Delta_{\hat{y}[j]} \oplus [\delta_{\hat{y}[j]}]_0 \oplus [\delta_{\hat{y}[j]}]_1$
3: **end for**
4: Initialize index $k \leftarrow 0$
5: **while** $k < 10$ and $\hat{y}[k] = 0$ **do**
6: Increment index $k \leftarrow k + 1$
7: **end while**
8: **Return** k ▷ The predicted label

D. Data Provider Model

In this model the data providers (Image provider and Model provider) create shares of their private data and communicate them with the compute servers for further computation. Compute servers perform the inferencing task and send the output shares to the Image provider. Compute servers are unaware of the clear output result. For secure inferencing task, we consider that the neural network model is pretrained and is proprietary to a model provider. Similarly, the image for the inferencing task is private to image data provider.

III. Enhancements, Optimizations, and Feature Additions to MOTION2NX

Before we proceed with a discussion of the enhancements proposed in our paper, let's first examine the limitations of MOTION2NX.

Limitations:

- No provision to implement data provider model
- Output is always reconstructed at the compute server(s)
- Memory issues : A simple 2-layer neural network inference requires about 3.2 GB of RAM
- No interoperatability between tensor and non-tensor operations.

A. Implementing Argmax and writing output shares

MOTION2NX can implement the functions (matrix multiplication, add etc.) using tensor and non-tensor operations. As discussed in Section II-A, tensor operations are optimized versions that enable faster calculations. Therefore, to perform an inference operation on a neural net, MOTION2NX authors

recommend using tensor operations. Also, practically we verified that inferencing task on MNIST dataset takes around 1 minute using tensor operations. A simple multiplication operation of two numbers in a non-tensor version takes about 1.5sec. An inferencing task on MNIST data set on a two layer neural network has operations of the order $\approx 10^6$. Clearly, the tensor operations are much more efficient compared to the non-tensor operations.

To implement the neural net we need to have an argmax function (providing the index of the maximum value in a vector) for appropriate classification of the MNIST image. However, MOTION2NX has no provision to implement the argmax function using the tensor operations. Although there are functions to compute max and argmax using non-tensor operations for two inputs, these functions cannot be used in the inferencing task as *there is no provision to interlink tensor and non-tensor operations in MOTION2NX*. We modified MOTION2NX to store the output shares of the neural net just before executing the argmax function, while ensuring that the corresponding clear values of the output shares are never reconstructed in the code. This feature is an improvement over the existing framework as it enables us to use tensor and non-tensor operations together in a sequential manner. The output shares of the last layer of the neural net are fed to the non-tensor implemetation of the argmax function to obtain the required shares of the predicted label. This argmax function operates on a vector with multiple (> 2) inputs, and internally uses the in-built max and argmax functions for two inputs in a recursive fashion for generating the output shares of the predicted label.

In our implementation, we pinpointed the stage at which output shares become accessible, just before the reconstruction step. Note that in the original MOTION2NX framework, the compute servers initiate the transmission of their private share for the purpose of reconstruction. We modified the MOTION2NX framework to halt the broadcast of the private shares, and instead save the local public and private shares to respective files at the two compute servers. It is important to note that the private shares of each server remain local (not shared with the other server), ensuring no loss of privacy.

B. Optimizing Memory Requirement

In the context of secure inferencing tasks using the MNIST dataset, we observed a substantial RAM requirement of approximately 3.2 GB (per server instance) for a two-layer neural network. In practical terms, this RAM demand poses a significant obstacle to performing inferencing tasks on a resource constrained machine. This issue is further aggravated when we work with a neural network with relatively large number of layers. To address this challenge, our primary objective was to reduce the memory requirement, thereby facilitating the use of more complex neural networks.

Notably, for a multi-layer neural network, MOTION2NX constructs the entire end-to-end circuit in a single step and retains the RAM memory until the entire execution has been completed. Our approach, as outlined in Section III-A, involves writing the shares of intermediate results in respective files while ensuring that no reconstruction is performed. This significantly reduces the RAM requirement for multi-layer neural network inferencing tasks.

Splits	RAM requirement	Execution time
No intra-layer split	3.2 GB	34 seconds
Layer 1: 2 splits	1.6419 GB	38 seconds
Layer 1: 4 splits	0.888 GB	43 seconds
Layer 1: 8 splits	0.4527 GB	47 seconds
Layer 1: 16 splits	0.253 GB	50 seconds
Layer 1: 64 splits	0.09888 GB	73 seconds

TABLE I: Compute servers run on the same machine for 2 layer NN. Note that there is no intra-layer split for layer2 in the above.

In the specific context of a 2-layer neural network inferencing model (with dimensions detailed in Section II-B1), the highest memory requirement arises from the matrix multiplication in layer-1. To mitigate this, we perform intra-layer optimization where the matrix multiplication task is implemented in smaller segments (or "splits"), resulting in a proportional decrease in the average RAM requirement. It is evident from Table I that the average RAM requirement scales down almost linearly with the number of splits employed.

C. Optimizing Execution Time using Helper Node Algorithm

After successfully reducing the average RAM requirement, our next objective was to optimize the execution time. Here, we implemented compute servers on different machines but on same LAN for realistic execution times. The numbers are detailed in Tables II and III. For secure matrix multiplication, the execution time is significantly impacted by the use of oblivious transfers (OTs) which occur behind the scenes. To address this issue, we introduce a semi-honest third-party Helper node that eliminates the need for OTs during matrix multiplication.

As a result of implementing the Helper-node algorithm, the execution time for the inference operation is reduced from 77 sec (for the baseline implementation with no intra-layer optimization on a 2-layer nural network) to 11 sec. Additionally, the RAM requirement when utilizing the Helper node algorithm is only 0.134 GB (see Table II).

Note that we refer to this algorithm as "ABY2.0 with a Helper node", drawing inspiration from the implementation of Beaver triples produced by a third-party helper in [10]. The primary objective of our Helper node is to eliminate the need for Oblivious Transfers (OTs) in both the online and setup phases of ABY2.0. In this context, we specifically delve into how the Helper node's algorithm modifies the ABY2.0 Multiplication Protocol, denoted as Protocol MULT$(<a>, <b>)$ in [9, Section 3.1.3].

In summary, Protocol MULT$(<a>, <b>)$ takes ABY2.0 shares of a and b from two parties as inputs. It performs operations to generate output shares representing the product $a \times b$. Importantly, in this protocol, neither party possesses knowledge of the clear output unless they engage in communication to share their respective output secret shares.

It's worth noting that Protocol MULT $(<a>, <b>)$ relies on an OT-based setupMULT during the pre-processing phase.

Our specific goal is to eliminate this setupMULT and replace it with a more efficient approach using helperNODE. Here, the objective is to calculate

$$\delta_{ab} = [([\delta_a]_0 + [\delta_a]_1)([\delta_b]_0 + [\delta_b]_1)], \quad (1)$$

and share the additive components of this expression with both server nodes. In this process, party 0 conveys $[\delta_a]_0$ and $[\delta_b]_0$ to the Helper node through a reliable channel, while party 1 conveys $[\delta_a]_1$ and $[\delta_b]_1$ to the Helper node via another reliable channel. Subsequently, the Helper node computes δ_{ab} and distributes additive shares to both parties. It is crucial to note that the Helper node is *incapable of reconstructing the actual values of a and b as it possesses no knowledge of Δ_a and Δ_b*. Additionally, we assume that both the compute server parties and the Helper node operate in a semi honest fashion. It is important to emphasize that if the information from the Helper node is shared with either of the two servers, it jeopardizes the data privacy of the input data. To uphold these properties, one may consider running the compute servers and the Helper node on separate secure enclave machines (see [4] for details on the secure enclave machines).

In Algorithm 3, we provide an overview of the multiplication algorithm using the Helper node. This algorithm needs the procedure HelperNODE in the setup phase. To enhance readability, Algorithm 3 addresses the efficient implementation of two scalar values, a and b. However, we carry forward this very idea into our optimized implementation for matrix multiplication.

Algorithm 3 Protocol HELPERMULT($<a>, <b>$)

1: **Setup Phase:**
2: P_i for $i \in 0, 1$ samples random $[\delta_y]_i \in_R \mathbb{Z}_2^{64}$.
3: Parties execute HELPERNODE($[\delta_a], [\delta_b]$) to obtain $[\delta_{ab}]_i$.
4: **Online Phase:**
5: P_i for $i \in 0, 1$ locally computes $[\Delta_y]i = i\Delta_a\Delta_b - \Delta_a[\delta_b]_i - \Delta_b[\delta_a]_i + [\delta_{ab}]_i + [\delta_y]_i$ and sends to P_{1-i}.
6: P_i for $i \in 0, 1$ computes $\Delta_y = [\Delta_y]_0 + [\Delta_y]_1$.

1: **procedure** HELPERNODE($[\delta_a], [\delta_b]$)
2: P_i for $i \in 0, 1$ send $([\delta_a]_i, [\delta_b]_i)$ to the Helper node.
3: Helper node computes equation (1).
4: Helper node creates arithmetic shares $([\delta_{ab}]_0, [\delta_{ab}]_1)$, such that $\delta_{ab} := [\delta_{ab}]_0 + [\delta_{ab}]_1$
5: Helper node sends $[\delta_{ab}]_i$ to P_i for $i \in 0, 1$
6: **end procedure**

IV. Numerical Evaluation

Tables II and III outline the execution time for 2-layer and 5-layer neural networks, respectively, when the compute resources are run on the same LAN. In practice, the compute servers are typically hosted on the cloud (on different LANs). To determine the execution time, we deployed compute server 0 on Microsoft Azure cloud and compute server 1 (and the Helper node) on AWS cloud. The image provider and weights/model provider run on separate local machines. See Table IV for details of the cloud configurations. The image provider's terminal accepts a handwritten digit and converts it into a 28×28 .csv file, which is then flattened to a 784×1 format. The model provider possesses the weights and biases of a pretrained model. Both the image provider and model provider generate ABY2.0 shares of their respective private data and send these shares to the compute servers hosted on the cloud. Following the sharing of these secret shares, the image provider awaits the output shares of the predicted image label. After the execution of the secure inferencing task at the two compute servers, these servers send their respective output shares back to the image provider for the reconstruction of the predicted image label in plain text. Tables V and VI list details

Splits	RAM requirement	Execution time
No intra-layer split	3.2 GB	77 seconds
Layer 1: 8 splits	0.4527 GB	100.3 seconds
Layer 1: 16 splits	0.253 GB	108.45 seconds
Layer 1: 64 splits	0.09888 GB	121.53 seconds
Helper node	0.134GB	11 seconds

TABLE II: Compute servers and Helper node are run on different machine on same LAN for 2-layer neural net. Note that there is no intra-layer split for layer2 in the above.

Which layers are split, number of splits	RAM requirement	Time of execution
No intra-layer split	8.03 GB	200 seconds
Layer 1: 8 splits Layer 2: 4 splits Layer 3: 2 splits Layer 4,5: No split	0.894 GB	212 seconds
Layer 1: 16 splits Layer 2: 8 splits Layer 3: 4 splits Layer 4: 2 splits Layer 5: No split	0.461 GB	210 seconds
Helper node	0.2 GB	32 seconds

TABLE III: Compute servers and Helper node are run on different machine on same LAN for 5-layer neural net.

Server 0	Azure: b1s 1vcpu, 1 GB RAM, 30 GB SSD
Server 1	AWS: t2.micro 1vcpu, 1 GB RAM, 30 GB SSD
Helper node	AWS: t2.nano 1vcpu, 0.5 GB RAM, 30 GB SSD
Image provider	Personal laptop
Weights provider	Personal laptop

TABLE IV: Cloud configuration

Splits	RAM requirement	Time of execution
Layer 1: 16 splits	0.253 GB	200 seconds
Layer 1: 64 splits	0.0988 GB	280 seconds
Helper node	0.134 GB	12 seconds

TABLE V: Compute servers and Helper node are run on cloud for 2-layer neural net. Note that there is no intra-layer split for layer2 in the above.

Which layers are split, number of splits	RAM requirement	Time of execution
Layer 1: 16 splits Layer 2: 8 splits Layer 3: 4 splits Layer 4: 2 splits Layer 5: No split	0.461 GB	525 seconds
Helper node	0.2 GB	34 seconds

TABLE VI: Compute servers and Helper node are run on cloud for 5-layer neural net.

of the 2-layer and 5-layer neural networks, respectively, when compute servers are run on cloud. As expected, the execution time with compute servers running on cloud machines (on different LANs) is higher compared to the case where compute servers are part of the same LAN (compare Tables II, III with Tables V, VI).

We remark that the accuracy of our secure neural network inferencing implementation (using 64 bit fixed-point arithmetic including 13 bits for representing the fractional part) on the MNIST dataset was roughly similar to the corresponding accuracy obtained using a python floating-point implementation.

V. Model known to compute servers

We recognize that neural network model parameters could be considered common knowledge, yet the image provider still prioritizes data privacy. To address such scenarios, we introduced a new functionality where common knowledge variables are treated as known unencrypted values. To achieve this, we have introduced two operations, namely "ConstantMul" and "ConstantAdd", within the MOTION2NX framework. These operations were not previously available in the framework. We discuss these algorithms in Algorithm 4 and 5, respectively.

In this context, we observed that the inference time and RAM requirements are significantly reduced when compared with the baseline (where the model is private to one of the data providers). In this context, for a two-layer neural network, the inference time is approximately 13 seconds with a RAM requirement of 0.134 GB. In the case of a five-layer neural network, the inference time is around 34 seconds, and the RAM requirement is 0.2 GB. These numbers are similar to the corresponding numbers obtained using the helper node. In this model, this reduction in execution time is expected since the overhead of performing OTs during secure multiplication between two private values is eliminated. Let a be the private data, b be the known data. In the following, we use ABY2.0 shares of a and uint64 equvivalent of b with f fractional bits. Let $y = a \times b$. It's important to highlight that during the online phase, after the multiplication of the shares by the constant value, we transformed the ABY2.0 shares into the GMW arithmetic shares. Note that the GMW shares are represented by Y_i in Algorithm 4. The transformation to GMW shares was needed to prevent the occurrence of wrap-around errors during the truncation operation performed on Y_i (see [8, paragrah 4, Section 5.1.1] for a discussion of wrap-around errors during the truncation of respective shares). The truncation operation ensures that after the multiplication operation, the fractional bits continue to be represented by f least significant bits.

Algorithm 4 Protocol ConstantMULT($<a>, b$)

1: **Setup Phase:** P_i for $i \in \{0,1\}$ samples $[\delta_y]_i \in_{\mathbb{R}} Z_{2^{64}}$
2: P_i updates $[\delta_a]_i$ with $[\delta_a]_i \times b$
3: **Online Phase:** P_i updates Δ_a with $\Delta_a \times b$
4: P_0 locally generates $Y_0 = 0 \times \Delta_a - [\delta_a]_0$
5: P_1 locally generates $Y_1 = 1 \times \Delta_a - [\delta_a]_1$
6: P_i for $i \in \{0,1\}$ performs truncation operation: $Y_i = \frac{Y_i}{2^f}$
7: P_i locally computes $[\Delta_y]_i = Y_i + [\delta_y]_i$ and sends to P_{1-i}
8: Both P_0 and P_1 calculate $\Delta_y = [\Delta_y]_0 + [\Delta_y]_1$

Algorithm 5 Protocol ConstantAdd($<a>, b$)

1: **Setup Phase:** P_i for $i \in \{0,1\}$ computes $[\delta_y]_i = [\delta_a]_i$
2: **Online Phase:** Both P_0 and P_1 compute $[\Delta_y] = [\Delta_a] + b$

VI. Conclusion and Future Work

We modified and enhanced the MOTION2NX framework to bridge the gap between scalability, memory efficiency, and privacy. In particular, we optimized the memory usage, reduced the execution time using a third-party Helper node, and enhanced the efficiency while still preserving data privacy. These optimizations enable MNIST dataset inference in just 32 seconds with only 0.2 GB of RAM for a 5-layer neural network. In contrast, the previous baseline implementation required 8.03 GB of RAM and 200 seconds of execution time. These optimizations yielded remarkable results in terms of reduced inference time and lower average memory consumption.

Acknowledgements

We want to thank all the interns (Udbhav, Rishab, Rashmi, and Shreyas) who extended their support to this project. We also thank Abhilash for helping us setup the cloud instances.

References

[1] Case Number: WP (C) 494/2012. Fundamental right to privacy, 2021. https://www.scobserver.in/cases/puttaswamy-v-union-of-india-fundamental-right-to-privacy-case-background/, Accessed: 2023-09-25.

[2] Javier Alvarez-Valle, Pratik Bhatu, Nishanth Chandran, Divya Gupta, Aditya Nori, Aseem Rastogi, Mayank Rathee, Rahul Sharma, and Shubham Ugare. Secure medical image analysis with cryptflow, 2020. arXiv,2012.05064.

[3] Amazon. VPS, web hosting pricing, 2023. https://aws.amazon.com/lightsail/pricing/.

[4] Jethro Gideon Beekman. *Improving cloud security using secure enclaves*. PhD thesis, UC Berkeley, 2016.

[5] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Illeborg Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography*, 2009.

[6] Lennart Braun, Rosario Cammarota, and Thomas Schneider. A generic hybrid 2PC framework with application to private inference of unmodified neural networks (extended abstract). In *NeurIPS 2021 Workshop Privacy in Machine Learning*, 2021. https://openreview.net/forum?id=CXFh9utHuw2.

[7] Andrei Lapets, Eric Dunton, Kyle Holzinger, Frederick Jansen, and Azer Bestavros. Web-based multi-party computation with application to anonymous aggregate compensation analytics, 2015. https://open.bu.edu/handle/2144/21773.

[8] Payman Mohassel and Peter Rindal. ABY3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 35–52, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved Mixed-Protocol secure Two-Party computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2165–2182. USENIX Association, August 2021.

[10] TF encrypted. tf-encrypted, 2023. https://github.com/tf-encrypted/tf-encrypted/blob/master/tf_encrypted/protocol/pond/pond.py.

[11] Thijs Veugen, Frank Blom, Sebastiaan J. A. de Hoogh, and Zekeriya Erkin. Secure comparison protocols in the semi-honest model. *IEEE Journal of Selected Topics in Signal Processing*, 9(7):1217–1228, 2015.