On the Application of Binary Neural Networks in Oblivious Inference

Mohammad Samragh*  
UC San Diego  
msamragh@ucsd.edu

Siam Hussain*  
UC San Diego  
s2hussai@ucsd.edu

Xinqiao Zhang  
UC San Diego, San Diego State University  
x5zhang@ucsd.edu

Ke Huang  
San Diego State University  
khuang@sdsu.edu

Farinaz Koushanfar  
UC San Diego  
farinaz@ucsd.edu

Abstract

This paper explores the application of Binary Neural Networks (BNN) in oblivious inference, a service provided by a server to mistrusting clients. Using this service, a client can obtain the inference result on her data by a trained model held by the server without disclosing the data or leaning the model parameters. We make two contributions to this field. First, we devise light-weight cryptographic protocols designed specifically to exploit the unique characteristics of BNNs. Second, we present dynamic exploration of the runtime-accuracy tradeoff of BNNs in a single-shot training process. While previous works trained multiple BNNs with different computational complexities (which is cumbersome due to the slow convergence of BNNs), we train a single BNN that can perform inference under different computational budgets. Compared to CryptoFlow2, the state-of-the-art in oblivious inference of non-binary DNNs, our approach reaches 2× faster inference at the same accuracy. Compared to XONN, the state-of-the-art in oblivious inference of binary networks, we achieve 2× to 11× faster inference while obtaining higher accuracy.

1. Introduction

There is an increasing surge in cloud-based inference services that employ deep learning models. In this setting, the server trains and holds the DNN model and clients query the model to perform inference on their data. One major shortcoming of such service is the leakage of clients’ private data to the server, which can hinder commercialization in certain applications. For instance, in medical diagnosis [1], clients would need to expose their “plaintext” health information to the server, which violates patient privacy regulations such as HIPAA [2].

One attractive option for ensuring clients’ content privacy is the use of modern cryptographic protocols as they provide provable security guarantees [3–13]. Let $f(\theta, x)$ be the inference result on client’s input $x$ using server’s parameters $\theta$. By executing cryptographically-secure operations, client and server can jointly compute $f(\theta, x)$ without revealing $x$ to the server or $\theta$ to the client. We refer to this process as *oblivious inference* in the remainder of the paper. Unlike plaintext inference, oblivious inference protects the privacy of both parties. The challenge, however, is the excessive computation and/or communication overhead associated with privacy-preserving computation. For example, the contemporary state-of-the-art for performing oblivious inference on a single CIFAR-10 image requires exchange of $\sim 3.4$ GB of data and takes $\sim 10$ seconds [14].

Early research on oblivious inference mostly focused on developing protocols for inference of a given DNN model, without making major modifications to the model itself [3–13]. Recently, a body of work has explored modifying the DNN architecture such that the resulting model is more amenable to secure computation [14–17]. Other potential directions for enhancing oblivious inference could include pruning [18], tensor decomposition [19], quantization [20], and Binary Neural Networks (BNNs) [21]. In this work, we study BNN as a candidate for fast and scalable oblivious inference. We show that a BNN has several unique characteristics that allow translating its computations to simple and efficient cryptographic protocols.

The benefits of employing BNNs for oblivious inference were first noted by XONN [14]. Despite achieving significant runtime improvement compared to non-binary DNN inference, there are opportunities provided by BNNs that have not been leveraged by XONN. Part of the inefficiency of XONN is due to the usage of a single secure computation protocol as a blackbox for all neural network layers after the input layer. In this work, we introduce a new hybrid approach where the underlying secure computation protocol is customized to each layer, such that the total execution cost for oblivious inference on all layers is minimized. We design a composite custom secure execution protocol, specifically optimized for BNN operations, using standard
security primitives. Our protocol significantly improves the efficiency of XONN as we show in our experiments.

One standing challenge in oblivious inference is finding architectures that are both accurate and amenable to secure computation. Since BNNs suffer from long training time and poor convergence, searching for such architectures could be quite inefficient. We address the inefficiency challenge by training a single BNN that can operate under different computational budgets. Our adaptive BNN offers a tradeoff between accuracy and inference time, without requiring to train separate models. Figure 1 presents the tradeoff achieved by our flexible BNN on the 7-layer VGG network trained on CIFAR-10. With the combined power of our custom oblivious inference protocols and adaptive BNN training schemes, our method outperforms prior art both in terms of accuracy and runtime. Our solution is $\sim 2 \times$ faster than XONN [12], the state-of-the-art non-binary DNN inference framework, and $2 \times$ to $11 \times$ faster than XONN, the previous oblivious BNN inference framework.

2. Scenario and Threat Model

Figure 2 presents the scenario in oblivious inference. The neural network architecture $f$ is known by both server and client. The server holds the set of trained parameters, i.e., $\theta = \{\theta^1, \ldots, \theta^L\}$, and the client holds the input query to the neural network, i.e., $x$. The two parties engage in a secure function evaluation protocol, where the client learns the inference result $y = f(\theta, x)$. Similar to prior work, we consider the honest-but-curious scenario [10–17]. In this threat model, the two parties follow the protocol that they agree upon to compute the output, yet they may try to learn about the other party’s data as much as they can. As such, the protocol should guarantee the following requirements:

- $x$ or $f(\theta, x)$ are not revealed to the server.
- $\theta$ is not revealed to the client.
- Client and server do not learn intermediate activations.

3. Background

This section provides a high-level outline of the necessary terminologies. Following the convention in secure computation literature, we refer to server and client as Alice and Bob, respectively.

Secure Function Evaluation Protocol. During oblivious inference, Alice and Bob engage in a Secure Function Evaluation (SFE) protocol, which is essentially a set of rules specifying the messages communicated between them. By following these rules, they jointly compute the output of a function that takes the inputs from both of them without disclosing any information about Alice’s data to Bob and vice versa. Depending on protocol agreements, the result of the computation can be exposed to both parties, only one of them, or neither of them.

Additive Secret Sharing (AS) is a method for distributing a secret $x$ between Alice and Bob such that Alice holds $[x]_A = x + r$ and Bob holds $[x]_B = -r$, where $r$ is a random value. Individually, both $[x]_A$ and $[x]_B$ are random values, hence, Alice and Bob cannot independently decipher the original message $x$. Only by combining $[x]_A$ and $[x]_B$ can one recover the actual secret as $x = [x]_A + [x]_B$. There exists standard SFE protocols to perform addition and multiplication on secret-shared data such that the result is also shared between the two parties. We employ these protocols in oblivious inference to ensure that neither the input nor the output of a layer is revealed to the involved parties. We refer curious readers to [22] for more details.

Oblivious Transfer (OT) is a protocol between two parties – a sender (Bob) who has two messages $(\mu_0, \mu_1)$, and a receiver (Alice) who has a selection bit $i \in \{0, 1\}$ [23]. Through OT, Alice obtains the intended message $\mu_i$, without revealing the selection bit $i$ to Bob. Alice does not learn the other message $\mu_{1-i}$. OT requires public key cryptography, which is costly in general. In the following, we introduce more efficient methods for OT computation.

OT extension enables extending a constant number of ‘base OTs’ to a large number of OTs through cheaper symmetric key cryptography [24]. The first step in OT-extension is called Random OT (ROT) [25]. In ROT, Alice provides the selection bit $i$ and Bob does not provide any input. After ROT execution, Bob receives two random 128-
We show in Section 4.1 that conditional summations can be computed via multiplying vector dot products of the form $\sum_{i=1}^{N} w_i x_i$, where the $\oplus$ operator denotes bit-wise XOR and $H(k)$ is a cryptographically-secure random number generator [26] with $k$ as the seed. Bob transmits $\{v_0, v_1\}$ to Alice, who computes $\mu_i = v_i \oplus H(k_i)$.

In Section 4.1, we design a protocol for oblivious matrix multiplication, which enables oblivious evaluation of convolution and fully-connected layers. We build our protocol by only using OT and AS which we outlined above. However, AS and OT are not efficient for evaluating non-linear activations and Max-pooling.

Garbled Circuit (GC) is an SFE protocol that can be used for evaluation of an arbitrary function (linear or non-linear). The downside of GC is its heavy communication overhead. Therefore, we limit its usage to nonlinear operations. We refer curious readers to [27–29] for more details about GC.

### 4. Cryptographically Secure BNN Inference

BNNs were originally introduced to minimize memory footprint and computation overhead of plaintext inference. In this section, we provide insights on why BNNs are also useful for very efficient and fast oblivious inference.

The first favorable property of BNNs is enforcing the weights to $+1$ or $-1$. With this restriction, multiplying a feature $x$ by a weight $w$ is equivalent to computing either $+x$ or $-x$. This simple property becomes useful when computing vector dot products of the form $\sum_{i=1}^{N} w_i x_i$, which can be computed via $N$ conditional additions/subtractions. We show in Section 4.1 that conditional summations can be computed using OT and AS, both of which are known to be very efficient and lightweight cryptographic tools.

In oblivious inference, nonlinear operations are evaluated through heavy cryptographic primitives such as GC, resulting in large runtime and communication overheads. The large communication cost of GC is directly related to the bit-widths of GC inputs. The second advantage of BNNs is their 1-bit hidden layer feature representation, which significantly reduces the GC evaluation cost when compared to non-binary features. In Section 4.2, we expand on low-bit nonlinear operations and their efficient GC evaluation.

We present the overall flow for oblivious BNN inference in Figure 3. The inputs and outputs of all layers are in AS format, e.g., server and client have $[Y_i]_A$ and $[Y_i]_B$ rather than $Y_i$. To obliviously evaluate linear layers (CONV or FC), we propose a novel custom protocol for binary matrix multiplication that directly works on AS data. We merge batch normalization (BN), binary activation (BA), and max-pooling (MP) into a single nonlinear function $f(\cdot)$. To securely evaluate $f([Y_i]_A, [Y_i]_B)$, three consecutive steps should be taken:

1. Securely translating the input from AS to GC. This step prepares the data to be processed by GC.
2. Computing the nonlinear layer through GC protocol.
3. Securely translating the result of the GC protocol to AS. This step prepares the data to be processed in the following linear layer.

Using this hybrid approach, we achieve a significantly faster oblivious inference compared to the state-of-the-art [14].

#### 4.1. Linear Layers

Fully-connected and convolutional layers require computing $Y = WX$, with weight matrix $W$ and input $X$. In secure matrix multiplication, the input is secret shared between the server and the client, i.e., $X = [X]_A + [X]_B$. Bob (the client) has $[X]_B$ whereas Alice (the server) has the weight $W$ and $[X]_A$.

The matrix multiplication is computed as follows:

$$W([X]_A + [X]_B) = W[X]_A + W[X]_B$$  \hspace{1cm} (1)

Alice can compute $W[X]_A$ locally and only $W[X]_B$ needs secure evaluation. After evaluating $Y = WX$,

- Alice gets $[Y]_A$ but does not learn $[X]_B$ or $[Y]_B$.

\[\text{At the first layer, only client has the input share, hence } [X]_A = 0\]
Algorithm 1: One-time setup for matrix-mult.

Input: from Alice $W \in \{-1,+1\}^{M \times N}$
Output: to Alice $K_A \in \mathbb{Z}^{M \times N}$
Output: to Bob $\{K^0_B, K^1_B\} \in \mathbb{Z}^{M \times N}$
Remark:

\[ K_A[m,n] = \begin{cases} 
K^0_B[m,n] & \text{if } W[m,n] = -1 \\
K^1_B[m,n] & \text{if } W[m,n] = 1
\end{cases} \]

1. for $m \in [M]$ do
2. \hspace{1em} for $n \in [N]$ do
3. \hspace{2em} Alice and Bob engage in ROT where:
4. \hspace{3em} Alice receives $K_A[m,n]
5. \hspace{3em} Bob receives $\{K^0_B[m,n], K^1_B[m,n]\}

- Bob gets $[Y]_B$ but does not learn $W$ or $[Y]_A$.

The above computation is performed in two phases: (1) the setup phase, shown in Algorithm 2, where Alice and Bob perform ROTs. Note that the setup phase only depends on the weight matrix which remains unchanged over a large number of inferences. Therefore this phase is performed only once and the cost is amortized among all future oblivious inferences. (2) the inference phase, shown in Algorithm 2, which is performed separately for each inference. Initially, Alice sets her output share to $W[X]_A$ (line 1) and Bob sets his share to zero (line 2). Next, they obliviously evaluate $W[X]_B$ one row at a time in the outer loop of Algorithm 2 (lines 3-14). Specifically, the $m$-th iteration of the outer loop evaluates the $m$-th row of the output as:

\[ y_n = [y_n]_A + [y_n]_B = \sum_{n=1}^{N} W[m,n] X[n,:] \]

The inner loop of Algorithm 2 (lines 6-12) computes the above summation by running OT for $n \in [N]$. After each OT invocation, Alice receives either $\mu_0 = r - [X[n,:]]_B$ or $\mu_1 = r + [X[n,:]]_B$ depending on the selection bit. It is easy to see that $\mu_0$ (known by Alice) and $-r$ (known by Bob) are the arithmetic shares of $W[m,n] [X[n,:]]_B$.

4.2. Nonlinear Layers

In this section, we outline and leverage characteristics of BNNs for oblivious inference of nonlinear layers. The cascade of batch normalization (BN) and binary activation (BA) takes input feature $y$ and returns $\hat{y} = \text{sign}(\alpha y + \beta) = \text{sign}(y + \frac{\beta}{\alpha})$, where $\alpha$ and $\beta$ are the BN parameters. Since both $\alpha$ and $\beta$ belong to the server, the parameter $\eta = \frac{\beta}{\alpha}$ can be computed offline. The GC evaluation of BN and BA only entails adding $\eta$ to $y$ and computing the sign of the result, which can be evaluated by relatively low GC cost [30]. Moreover, binary Max-Pooling can be efficiently evaluated at the bit-level. Taking the maximum in a window of binarized scalars is equivalent to performing logical $\text{OR}$ among the values, which is also efficient in GC [30].

Algorithm 2 presents our efficient protocol for oblivious evaluation of nonlinear layers in BNNs, which leverages the insights discussed above. Our protocol receives secret-shared data $[Y]_A$ and batch-normalization parameter values $\eta = \frac{\beta}{\alpha}$ from the server, as well as $[Y]_B$ from the client. It then computes $\hat{Y}$ by applying batch normalization, binary activation, and max-pooling on $Y$. Upon completion of the protocol, server and client receive $[\hat{Y}]_A$ and $[\hat{Y}]_B$, respectively, which they use to evaluate the proceeding layer.

4.3. Communication Cost

Recall that each layer execution is done via SFE protocol, where the two involved parties cooperatively compute output shares of their own. During the protocol, each party may perform certain computation, storage, or random data generation internally on their own device. In privacy-preserving computation, these type of local processes are deemed free operations. In practice, the runtime of the
Algorithm 3: Protocol for secure non-linear operations.

Input: from Alice $[Y]_A$
Input: from Alice $\eta$
Input: from Bob $[Y]_B$
Output: to Alice $[Y]_A$
Output: to Bob $[Y]_B$

Remark: $[Y]_A + [Y]_B = f([Y]_A + [Y]_B + \eta)$
Remark: $f(\cdot)$ denotes BN, BA, and optional MP.
1. Alice locally computes $[Y]_A + \eta$
2. Bob locally generates random tensor $R$
3. Alice and Bob engage in GC where:
   4. Alice inputs $[Y]_A + \eta$
   5. Bob inputs $[Y]_B$ and $R$
   6. GC computes $F = R + f([Y]_A + \eta + [Y]_B)$
   7. GC returns $F$ only to Alice
8. Alice sets $[Y]_A = F$
9. Bob sets $[Y]_B = -R$

Table 1: Communication Cost for different stages of our oblivious inference protocols. Here, $b$ is the bitwidth for arithmetic sharing\(^2\). $\kappa$ is a security parameter, and its standard value is 128 in recent literature. For max-pooling, $w$ is the window size. In cases where max-pooling is applied, the dimensionality is reduced from $L$ to $L' \approx \frac{1}{w^2}$.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Underlying Operation</th>
<th>Communication (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mat-Mult</td>
<td>$Y \leftarrow W([X]_A + [X]_B)$</td>
<td>$NbML$</td>
</tr>
<tr>
<td>BN+BA</td>
<td>$Y \leftarrow \text{sign}([Y]_A + \eta + [Y]_B)$</td>
<td>$5kbML$</td>
</tr>
<tr>
<td>MP</td>
<td>$Y \leftarrow \text{maxpool}_{w\times w}(\hat{Y})$</td>
<td>$2(w^2 - 1)\kappa ML'$</td>
</tr>
<tr>
<td>SS</td>
<td>$[\hat{Y}]_A \leftarrow \hat{Y} + R$</td>
<td>$3kbML'$</td>
</tr>
</tbody>
</table>

process is dominated by the exchange of messages between the two parties, not the internal computations. In our protocols (Algorithms 2& 3), message exchanges occur during OT or GC invocations. We provide the communication cost of our protocols in Table 1. By plugging in the parameters of this table, one can compute the total execution cost for oblivious inference of a given BNN architecture. As we show in our experiments, the communication cost is closely tied with the runtime of our protocols.

5. Training Adaptive BNN

One of the primary challenges of BNNs is to ensure inference accuracy comparable to the non-binarized model. Since the introduction of BNNs, there have been tremendous efforts to improve inference accuracy by increasing the number of channels per convolution layer [31], increasing the number of computation bits [32], or introducing new connections and nonlinear layers [33, 34], to name a few. In this paper, we improve the accuracy of the base BNN by multiplying its width, e.g., by training an architecture with twice as many neurons at each layer. In practice, specifying the appropriate width for a BNN architecture requires exploring models with various widths, which can be quite time-consuming and cumbersome. Each model with a certain width should be trained and stored separately. What aggravates the problem is that BNNs suffer from convergence issues unless the data augmentation and training hyperparameters are carefully selected [35].

A related field of research is training dynamic DNNs [36], with the goal of providing flexibility at inference time. In this realm, we find Slimmable Networks [37] quite compatible to our problem setting and adapt them to BNNs. Our goal is to train a single network with a maximum width, say $4 \times$ the base network, in a way that the model can still deliver acceptable accuracy at lower widths, e.g., $1 \times$ or $2 \times$ the base network. Once this model is trained, it can operate under any of the selected widths, thus, providing a tradeoff between accuracy and runtime.

Slimmable BNNs Definition. Let us denote the base BNN as $M_1$ and represent BNNs with $s \times$ higher width at each layer with $M_s$. Our goal is to train $M_{s_1} \subset M_{s_2} \subset M_{s_n}$ for a number of widths $\{s_i\}_{i=1}^n$. The weights of $M_{s_i}$ are a subset of the weights of $M_{s_{i+1}}$. Therefore, having $M_{s_n}$, we can configure it to operate as any $M_{s_i}$ for $i \leq n$.

Training Slimmable BNNs. For a given minibatch $X$, each subset model computes the output as $\hat{Y}_{s_i} = M_{s_i}(X)$, resulting in $\{\hat{Y}_{s_1}, \ldots, \hat{Y}_{s_n}\}$ computed by $M_{s_1}, \ldots, M_{s_n}$. The ground-truth label $Y$ is then used to compute the cumulative loss function as $\sum_{i=1}^n L(Y, \hat{Y}_{s_i})$, where $L(\cdot, \cdot)$ represents cross-entropy. The BNN weights are then updated using the standard gradient approximation rule suggested in [21].

6. Evaluations

Standard Benchmarks. We perform our evaluation on several networks trained on the CIFAR-10 dataset, shown in Table 2. The BC1 network has been evaluated by the majority oblivious inference papers [10–15, 17, 38, 39]. Other models are evaluated by XONN [14], the state-of-the-art for oblivious inference of binary networks. For brevity, we omit details about layer-wise configurations and refer curious readers to [14] for further information.

Training. For all benchmarks, we use standard backpropagation algorithm proposed by [21] to train our binary networks. We split the CIFAR10 dataset to 45k training examples, 5k validation examples, and 10k testing examples, and train each architecture for 300 epochs. We use Adam optimizer with initial learning rate of 0.001, and the learning rate is multiplied by 0.1 after 101, 142, 184 and 220 epochs. The batch size is set to 128 across all CIFAR10 training ex-
Table 2: Summary of the trained binary network architectures evaluated on the CIFAR-10 dataset.

<table>
<thead>
<tr>
<th>Arch.</th>
<th>Previous Papers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC1</td>
<td>[10], [38], [39], [11], [14], [17], [12], [15], [13]</td>
<td>7 CONV, 2 MP, 1 FC</td>
</tr>
<tr>
<td>BC2</td>
<td>[14]</td>
<td>9 CONV, 3 MP, 1 FC</td>
</tr>
<tr>
<td>BC3</td>
<td>[14]</td>
<td>9 CONV, 3 MP, 1 FC</td>
</tr>
<tr>
<td>BC4</td>
<td>[14]</td>
<td>11 CONV, 3 MP, 1 FC</td>
</tr>
</tbody>
</table>

Experiments. The training data is augmented by zero padding the images to $40 \times 40$, and randomly cropping a $32 \times 32$ window from each zero-padded image.

Evaluation Setup. The training codes are implemented in Python using the Pytorch Library. We use a single Nvidia Titan Xp GPU to train all benchmarks. We design a library for oblivious inference in C++. For implementation of OT and GC, we use the standard emp-toolkit library. To run oblivious inference, we translate the model description and trained parameters from Pytorch to the equivalent description in our C++ library. For measurements, we run our oblivious inference code on a computer with 2.2 GHz Intel Xeon CPU and 16 GB RAM. For runtime measurements, we consider two real-world network settings, namely LAN with a throughput of 1.25 GBps, round trip time of 0.25ms, and WAN with a throughput of 20 MBps, round trip time of 50ms. Reported runtimes do not include the setup time.

6.1. Evaluating Flexible BNNs

Let us start by evaluating our adaptive BNN training. We train slimmable networks with maximum $4 \times$ width of the base models presented in Table 2. During training, we re-iterate through subsets of widths $\{1 \times, 1.5 \times, \ldots, 4 \times\}$ and perform gradient updates as explained in Section 5.

Figure 4 presents the test accuracy of each network at different widths. We also report the accuracy of independently trained networks reported by XONN. The test accuracy of a particular base BNN architecture can be improved by increasing its width. Our adaptive networks obtain better accuracy than independently trained BNNs at each width. Once the adaptive network is trained, the server can provide oblivious inference service to clients, which we discuss in the following section.

6.2. Oblivious Inference

Recall that the runtime of oblivious inference is dominated by data exchange between client and server. We compare the communication cost and runtime of our custom protocol with XONN’s GC implementation in Figure 5. The horizontal axis in each figure presents the network width. The left and right vertical axes respectively show the runtime (in seconds) and communication (in Giga-Bytes). The figure shows that for all the benchmarks, the runtime and communication of our method are significantly smaller than XONN. As seen, increasing the network width results in higher communication and runtime, which is the cost we

Figure 5: Runtime and communication cost of each architecture at different widths.
pay for higher inference accuracy.

![Figure 6: Improvements in LAN runtime and communication compared to XONN. Our protocols achieve 2× to 11× in runtime and 4× to 11× communication reduction.](image)

Figure 6 summarizes the performance boost achieved by our protocols, i.e., 2× to 11× lower runtime and 4× to 11× lower communication compared to XONN. The enhancement is more significant at higher widths, which shows the scalability for our method. To illustrate the reason behind our protocol’s better performance, we focus our attention to the BC2 network at width 2.5, and show the breakdown of its communication cost in Figure 7. For the XONN protocol, most of the cost is from linear operations, which we reduce from 2.16GB to 0.15GB. In nonlinear layers, our cost is slightly more than XONN’s, i.e., 0.25GB versus 0.09GB, which is due to the extra cost of conversion between AS and GC. Overall, the total communication is reduced from 2.25GB to 0.4GB compared to XONN.

**Comparison to Non-binary Models.** Among the architectures presented in Table 2, BC1 has been commonly evaluated in contemporary oblivious inference research. In Figure 1 we compare the performance of our method to the best-performing earlier work on this benchmark. The vertical and horizontal axes in the figure represent test accuracy and runtime, hence, points to the top-left corner are more desirable. Our method achieves a better accuracy/runtime tradeoff than all contemporary work while providing flexibility. Compared to Cryptflow2 (the most recent oblivious inference framework at the time of this paper), our method achieves ∼ 2× faster inference at the same accuracy.

![Figure 7: Breakdown of communication cost at linear and nonlinear layers for BC2 network. Our protocol significantly reduces XONN’s GC-based linear layer cost, with a slight increase in nonlinear layer cost.](image)

![Figure 8: Inference runtime in WAN setting with ∼ 20 MBps bandwidth and ∼ 50 ms network delay.](image)

**Evaluation in Wide Area Network (WAN).** So far we reported our runtimes for the setting where client and server are connected via LAN, which is the most common assumption among prior work. We now extend our evaluation to the WAN setting, where the client and server are connected via LAN, which is the most common assumption among prior work. We now extend our evaluation to the WAN setting, where the bandwidth is ∼ 20MBps and the delay is ∼ 50ms. The aforesaid bandwidth and delay correspond to the connection speed between two AWS instances located in “US-West-LA-1a” and “US-East-2a”. Runtimes are reported in Figure 8, showing varying inference time from 13 to 367 seconds depending on architecture and width. The results show the great potential of BNNs for commercial use. Indeed, the delay introduced by oblivious inference might not be tolerable in many applications that require real-time response, e.g., Amazon Alexa. However, there exist many applications where guaranteeing privacy is much more crucial than runtime, and several seconds or even minutes of delay can be tolerated. We evaluate two such applications in the following section.

6.3. Evaluation on Private Tasks

In this section, we study the application of oblivious inference in face authentication and medical data analysis. Both applications involve sensitive features that the client wishes to keep secret: revealing medical data is against the HIPPA [2] regulation, and facial features can be used by malicious hackers to authenticate into the client’s personal accounts. Since we do not have access to real private data, our best choice is to simulate these tasks using similar datasets that are publicly available to the research community. We evaluate our method on FaceScrub [41,42] and Malaria Cell Infection [43] as representatives for face authentication and medical diagnosis, respectively.
Oblivious inference was shown to be conceptually practical for small sized neural networks in CryptoNets [44]. Using CryptoNets, an inference on MNIST data would take ~ 300 seconds, which motivated researchers to invest in the field. Since then, a plethora of more efficient protocols for oblivious inference have been proposed [3–13]. These works mainly focus on optimization of security primitives for oblivious inference, without making major modifications to the model.

A second line of research has been focused on identifying DNN models that are inherently amenable to secure execution protocols. Several DNN modification examples include replacing ReLU operations with square function [15, 17, 44], using dimensionality reduction at the input layer [45], and neural architecture search [16]. Concurrently, researchers in ML community have devised DNN optimization techniques such as pruning [18], quantization [20], tensor factorization [19], and binary neural networks [21]. Among the above, BNNs are especially compelling candidates for oblivious inference, since they translate linear arithmetic to bitwise operations. XONN [14] was the first work to notice the special use case of binary networks for cryptographically secure inference using GC [28], noting that XNOR operations that frequently appear in BNNs can be evaluated for free in GC.

Despite improving oblivious inference time, XONN does not completely utilize the full set of opportunities provided by BNNs. Instead of using GC as a black box, we propose a hybrid protocol where GC is only used for non-linear operations. We propose a novel protocol for matrix multiplication based on secret sharing and oblivious transfer. By exploiting the characteristics of BNN linear operations, our protocol achieves up to $11 \times$ reduction in runtime compared to XONN. A remaining challenge with BNNs is their low inference accuracy, which XONN addresses partially by brute-force training of many BNN models, and choosing the one with proper accuracy/runtime for deployment. Alternatively, we show that BNNs can be trained via the Slimmable Network training technique [37]. We provide accurate and efficient BNN benchmarks for oblivious inference, that offer a tradeoff between execution cost and inference accuracy.

Last but not least, variants of BNNs are being developed to enhance inference accuracy, opening exciting avenues for future research. Developing custom protocols to securely evaluate residual connections [33], residual activation binarization [32], and PReLU nonlinearity [34] are interesting future directions for oblivious BNN inference. Our current oblivious inference implementation does not support these operations. However, the aforementioned techniques can be integrated and tested in future work, which may or may not result in improved accuracy-runtime tradeoff.

### 8. Conclusion

This paper studies the application of binary neural networks in oblivious inference, where a server provides a privacy-preserving inference service to clients. Using this service, clients can run the neural network owned by the server, without revealing their data to the server or learning the parameters of the model. We explore favorable characteristics of BNNs that make them amenable to oblivious inference, and design custom cryptographic protocols to leverage these characteristics. In contrast to XONN [14], which uses GC to evaluate both linear and non-linear layers, we use GC only for nonlinear layers. We present a custom protocol for linear layers using OT and AS, which leads to $2 \times$ to $11 \times$ performance improvement compared to XONN. We also address the problem of low inference accuracy by training adaptive BNNs, where a single model is trained to be evaluated under different computational budgets. Finally, we extend our evaluations to computer vision tasks that perform inference on private data, i.e., face authentication and medical data analysis.
References


