1 IMPLEMENTATION DETAILS

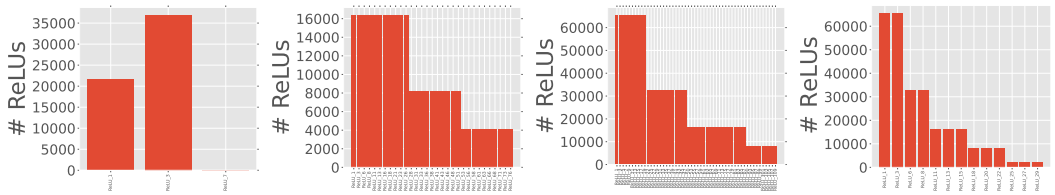
We implement the Delphi execution protocol in Python and use it as a framework to compare garbled circuits vs TABULA for private neural network inference. Our garbled circuits setup exactly follows the protocol layed out by Delphi. TABULA also follows Delphi's execution protocol, with some minor differences. Below, we describe the implementation details of our system.

Low Precision Finite Field Execution
In the online inference phase for TABULA, we follow the standard execution paradigm of quantized inference. That is, we decompose all weights/inputs into a floating point scalar and a tensor component in $\mathbb{F}_p$, where the approximation of the floating point tensor value is the product between the two. Only the $\mathbb{F}_p$ tensor component is secretly shared between the client and server, whereas the floating point scalar across network layers are public. Linear operations (matmuls and convs) multiply the scalar and perform the underlying linear operation on the $\mathbb{F}_p$ component; additions require the scalar to be the same, which necessitates making biases and skip connections have the same scales as the operand being added to. Determining the scales to make this correct is done only once ahead of time on the neural network architecture. Note that, this execution paradigm for TABULA differs from Delphi's protocol in that Delphi's uses a static 16-bit fixed point representation across all layers, which introduces more error as the scale is fixed. We note that we do make public the per-layer scalar values (1 scalar value per layer of the network). However, this leaks minimal information about the network's underlying training data, and information leakage does not increase as more inferences are performed.

Determining Finite Field Precision for Neural Network
As noted previously, TABULA decomposes each layer into a floating point scalar and the finite field tensor, which approximates the real valued tensor. During inference time, all finite field weight and activation tensors of the neural network must be representable by $\mathbb{F}_p$ or risk overflowing the finite field which would lead to wildly incorrect predictions. As TABULA relies on smaller finite fields ($<$ 16 bit), we need to choose the scales of each layer such that 1) the product of the scale and the finite field tensor for an individual layer approximates as close as possible the real valued tensor and 2) the output of that layer at inference time is representable by $\mathbb{F}_p$. Assigning precisions/scales to each layer of the network is done through a search (in our case, random search). That is, we randomly assign different scale factors to each layer of the network, quantize the layers to different precisions, run the network to obtain the maximum value seen across weights and activation values, and record accuracy; we repeat this process numerous times to obtain a pareto curve of precision vs accuracy for the target network/task. Note that the same tools used in quantization can be applied here, like neural architecture search. Further note that, in our implementation, to determine the $\mathbb{F}_p$ tensor, we use standard post-training quantization given the scale assignment (no retraining is done), and improvements can be obtained through advanced techniques.

Fused ReLU Truncation
During inference time, as the multiplication of two n bit numbers is a $2n$ bit number, it is necessary to truncate the resulting value back to n bit to continue execution, otherwise inference may eventually overflow the finite field, leading to an incorrect result. Delphi's approach with garbled circuits relies on a truncation trick introduced in SecureML, which simply truncates the secretly shared integers. Our approach with TABULA fuses the truncation operation with the ReLU operation (hence, a ReLU operation also does a truncation operation, through a single lookup to H). We note that, because the Delphi protocol with garbled circuits relies on the truncation trick introduced in SecureML, it is unable to use significantly smaller finite fields ($<$ 32 bit), as the probability of the truncation completely failing is proportional to the finite field size. An important note is that, as a matrix multiplication consists of numerous multiply and additions, the output of the operation may require many more bits to represent than the inputs. Hence, operations like batchnorm which limit the range of outputs are extremely helpful in preventing finite field overflow.

Computing the PRF
In TABULA, we choose to use the BLAKE-2s cryptographic hash function, which obtains faster speeds than older hash functions like SHA, MD5, while maintaining security. Note as mentioned in the results, computing the PRF increasingly becomes the main performance bottleneck when performing neural network inference, and hence developing a fast PRF becomes critical to performance. In our implementation, we do not parallelize computing the PRF across tensors, which is another potential area to optimize.



(a) LeNet MNIST (b) ResNet-32 Cifar10 (c) ResNet-34 Cifar100 (d) VGG-16 Cifar100

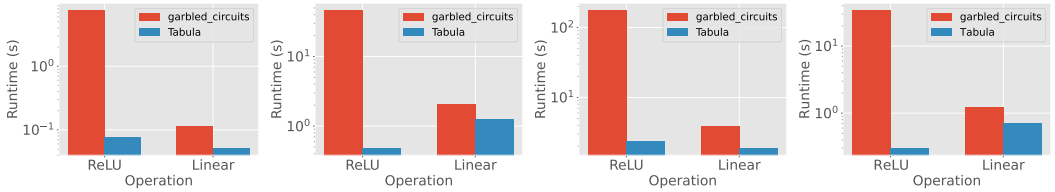
Figure 1: Number of ReLUs for different layers of various neural network architectures.

2 ReLU Breakdown

We show the number of ReLU operations in each ReLU layer across different networks, shown in Figure 1. Larger networks like ResNet have significantly more ReLU operations than smaller networks, hence, Tabula sees increasingly greater performance benefits with large neural network architectures. For networks like ResNet, we also see smaller ReLUs in the latter parts of the networks, which reduces runtime when execution nears the end of the network.

3 Runtime Breakdown

We show the runtime breakdown of garbled circuits vs Tabula for various networks, shown in Figure 2. As shown, Tabula obtains significant runtime speedups on the ReLU operation over garbled circuits. From the plots we also notice two things. First, Tabula is fast enough where the main performance bottleneck is now split between linear and ReLU operations. Secondly, Tabula attains a very small speedup when performing linear operations, which we believe is due to the fact that it uses less memory. Concretely, garbled circuits implementations use a significant amount of memory during inference time, which often overflowed to swap. Note that this is because all garbled circuits for the network need to be loaded into memory during the preprocessing phase, which incurs massive memory overheads (the alternative is loading them individually at inference time, which we found significantly degraded performance). Note that, this is yet another benefit of Tabula: memory usage on the client's device is significantly reduced and shifted to the server.



(a) LeNet MNIST (b) ResNet-32 Cifar10 (c) ResNet-34 Cifar100 (d) VGG-16 Cifar100

Figure 2: Runtime breakdown of garbled circuits vs Tabula across various network architectures.