CRYPTGPU: Fast Privacy-Preserving Machine Learning on the GPU

Sijun Tan*, Brian Knott†, Yuan Tian*, and David J. Wu*

*University of Virginia
{st8eu, yuant, dwu4}@virginia.edu

†Facebook AI Research
brianknott@fb.com

***Abstract*—We introduce CRYPTGPU, a system for privacy-preserving machine learning that implements *all* operations on the GPU (graphics processing unit). Just as GPUs played a pivotal role in the success of modern deep learning, they are also essential for realizing scalable *privacy-preserving* deep learning. In this work, we start by introducing a new interface to losslessly embed cryptographic operations over secret-shared values (in a *discrete* domain) into floating-point operations that can be processed by highly-optimized CUDA kernels for linear algebra. We then identify a sequence of "GPU-friendly" cryptographic protocols to enable privacy-preserving evaluation of both linear *and* non-linear operations on the GPU. Our microbenchmarks indicate that our private GPU-based convolution protocol is over $150\times$ faster than the analogous CPU-based protocol; for non-linear operations like the ReLU activation function, our GPU-based protocol is around $10\times$ faster than its CPU analog.**

With CRYPTGPU, we support private inference *and* private training on convolutional neural networks with over 60 million parameters as well as handle large datasets like ImageNet. Compared to the previous state-of-the-art, when considering large models and datasets, our protocols achieve a $2\times$ to $8\times$ improvement in private inference and a $6\times$ to $36\times$ improvement for private training. Our work not only showcases the viability of performing secure multiparty computation (MPC) *entirely* on the GPU to enable fast privacy-preserving machine learning, but also highlights the importance of designing new MPC primitives that can take full advantage of the GPU's computing capabilities.

I. INTRODUCTION

Deep learning has enabled numerous applications in the form of digital voice assistants, video monitoring and surveillance systems, and even systems for disease diagnosis and treatment planning. But these new and exciting applications raise challenging questions regarding user privacy. After all, modern machine learning algorithms are largely data-driven and training image recognition, speech recognition, or disease predictor systems all rely on aggregating and analyzing sensitive user data. Even model inference raises privacy concerns as increasingly often, voice or video recordings from a mobile or IoT device are outsourced to the cloud for analysis.

To address some of the privacy challenges associated with the widespread deployment of deep learning technologies, a number of works [1, 2, 3, 4, 5, 6] in the last few years have introduced cryptographic frameworks based on secure multiparty computation (MPC) [7, 8] to enable *privacy-preserving deep learning* (see Section V for a more comprehensive survey). At a high level, MPC protocols allow a set of mutually-distrusting parties to compute an arbitrary function over secret inputs such that at the end of the computation, the parties only learn the output of their computation, and nothing more. In particular, all information about other parties' inputs are completely hidden (up to what could be inferred based on the output[1]).

While there have been considerable advances in the concrete efficiency of MPC protocols, current approaches remain computationally expensive and do not scale well to the types of neural networks typically used in modern machine learning systems. Until recently, cryptographic protocols for private inference over deep neural networks have been limited to small datasets such as MNIST [11] or CIFAR [12]. In contrast, the current baseline for object recognition is ImageNet [13], a dataset that is over $1000\times$ larger than CIFAR/MNIST and contains 1000 different classes (compared to just 10 classes for MNIST and CIFAR). Similarly, state-of-the-art deep learning models for computer vision such as ResNet-152 [14] contain over 150 layers and over 60 million parameters. In contrast, most protocols for privacy-preserving machine learning have been constrained to relatively shallow networks with just tens of layers and a few hundred thousand parameters.

Recently, two systems FALCON [6] and CRYPTFLOW [5] have made considerable headway towards scalable privacy-preserving machine learning. For the first time, they demonstrate the ability to perform privacy-preserving machine learning at the scale of ImageNet (or Tiny ImageNet [15] in the case of FALCON) and with much larger models (e.g., AlexNet [16], VGG-16 [17], and the ResNet family of models [14]). In spite of these advances, there still remains considerable overhead: for example, private training of AlexNet on Tiny ImageNet is estimated to still take *over a year* using FALCON. CRYPTFLOW currently only supports private inference and not private training. Both works argue that hardware acceleration with graphics processing units (GPUs) will be essential for scaling up privacy-preserving deep learning, especially in the case of private training.

The importance of GPU acceleration. GPUs and hardware acceleration have played a critical role in the evolution of modern deep learning. Today, convolutional neural networks (CNNs) have become a staple for modern computer vision.

[1]There are settings where even learning the exact output is problematic and can reveal compromising information about other parties' inputs. Techniques like differential privacy [9, 10] provide a defense against these types of attacks. We discuss this in greater detail in Section V.

However, in the immediate years following their introduction in the seminal work of LeCun et al. [18], CNNs did not see widespread adoption. This was in large part due to the high computational costs of the backpropagation training algorithm. Starting the mid-2000s, several works [19, 20] showed that CNN training could be greatly accelerated through the use of graphics processing units (GPUs). This culminated with the breakthrough moment when Krizhevsky et al. [16] introduced "AlexNet" and won the ImageNet Large Scale Visual Recognition Challenge in 2012 using a large CNN *trained entirely* on the GPU. Since AlexNet, CNNs have become a mainstay of computer vision. Modern machine learning frameworks like PyTorch [21] and TensorFlow [22] all support and rely heavily on not only GPUs, but even custom-designed application-specific integrated circuits (ASICs) such as Google's tensor processing unit [23].

Privacy-preserving machine learning on the GPU. Hardware acceleration has become a core component for evaluating and training deep learning models. Given that MPC protocols necessarily incur a non-zero overhead on top of the plaintext computation, it is essential for cryptographic protocols to be able to leverage GPU acceleration in order to have *any* chance of scaling up to support training and inference over deep models. After all, if we are bound to CPU-based computations (as nearly *all* existing MPC frameworks have), then it is infeasible to even run the machine learning algorithm on *plaintext* data.

A. Our Contributions

In this work, we introduce CrypTGPU, a new cryptographic MPC framework built on top of PyTorch and CrypTen [24] where *all* of the cryptographic operations (both linear *and* non-linear) are implemented on the GPU. CrypTGPU operates in the standard 3-party setting where we assume that all inputs are secret-shared across three non-colluding servers who execute the MPC protocol. The inputs are secret shared using a 2-out-of-3 *replicated secret sharing* scheme [25, 26] (see Section III for the full details). Our system provides security against a single semi-honest corruption. We describe our threat model formally in Section III-A.

CrypTGPU can perform private inference over modern computer vision models such as ResNet-152 on ImageNet images in just over 25s (2.3× faster than the previous state-of-the-art CrypTFlow [5]). For smaller networks like AlexNet, private inference over ImageNet requires just 1.5s.

Further improvements to the costs of private training are possible if we consider *batch inference*, which also benefits from GPU parallelism. For example, batch inference over ResNet-152 reduces the cost of private inference from 25s for a single image to 13.2s per image when amortized over a batch of 8 images.

For private training (which has a greater potential to benefit from GPU acceleration), we demonstrate a 36× speed-up for private training of AlexNet on the Tiny ImageNet database compared to Falcon. Whereas it would have taken over a year to privately train Falcon on Tiny ImageNet, our GPU-accelerated system would be able to do so in just over a week (see Section IV-B). Beyond these performance results, our work highlights the potential of leveraging GPUs to accelerate privacy-preserving deep learning in much the same way GPUs have dramatically accelerated standard deep learning. Our work also highlights the importance of developing new types of cryptographic protocols that are "GPU-friendly" and can take advantage of the parallelism provided by GPUs.

Cryptography on the GPU. While NVIDIA's CUDA (Compute Unified Device Architecture) platform [27] supports general-purpose computations on the GPU, directly translating code written for the CPU onto the GPU is unlikely to translate to immediate performance gains. The architectural differences between the CPU and the GPU introduce several additional hurdles that must be overcome in order to have an efficient implementation:

- **Leveraging existing CUDA kernels.** The first challenge is that highly optimized CUDA kernels for computing deep learning primitives (i.e., convolutions, pooling, matrix multiplication) are designed to operate on *floating-point* inputs, and there does not currently exist kernels for computing on integer values. In MPC, we typically compute over discrete objects (i.e., ring or field elements). To leverage optimized kernels for these basic primitives, we need a way to *embed* the integer-valued cryptographic operations into (64-bit) floating-point arithmetic that can in turn be operated on by these kernels. CrypTGPU enables this by introducing a new abstraction called a `CUDALongTensor` that models tensors (i.e., multi-dimensional arrays) of integer values, but seamlessly translates the integer-valued computations into a corresponding set of floating-point computations. We describe our construction in Section II-B. The Delphi system encountered a similar challenge, but as we discuss in Remark II.3, their solution does not extend well to our setting. A critical difference is that Delphi considers private inference where the model is *public* while in this work, we assume that the model is also hidden (i.e., secret-shared).
- **"GPU-friendly" cryptography.** The GPU architecture is optimized for performing a large number of *simple* computations on *blocks* of values. This means that operations like component-wise addition and multiplication of vectors/matrices are fast while operations that involve large numbers of conditional statements are slower. While there is support for integer addition and multiplication, operations like computing a modular reduction by a prime incurs considerably more overhead [27]; for instance, we observed a 40× difference in the running time of point-wise addition vs. point-wise modular reduction. Thus, when choosing and designing cryptographic protocols for the GPU, one must carefully calibrate them for the architecture. Protocols like Yao's garbled circuits [28] are less well-suited for taking advantage of GPU parallelism compared to a vectorized secret-sharing-based protocol. Similarly, protocols that require extensive finite field arithmetic (and thus, require

modular reductions) will incur more overhead on the GPU compared to protocols that only rely on arithmetic modulo a power of 2. We also design protocols for common non-linear functions (e.g., exponentiation and division) that are specifically optimized for our particular setting. We describe the cryptographic protocols we use in Section III.

Systematic evaluation of GPU-based MPC. We present a comprehensive and systematic evaluation of CRYPTGPU to quantify the advantages of a GPU-based MPC protocol and compare against previous protocols for privacy-preserving machine learning. We specifically measure the performance of our private training and inference protocols on a wide range of object recognition models (e.g., LeNet [18], AlexNet [16], and the ResNet family of networks [14]) and datasets (e.g., MNIST [11], CIFAR-10 [12], and ImageNet [13]). We describe our experimental methodology and measurements in Section IV.

We also collect fine-grained measurements to understand how the computational costs are split across the different layers of a network. For instance, in CPU-based systems like FALCON [6], the linear layers account for 86% to 99% of the overall computational costs of private training.[2] On the same model/datasets, our GPU-based approach evaluates the same linear layers with a $25\times$ to $72\times$ speed-up; this is a major source of the performance advantage of CRYPTGPU compared to previous systems. Consequently, the costs of our private training protocol is more evenly split between evaluating linear layers and non-linear layers. We provide the full details in Section IV-B and Table V.

In Section IV-C, we report microbenchmarks to quantify the performance advantages of using the GPU to execute all of the MPC protocols. For instance, we show that evaluating convolutions on secret-shared data (with secret-shared kernels) on the GPU is over $150\times$ faster than the corresponding protocol on the CPU. Even for non-linear operations like the ReLU (rectified linear unit) function, using a GPU-based MPC protocol still yields a $10\times$ speed-up over the same underlying CPU-based protocol.

Finally, since our MPC protocol represents real-valued inputs using a fixed-point encoding, and moreover, some of our protocols rely on approximations to non-linear functions, we also compare the accuracy of our private inference and private training algorithms to the analogous plaintext algorithms. As we show in Section IV-D, for the models and datasets we consider in this work, the behavior of our privacy-preserving algorithms closely matches their plaintext analogs.

An ML-friendly approach. One of the guiding principles behind our system design is to make it friendly for machine learning researchers to use. We build our system on top of CRYPTEN [24], which is itself built on top of the popular machine learning framework PyTorch [21]. Effectively, our work (much like CRYPTEN) provides a new cryptographic *back end* that supports computations on secret-shared values while retaining a similar front end as PyTorch. In fact, we note that our work on developing the `CUDALongTensor` module has already been integrated as part of CRYPTEN to support privacy-preserving GPU computations [24].

[2]While linear layers are simpler to evaluate from a cryptographic perspective (in comparison to non-linear layers), the size of the linear layers is typically much larger than that of the non-linear layers.

II. SYSTEM OVERVIEW

Similar to previous works on constructing efficient protocols for privacy-preserving machine learning [2, 3, 6, 5, 29, 30] (see also Section V), we assume that the data and model are (arbitrarily) partitioned across three parties. For example, the three parties could be three independent organizations seeking to collaboratively train a model on their joint data without revealing their inputs to each other. Our system is also applicable in the "server-aided" setting [31], where a group of (arbitrarily-many) clients seek to train a joint model on their data (or evaluate a secret-shared model on private inputs). In the server-aided setting, the clients first secret share their inputs to three independent cloud-service providers, who in turn run the cryptographic protocol on the secret-shared inputs. We design our protocols to provide security against a single semi-honest corruption. We provide a formal description of our threat model in Section III-A.

A. Background

Our starting point in this work is the CRYPTEN privacy-preserving machine learning framework [24]. CRYPTEN is built on top of the widely-used machine-learning framework PyTorch [21]. We adapt the basic architecture of CRYPTEN, and make modifications to support three-party protocols based on replicated secret sharing. We describe the main system architecture below.

GPU architecture. GPUs, and more recently, ASICs like Google's tensor processing units [23], have played a critical role in scaling up modern deep learning. These specialized hardware platforms support massive parallelism, making them well-suited for performing standard linear algebraic operations (e.g., convolutions or average pooling) as well as point-wise evaluation of functions on large blocks of neurons (e.g., evaluating an activation function or performing batch normalization). Popular frameworks for deep learning frameworks such as PyTorch [21] and TensorFlow [22] natively support computations on both the CPU and GPU.

CUDA is a parallel computing platform developed by NVIDIA for general-purpose computing on GPUs [27]. For deep learning in particular, CUDA libraries such as `cuBLAS` [32] and `cuDNN` [33] provide highly-optimized implementation for a wide-range of standard primitives such as convolutions, pooling, activation functions, and more. These libraries are designed for *floating-point* computations and do *not* support integer-valued analogs of these operations. Since cryptographic protocols typically operate over *discrete* spaces (e.g., a 64-bit ring) where the underlying algebra is implemented using integer-valued computations, one cannot directly translate an existing protocol to the GPU.

PyTorch. PyTorch [21] is a popular open-source machine learning framework designed for prototyping, implementing, and deploying deep neural networks. The PyTorch front end supports many standard neural network layers (e.g., convolutions, pooling, activation functions, etc.) as well as features such as automatic differentiation and gradient computation. The PyTorch back end natively supports computation on both CPUs as well as GPUs. This flexibility enables users to train complex models without needing to worry about the finer details of backpropagation. It also allows users to take advantage of GPU acceleration without needing to interface with low-level CUDA kernels. PyTorch also provides library support for distributing computations across multiple devices and/or GPUs.

Data in PyTorch is organized around *tensors*, which provide a general abstraction for n-dimensional arrays. PyTorch provides an expressive API for computing on and applying transformations to tensors. Especially importantly in our case, the PyTorch back end natively and seamlessly leverages GPU acceleration for tensor computations.

CrypTen. CrypTen [24] is a recent framework built on top of PyTorch for privacy-preserving machine learning. CrypTen provide a secure computing *back end* for PyTorch while still preserving the PyTorch front end APIs that enables rapid prototyping and experimentation with deep neural networks.

The main data abstraction in CrypTen is the `MPCTensor`, which functions like a standard PyTorch tensor, except the values are *secret shared* across multiple machines. Internally, CrypTen uses n-out-of-n additive secret sharing. For bilinear operations such as convolutions and matrix multiplications, CrypTen uses arithmetic secret sharing over a large ring (e.g., $\mathbb{Z}_{2^{64}}$), while for evaluating non-linear operations like an activation function, it uses Boolean secret sharing. CrypTen uses the ABY share-conversion techniques [34] to convert between arithmetic shares and Boolean shares.

CrypTen supports general n-party computation and provides security against a single semi-honest corruption. At the cryptographic level, elementary arithmetic operations are handled using Beaver multiplication triples [35], Boolean circuit evaluation is implemented using the Goldreich-Micali-Wigderson (GMW) protocol [7], and low-degree polynomial approximations are used for most non-linear operations. We note that while our system builds on CrypTen, we work in a 3-party model where parties compute using *replicated secret shares* (as in [26]). We describe this in Section III.

B. System Design and Architecture

The design of CrypTGPU is centered around the following principles:

- **Leverage existing CUDA kernels for linear algebra.** As mentioned in Section II-A, highly-optimized CUDA kernels exist for most linear algebra operations encountered in deep learning. However, these kernels only support computations on floating-point values and are not directly applicable for computing on discrete structures common in cryptographic protocols. Thus, we seek a way to keep all of the computation on the GPU itself.
- **Keep *all* computations on the GPU.** While some previous works on private machine learning [36, 4] show how to leverage the GPU for computing linear and bilinear functions, they then move the data out of the GPU to evaluate non-linear functions. In this work, we seek to keep *all* of the computations on the GPU, and as we show in Section IV-C, even computing non-linear functions can benefit greatly from GPU acceleration, provided that they are implemented using "GPU-friendly" cryptographic protocols (i.e., protocols that primarily rely on point-wise or component-wise vector operations).

Floating point computations. The cryptographic core of CrypTGPU relies on (additive) replicated secret sharing over the 64-bit ring $\mathbb{Z}_{2^{64}}$. Computing bilinear functions such as convolutions over secret-shared values essentially correspond to the parties running an analogous local operation on their shares, followed by a communication step (see Section III). Our goal is to take advantage of the GPU to accelerate each party's local computation on their individual shares. As noted in Section II-A, existing GPU libraries for linear algebra only support computation over 64-bit floating point values. Thus, to take advantage of GPU support for these operations, we need to embed the ring operations over $\mathbb{Z}_{2^{64}}$ (or equivalently, 64-bit integer operations) into 64-bit *floating point* operations.

Integer operations using floating-point arithmetic. Our approach for embedding 64-integer operations into 64-bit floating point operations relies on the following observations:

- **Exact computation for small values.** First, 64-bit floating point values have 52 bits of precision and can exactly represent *all* integers in the interval $[-2^{52}, 2^{52}]$. This means that for all integers $a, b \in \mathbb{Z} \cap [-2^{26}, 2^{26}]$, we can compute the product ab using their floating-point representations and still recover the correct value *over the integers*.
- **Bilinearity.** Operations like matrix multiplication and convolutions are *bilinear*. This means that for any choice of inputs $\mathbf{A}_1, \mathbf{A}_2, \mathbf{B}_1, \mathbf{B}_2$,

$$(\mathbf{A}_1 + \mathbf{A}_2) \circ (\mathbf{B}_1 + \mathbf{B}_2) = \\ \mathbf{A}_1 \circ \mathbf{B}_1 + \mathbf{A}_2 \circ \mathbf{B}_1 + \mathbf{A}_2 \circ \mathbf{B}_1 + \mathbf{A}_2 \circ \mathbf{B}_2,$$

  where $\circ$ denotes an arbitrary bilinear operation. Suppose now that we rewrite an input as an expansion in a smaller base; for example, we might write $\mathbf{A} = \mathbf{A}_0 + 2^{16}\mathbf{A}_1$ and $\mathbf{B} = \mathbf{B}_0 + 2^{16}\mathbf{B}_1$. Bilinearity ensures that $\mathbf{A} \circ \mathbf{B}$ can be expressed as a linear combination of the pairwise products $\mathbf{A}_0 \circ \mathbf{B}_0$, $\mathbf{A}_0 \circ \mathbf{B}_1$, $\mathbf{A}_1 \circ \mathbf{B}_0$, and $\mathbf{A}_1 \circ \mathbf{B}_1$. Computing $\mathbf{A} \circ \mathbf{B}$ from the pairwise products only requires *element-wise* additions and scalar multiplications.
- **CUDA kernels for element-wise operations.** To complete the puzzle, we note that there are optimized CUDA kernels for performing component-wise addition and scalar multiplication on 64-bit *integer* values.

To evaluate a bilinear operation $\circ$ like matrix multiplication or convolution (which do *not* have integer kernels), CRYPTGPU first decomposes each of the inputs $\mathbf{A}, \mathbf{B} \in \mathbb{Z}_{2^{64}}^{n \times m}$ into smaller inputs $\mathbf{A}_1, \ldots, \mathbf{A}_k, \mathbf{B}_1, \ldots, \mathbf{B}_k \in \mathbb{Z}_{2^w}^{n \times m}$ where $\mathbf{A} = \sum_{i=1}^{k} 2^{(i-1)w} \mathbf{A}_i$. Then, it computes the k^2 products $\mathbf{A}_i \circ \mathbf{B}_j$ using floating-point arithmetic on the GPU. As long as the entries of $\mathbf{A}_i \circ \mathbf{B}_j$ do not exceed 2^{52} in magnitude, all of these pairwise products are computed exactly. Finally, each component of the pairwise product is re-interpreted as a 64-bit integer. Computing $\mathbf{A} \circ \mathbf{B}$ from the pairwise products $\mathbf{A}_i \mathbf{B}_j$ amounts to evaluating a linear combination of tensors, which can be done efficiently using existing CUDA kernels for 64-bit integer operations. Note that since the final operations are taken modulo 2^{64}, it suffices to compute only the products $\mathbf{A}_i \mathbf{B}_j$ where $w(i + j - 2) < 64$.

When performing computations using floating-point kernels, CRYPTGPU decomposes each input into $k = 4$ blocks, where the values in each block are represented by a $w = 16$-bit value. For this choice of parameters, each bilinear operation is expanded into 10 pairwise products.

Remark II.1 (Smaller Number of Blocks)**.** While it may be tempting to decompose 64-bit values into $k = 3$ blocks, where each block consists of 22-bit values, this compromises correctness of our approach. Namely, correctness of the computation is guaranteed only if the entries in each of the intermediate pairwise products $\mathbf{A}_i \circ \mathbf{B}_j$ do not exceed the 52-bits of available floating-point precision. If the entries of $\mathbf{A}_i$ and $\mathbf{B}_j$ are 22 bits, then the entries in a single multiplication between an element in $\mathbf{A}_i$ and $\mathbf{B}_j$ will already be 44 bits. If we are evaluating a convolution (or matrix multiplication) where each output component is a sum of $2^8 = 256$ values, this exceeds the available precision and triggers an arithmetic overflow. This is problematic for larger networks. Using 16-bit blocks, we can handle bilinear operations involving up to 2^{20} intermediate products, which is sufficient for our applications.

Remark II.2 (Overhead of Block-wise Decomposition)**.** While decomposing each bilinear operation on integer values into $O(k^2)$ floating-point operations on same-sized inputs can appear costly, CRYPTGPU takes advantage of GPU parallelism to mitigate the *computational* overhead. Namely, for convolutions, CRYPTGPU uses group convolution (`cudnnConvolutionForward`) to compute the convolutions in parallel. Similarly, for matrix multiplications, CRYPTGPU uses a batch matrix multiplicative kernel (`cublasSgemm`) to multiply matrices in parallel. We observe that for small inputs (e.g., 64×64 inputs), this approach only incurs a modest $2\times$ overhead (compared with evaluating a single convolution of the same size) and increases to roughly $9\times$ for larger 224×224 inputs.

While the computational overhead of our embedding is partially mitigated through parallelism, this approach does increase the memory requirements of our protocol. This does not have a significant effect on privacy-preserving inference, but it does limit the batch size we can handle during privacy-preserving training (recall that during training, a single iteration of the optimization algorithm processes a *batch* of instances). Scaling up to support larger batch sizes during privacy-preserving training would likely necessitate distributing the computation across multiple GPUs rather than a single GPU (as is also the case for training deep models in the clear).

Remark II.3 (Comparison with DELPHI)**.** The DELPHI system [4] leverage GPUs for evaluating convolutions on secret-shared inputs in their private inference system. In their setting, the parameters are chosen so that the outputs of the convolution are always within the interval $[2^{-52}, 2^{52}]$, and as such, the existing floating-point kernels for convolution can be used without incurring any floating-point precision issues. In particular, DELPHI uses a 32-bit ring and 15 bits of fixed-point precision. The system works in the setting where the model parameters are assumed to be *public*: namely, the convolution kernels are *not* secret-shared. In this way, convolutions are evaluated between a *plaintext* value and a secret-shared value, which ensures that the resulting outputs are bounded. In our setting, both the model and the inputs are secret-shared so we cannot directly embed the integer-valued operations into 64-bit floating-point computations. In fact, as we discuss in Section IV-C, to have sufficient precision when scaling up to deeper models and larger datasets, it is often necessary to use a larger ring (i.e., a 64-bit ring) for the arithmetic secret sharing.

The `CUDALongTensor` abstraction. CRYPTGPU provides a new abstraction called a `CUDALongTensor` for embedding 64-bit integer-valued operations into 64-bit floating-point arithmetic. Similar to CRYPTEN's `MPCTensor`, the `CUDALongTensor` abstractly represents a secret-shared tensor of 64-bit *integers* and is backed by a standard PyTorch tensor of 64-bit integers. In the back end, whenever an elementary operation needs to be evaluated on the underlying tensor, CRYPTGPU proceeds as follows:

- If optimized CUDA kernels exist for evaluating the chosen operation on integer-valued tensors (e.g., point-wise addition or point-wise multiplication), then the corresponding CUDA kernel is directly invoked.
- For bilinear operations where optimized CUDA kernels only exist for computations on floating-point inputs (e.g., convolutions, matrix multiplications), then CRYPTGPU applies the above technique of first decomposing the input into $k = 4$ tensors of 16-bit values, computing all necessary $O(k^2)$ pairwise products of the resulting blocks (using the floating point kernel), and re-combines the pairwise products to obtain the final output.

III. THREAT MODEL AND CRYPTOGRAPHIC DESIGN

In this section, we provide a formal specification of our threat model and a description of the private inference and training functionalities we develop. We then describe the cryptographic sub-protocols we use to construct our privacy-preserving training and inference protocols.

We begin by introducing the notation we use in this work. For a finite set S, we write $x \xleftarrow{R} S$ to denote that x is drawn

uniform at random from S. We use boldface letters (e.g., $\mathbf{x}, \mathbf{y}$) to denote vectors and use non-boldface letters (e.g., x_i, y_i) to denote their components. We denote our three parties by P_1, P_2, P_3. To simplify notation, whenever we use an index $i \in \{1,2,3\}$ to denote a party (or a share), we write $i-1$ and $i+1$ to denote the "previous" party and the "next" party, respectively. For example, P_{3+1} refers to P_1.

We say that a function f is negligible in a parameter λ if $f(\lambda) = o(\lambda^{-c})$ for all $c \in \mathbb{N}$. We say an algorithm is efficient if it runs in probabilistic polynomial-time in the length of its input. We say that two families of distributions $\mathcal{D}_1 = \{\mathcal{D}_{1,\lambda}\}_{\lambda\in\mathbb{N}}$ and $\mathcal{D}_2 = \{\mathcal{D}_{2,\lambda}\}_{\lambda\in\mathbb{N}}$ are computationally indistinguishable (i.e., $\mathcal{D}_1 \overset{c}{\approx} \mathcal{D}_2$) if no efficient adversary can distinguish samples from $\mathcal{D}_1$ and $\mathcal{D}_2$ except with negligible probability.

A. Threat Model

Similar to several recent 3-party protocols [26, 37, 2, 6], we design our system in the honest-majority model. Moreover, we focus on semi-honest adversaries. Namely, we assume that each of the three computing parties follow the protocol, but may *individually* try to learn information about other parties' inputs. Formally, we consider the standard simulation-based notion of security in the presence of semi-honest adversaries [38, 39]:

Definition III.1 (Semi-Honest Security)**.** Let $f\colon (\{0,1\}^n)^3 \to (\{0,1\}^m)^3$ be a randomized functionality and let π be a protocol. We say that π securely computes f in the presence of a single semi-honest corruption if there exists an efficient simulator $\mathcal{S}$ such that for every corrupted party $i \in \{1,2,3\}$ and every input $\mathbf{x} \in (\{0,1\}^n)^3$,

$$\{\mathsf{output}^\pi(\mathbf{x}), \mathsf{view}_i^\pi(\mathbf{x})\} \overset{c}{\approx} \{f(\mathbf{x}), \mathcal{S}(i, x_i, f_i(\mathbf{x}))\}$$

where $\mathsf{view}_i^\pi(\mathbf{x})$ is the view of party i in an execution of π on input $\mathbf{x}$, $\mathsf{output}^\pi(\mathbf{x})$ is the output of all parties in an execution of π on input $\mathbf{x}$, and $f_i(\mathbf{x})$ denotes the i^{th} output of $f(\mathbf{x})$.

Computing on secret-shared values. In this work, we consider two main settings: private inference and private training on secret-shared inputs. We use standard 3-out-of-3 additive secret sharing as well as 2-out-of-3 replicated secret sharing. Abstractly, we model both types of secret sharing as a pair of algorithms $(\mathsf{Share}, \mathsf{Reconstruct})$ with the following properties:

- On input $x \in \{0,1\}^n$, the share algorithm $\mathsf{Share}(x)$ outputs a tuple of three shares (x_1, x_2, x_3).
- The reconstruction algorithm $\mathsf{Reconstruct}(S)$ takes a set of shares T and outputs a value $x \in \{0,1\}^n$ if successful and $\perp$ if not.

Correctness of a threshold secret sharing scheme with threshold t says that for any subset of shares $T \subseteq \mathsf{Share}(x)$ of size at least t, $\mathsf{Reconstruct}(T) = x$. Perfect security says that there exists a probabilistic polynomial-time simulator $\mathcal{S}$ such that for every subset $T \subseteq \{1,2,3\}$ where $|T| < t$ and every input $x \in \{0,1\}^n$,

$$\{(x_1, x_2, x_3) \leftarrow \mathsf{Share}(x) : (x_i)_{i\in T}\} \equiv \{\mathcal{S}(1^n, T)\}.$$

We now formally define our notion of private inference and private training on secret-shared inputs:

- **Private inference:** Inference is the problem of evaluating a trained model M on an input x. We denote this operation as $\mathsf{Eval}(M, x)$. In private inference, the ideal functionality f maps secret shares of an input x and a model M to a secret share of the output $\mathsf{Eval}(M, x)$. Namely, on input $((M_1, x_1), (M_2, x_2), (M_3, x_3))$, the ideal functionality outputs $\mathsf{Share}(\mathsf{Eval}(M, x))$ where $M \leftarrow \mathsf{Reconstruct}(\{M_1, M_2, M_3\})$ and $x \leftarrow \mathsf{Reconstruct}(\{x_1, x_2, x_3\})$. In particular, a private inference protocol ensures privacy for the model M, the input x, and the output $\mathsf{Eval}(M, x)$.
- **Private training:** In private training, the goal is to run a training algorithm Train on some dataset D. In this case, the ideal functionality f maps secret shares of the dataset (D_1, D_2, D_3) to a secret share of the model $\mathsf{Share}(\mathsf{Train}(D))$ where $D \leftarrow \mathsf{Reconstruct}(D_1, D_2, D_3)$. In this case, each party individually learn nothing about the input dataset D or the resulting learned model $\mathsf{Train}(D)$.

B. Cryptographic Building Blocks for Private Inference

We now describe the main MPC building blocks we use for private inference on deep neural networks. First, we decompose the neural network inference algorithm into a sequence of elementary operations: linear/pooling/convolution layers and activation function evaluation (ReLU). To obtain our protocol π for computing the ideal functionality for private inference, we *sequentially* compose the semi-honest secure protocols for realizing each of the elementary operations. Correctness and semi-honest security of the overall protocol then follows by correctness and security of the underlying sub-protocols together with the sequential composition theorem [38].

"GPU-friendly" cryptography. As alluded to in Sections I-A and II-B, we seek cryptographic protocols that are particularly amenable to GPU acceleration. For example, protocols that involve conditionals (such as garbled circuits [28]) or require extensive finite field arithmetic are more challenging to support efficiently on the GPU. For this reason, we focus primarily on secret-sharing based protocols and work over a ring with a power-of-two modulus. In the following description, we elect to use cryptographic protocols where the underlying implementations vectorize and whose evaluation can be expressed primarily in terms of point-wise or component-wise operation on blocks of data.

Secret sharing. We work over the ring $\mathbb{Z}_n$ where $n = 2^k$ is a power of 2. In our specific implementation, $k = 64$. To secret share a value $x \in \mathbb{Z}_n$, sample shares $x_1, x_2, x_3 \xleftarrow{R} \mathbb{Z}_n$ such that $x_1 + x_2 + x_3 = x$. Following Araki et al. [26], our default sharing is a 2-out-of-3 "replicated secret sharing" [25] where each party holds a pair of shares: P_1 has (x_1, x_2), P_2 has (x_2, x_3), and P_3 has (x_3, x_1). We denote this by $[\![x]\!]^n = (x_1, x_2, x_3)$. In some cases, we will also consider a 3-out-of-3 additive secret sharing scheme where party P_i holds x_i (but *none* of the other shares).

Fixed point representation. Machine learning algorithms natively operate on real (i.e., floating-point) values while the most efficient cryptographic protocols are restricted to computations over discrete domains such as rings and finite fields. Following previous work, we use a fixed-point encoding of all values occurring in the computation, and then embed the integer-valued fixed-point operations in the ring $\mathbb{Z}_n$. Specifically, if we consider a fixed-point encoding with t bits of precision, a real value $x \in \mathbb{R}$ is represented by the integer $\lfloor x \cdot 2^t \rceil$ (i.e., the nearest integer to $x \cdot 2^t$). The ring modulus n is chosen to ensure no overflow of the integer-valued fixed-point operations. CRYPTGPU sets $n = 64$; we discuss this choice in detail in Section IV-D.

Protocol initialization. In the following description, we assume that the parties have many independent secret shares of 0. This will be used for "re-randomization" during the protocol execution. We implement this using the approach of Araki et al. [26]. Specifically, let F be a pseudorandom function (PRF). At the beginning of the protocol, each party P_i samples a PRF key k_i and sends k_i to party P_{i+1}. The j^{th} secret share of 0 is the triple (z_1, z_2, z_3) where $z_i = F(k_i, j) - F(k_{i-1}, j)$.

Linear operations. Linear operations on secret-shared data only require local computation. Namely, if $\alpha, \beta, \gamma \in \mathbb{Z}_n$ are public constants and $[\![x]\!]^n, [\![y]\!]^n$ are secret-shared values, then $[\![\alpha x + \beta y + \gamma]\!]^n = (\alpha x_1 + \beta y_1 + \gamma, \alpha x_2 + \beta y_2, \alpha x_3 + \beta y_3)$. Each of the parties can compute their respective shares of $[\![\alpha x + \beta y + \gamma]\!]^n$ from their shares of $[\![x]\!]^n$ and $[\![y]\!]^n$ and the public coefficients α, β, γ.

Multiplication. To multiply two secret-shared values $[\![x]\!]^n = (x_1, x_2, x_3), [\![y]\!]^n = (y_1, y_2, y_3)$, each party P_i locally computes $z_i = x_i y_i + x_{i+1} y_i + x_i y_{i+1}$. By construction, $z_1 + z_2 + z_3 = xy \in \mathbb{Z}_n$. This yields a 3-out-of-3 additive sharing of z. To obtain replicated shares of z, party P_i sends P_{i+1} a blinded share $z_i + \alpha_i$, where $(\alpha_1, \alpha_2, \alpha_3)$ is a fresh secret-sharing of 0 (derived from the PRF as described above).

Since x, y are fixed-point encodings, the parties additionally need to *rescale* z after computing the product (i.e., divide it by the scaling factor 2^t). In this work, we use the truncation protocol Π_{trunc1} from ABY^3 [2] to implement this procedure. We note that Mohassel and Rindal propose two versions of the share truncation protocol: a two-round protocol Π_{trunc1} that only relies on elementary arithmetic operations and a one-round protocol Π_{trunc2} that relies on precomputed "truncation tuples". While generating the truncation tuples can be done in a separate offline phase, doing so requires implementing a Boolean bit extraction circuit over secret-shared values. In contrast, Π_{trunc1} relies exclusively on arithmetic operations, and naturally extends to our tensor-based computing model. For this reason, we use the two-round truncation protocol Π_{trunc1} in our implementation. This has the added advantage that we avoid a separate (and potentially expensive) preprocessing step. Both of these share-truncation protocols are not exact and may introduce 1 bit of error in the *least* significant bit of the secret-shared value (i.e., with t bits of fixed-point precision, the error introduced is bounded by 2^{-t}). We provide an empirical assessment of the error (and resulting model accuracy) in Section IV-D.

Convolutions and matrix multiplication. The above protocols for computing linear functions as well as products of secret-shared values directly vectorize to yield protocols for computing linear functions on tensors as well as bilinear operations like matrix multiplication and convolution. Linear functions on secret-shared tensors only require local computation. Bilinear operations on secret-shared tensors like matrix multiplications and convolutions are implemented by computing three separate products (as described in the multiplication protocol above). These computations over secret-shared tensors directly map to analogous computations on local shares, so we can take advantage of existing highly-optimized CUDA kernels for evaluating these operations via the technique from Section II-B.

As in several previous systems (e.g., [1, 2, 5]), when we compute products of secret-shared tensors, we only apply the truncation protocol to the *result* of the product and *not* after each individual multiplication. This has a significant impact on the performance of the protocol for two reasons: (1) we can use existing CUDA kernels optimized for matrix products and convolutions without needing to modify how the elementary multiplications are performed; and (2) the total communication in the protocol is proportional to the size of the *output* rather than the number of *intermediate* element-wise multiplications.

Most significant bit. Several of our protocols rely on a protocol for computing the most significant bit $[\![\mathrm{msb}(x)]\!]^n$ of a secret-shared value $[\![x]\!]^n$. In our fixed-point representation, this corresponds to computing the sign of x. For this, we adopt the general approach from ABY^3. Namely, given an arithmetic secret sharing $[\![x]\!]^n = (x_1, x_2, x_3)$ of x, the parties re-interpret it as three *binary* shares of values $[\![x_1]\!]^2 = (x_1, 0, 0)$, $[\![x_2]\!]^2 = (0, x_2, 0)$, and $[\![x_3]\!]^2 = (0, 0, x_3)$. The parties now evaluate an addition circuit on the binary shares $[\![x_1]\!]^2, [\![x_2]\!]^2, [\![x_3]\!]^2$ to compute binary shares of the sum $[\![x]\!]^2$, which in particular, yields a binary share of $[\![\mathrm{msb}(x)]\!]^2$. Finally, to recover arithmetic shares of $[\![\mathsf{ReLU}(x)]\!]^n$ from $[\![x]\!]^n$ and $[\![\mathrm{msb}(x)]\!]^2$, we use the bit injection protocol from ABY^3 [2, §5.4], which only requires simple arithmetic operations.

The majority of this computation is the evaluation of the addition circuit over binary shares on the GPU. Evaluating a Boolean addition circuit on secret-shared binary values decomposes into a sequence of bitwise AND and XOR operations (along with communication for the AND gates), which can be computed using efficient GPU kernels. We provide microbenchmarks in Section IV-C.

ReLU activation function. The standard activation function we consider in our networks is the rectified linear unit (ReLU) [40, 16]: $\mathsf{ReLU}(x) := \max(x, 0)$. To compute the ReLU function, it suffices to construct a protocol for testing whether the fixed-point value x is positive or not. This cor-

responds to computing the most significant bit $\mathrm{msb}(x)$ of x, which we evaluate using the protocol described above.

C. Additional Building Blocks for Private Training

To support private training, we need to augment our existing toolkit with several additional protocols. Here, we consider a standard backpropagation setting with a softmax/cross-entropy loss function optimized using (minibatch) stochastic gradient descent (SGD) [41]. As with private inference, we decompose the backpropagation algorithm into a sequence of elementary operations and build our private training protocol by sequentially composing protocols for the elementary operations.

In this work, we consider classification tasks with d target classes. Each iteration of SGD takes an input $\mathbf{x} \in \mathbb{R}^m$ and a one-hot encoding of the target vector $\mathbf{y} \in \{0,1\}^d$ (i.e., $y_i = 1$ if $\mathbf{x}$ belongs to class i and $y_i = 0$ otherwise) and computes the cross-entropy loss:[3]

$$\ell_{\mathsf{CE}}(\mathbf{x};\mathbf{y}) := -\sum_{i\in[d]} y_i \log \tilde{z}_i, \tag{III.1}$$

where $\tilde{\mathbf{z}} \leftarrow \mathrm{softmax}(\mathbf{z})$, $\mathbf{z} \leftarrow \mathsf{Eval}(M,\mathbf{x})$, and M is the current model. For a vector $\mathbf{x} \in \mathbb{R}^d$, the softmax function

$$\mathrm{softmax}_i(\mathbf{x}) := e^{x_i} / \sum_{i\in[d]} e^{x_i}. \tag{III.2}$$

The gradient of ℓ_{CE} for the output layer $\mathbf{z}$ is then

$$\nabla_{\mathbf{z}} \ell_{\mathsf{CE}} = \mathrm{softmax}(\mathbf{z}) - \mathbf{y}.$$

We can use the private inference protocol from Section III-B to compute $[\![\mathbf{z}]\!]^n$ from $[\![\mathbf{x}]\!]^n$ and $[\![M]\!]^n$. To compute $[\![\nabla_{\mathbf{z}} \ell_{\mathsf{CE}}]\!]^n$, we need a protocol to compute softmax on secret-shared values.

For the ReLU layers, the gradient computation reduces to evaluating the derivative of the ReLU function. The gradients for the linear/convolution layers are themselves linear functions of the gradients from the preceding layer, and thus, can be handled using the protocols from Section III-B. In the following, we describe our protocols for evaluating the softmax and the derivative of the ReLU function on secret-shared values. Note that backpropagation does *not* require computing the value of the loss function (Eq. (III.1)), so we do *not* need a protocol for computing logarithms on secret-shared values.

Softmax. To avoid numeric imprecision from evaluating the exponential function in the softmax function (Eq. (III.2)) on very large or very small inputs, a standard technique is to evaluate the softmax on the "normalized" vector $(\mathbf{x} - \max_i x_i)$ [41]. A simple calculation shows that $\mathrm{softmax}(\mathbf{x} - \max_i x_i) = \mathrm{softmax}(\mathbf{x})$. This has the advantage that all inputs to the exponential function in Eq. (III.2) are at most 0, and the denominator is contained in the interval $[1, d]$. In the following, we describe protocols for evaluating the exponential function, division, and computing the maximum over a vector of secret-shared values. Together, this yields a protocol for computing a softmax on secret-shared values.

Exponentiation. We approximate the exponential function e^x needed to compute softmax with its limit characterization f_m:

$$f_m(x) := \left(1 + \frac{x}{m}\right)^m. \tag{III.3}$$

Using a Taylor expansion for the function $\ln(1+x)$ and assuming that $|x| < m$,

$$\frac{f_m(x)}{e^x} = \frac{e^{m\ln(1+x/m)}}{e^x} = e^{-O(x^2/m)}.$$

Thus, the degree-m approximation f_m provides a good approximation e^x on an interval of size $O(\sqrt{m})$ centered at 0. A common alternative approximation is to use Taylor series to approximate the exponential function. The advantage of using a Taylor series approximation of degree m is that it provides a good estimate in an interval of size $O(m)$ (as opposed to $O(\sqrt{m})$ using the approximation f_m). However, using a Taylor series approximation has several drawbacks:

- Evaluating a degree-m Taylor approximation requires m multiplications over $O(\log m)$ rounds. Computing f_m, in comparison, only requires $\log m$ multiplications. For a fixed degree m, the cost of computing the f_m approximation is *exponentially* smaller than computing the degree-m Taylor series approximation.
- The size of the smallest coefficient in the Taylor series of degree m is $1/m!$. In a fixed-point encoding scheme with t bits of precision, values less than 2^{-t-1} round to 0. This gives an upper bound on the degree of the Taylor expansion we can feasibly support. Alternatively, we could compute the terms $x^m/m!$ in the Taylor expansion as $\prod_{i\in[m]} \frac{x}{i}$, but this now requires $O(m)$ *rounds* of multiplications to compute.
- In our setting, the inputs to the exponential function are drawn from the interval $(-\infty, 0]$. The approximation $f_m(x)$ has the appealing property that as $x \to -\infty$, $f(x) \to 0$, which matches the behavior of e^x. In contrast, the Taylor approximation *diverges* as $x \to -\infty$. This can introduce significant errors in the computation (unless we use a Taylor approximation of sufficiently high degree). For the models and inputs we consider in Section IV-A, most inputs to the exponential function lie in the interval $[-45, 0]$. Ensuring that the Taylor approximation does not diverge for all inputs in this interval would require a high-degree approximation.

Thus, compared to a Taylor approximation, the limit-based approximation f_m is more efficient to evaluate (in terms of the number of multiplications) and more robust for handling large negative inputs that may arise in the computation.

Division. Computing the softmax function requires computing a quotient $[\![x/y]\!]^n$ on secret-shared values $[\![x]\!]^n$ and $[\![y]\!]^n$ and where $1 \le y \le Y$, for some bound Y. It suffices to compute the reciprocal $[\![1/y]\!]^n$ and compute the quotient using share multiplication. Similar to previous works [6], we use

[3] Technically, in minibatch SGD, each iteration takes a batch of N inputs and the loss function is the average of the loss function for all N inputs in the batch. For ease of exposition, we describe the setup for a single input, but everything generalizes naturally to the minibatch setting.

the iterative Newton-Raphson algorithm to approximate the value of $1/y$. Very briefly, the Newton-Raphson algorithm for approximating $1/y$ starts with an initial "guess" z_0 and iteratively computes $z_i \leftarrow 2z_{i-1} - yz_{i-1}^2$. In this work, we use a *fixed* initialization $z_0 = 1/Y$. This provides a highly-accurate estimate for $1/y$ for all $y \in [1, Y]$ using $O(\log Y)$ iterations of Newton's algorithm. To see this, let

$$\text{error}_i = |1/y - z_i| = \frac{1}{y}|1 - z_i y| \leq \varepsilon_i,$$

where $\varepsilon_i = |1 - z_i y|$. Substituting in the Newton-Raphson updates, $\varepsilon_i = \varepsilon_{i-1}^2$, so the *maximum* error after i iterations is $(1 - 1/Y)^{2^i} \leq e^{-2^i/Y}$.

We note that using a more accurate initialization for Newton-Raphson will allow convergence in fewer iterations. However, methods for computing a more accurate estimate [6] for the initialization typically rely on binary-valued operations (e.g., comparisons) and are *more* costly than using a fixed initialization and increasing the number of iterations. Note that a fixed initialization is possible in our setting because we are guaranteed that the values y lies in a fixed interval (due to the normalization in the softmax computation).

Maximum. The last ingredient we require for computing the softmax function is computing the maximum value $[\![\max_i x_i]\!]^n$ from a secret-shared vector $[\![\mathbf{x}]\!]^n$ where $\mathbf{x} \in \mathbb{R}^m$. We implement this using m invocations of a comparison protocol. To reduce the round complexity to $\log m$, we use a tree of comparisons where pairs of elements are compared each round, and the larger value in each pair advances to the next round. Comparing two secret-shared *fixed-point* values $[\![x]\!]^n, [\![y]\!]^n$ is equivalent to computing the most significant bit of their difference (i.e., $[\![\text{msb}(x-y)]\!]^n$). We implement this using the protocol from Section III-B.

Derivative of ReLU. During backpropagation, we also need to compute the derivative of the ReLU function $\text{ReLU}'(x)$, which is 0 if $x < 0$ and 1 if $x > 0$. This again corresponds to computing the most significant bit of the fixed-point encoding of x, which we implement using the protocol from Section III-B.

IV. System Implementation and Evaluation

We build CrypTGPU on top of CrypTen, which itself builds on PyTorch. First, we introduce the `CUDALongTensor` data type that represents a PyTorch tensor for 64-bit integer values (see Section II-B). Our design enables us to take advantage of optimized CUDA kernels for evaluating bilinear operations such as convolutions and matrix multiplications on secret-shared tensors. This suffices for evaluating arithmetic circuits on secret-shared tensors. Using these elementary building blocks, we then implement protocols for each of the operations described in Section III (i.e., the truncation protocol for fixed-point multiplication, ReLU computation, and the softmax function). Through composing these individual protocols together, we obtain an end-to-end system for private inference and private training.

Point-to-point communication. We leverage PyTorch's `torch.distributed` package for point-to-point communication between parties. The default communication mode in PyTorch is a "broadcast" mode where every message sent by a party is sent to *all* peers. To emulate point-to-point channels (as required by our protocol), we initialize a separate communication back end between each pair of parties. In this case, a "broadcast" channel between each pair of parties functions as a point-to-point channel between the parties.

Pseudorandom generators on the GPU. We use AES as the PRF in our protocol (used for share re-randomization in the truncation protocol). We use the `torchcsprng` PyTorch C++/CUDA extension [42] (based on the Salmon et al. protocol [43]) which enables AES evaluation on the GPU.

A. Experimental Setup for System Evaluation

We now describe our experimental setup for evaluating CrypTGPU as well as the specific parameters we use to instantiate our cryptographic protocols from Section III.

Deep learning datasets. We evaluate CrypTGPU on the following standard datasets for object recognition:

- **MNIST** [11]. MNIST is a dataset for handwritten digit recognition. The training set has 60,000 images and the test set has 10,000 images. Each digit is a grayscale (i.e., single-channel) 28×28 image. Due to its relatively small size, it is widely used as a benchmark in many privacy-preserving ML systems [1, 2, 5, 6].
- **CIFAR-10** [12]. CIFAR-10 is a dataset with 60,000 32×32 RGB images split evenly across 10 classes.
- **Tiny ImageNet** [15]. Tiny ImageNet is a modified subset of the ImageNet dataset. It contains 100,000 64×64 RGB training images and 10,000 testing images split across 200 classes. Compared to CIFAR-10, Tiny ImageNet is much more challenging: each image is $4\times$ larger and there are $20\times$ more classes.
- **ImageNet** [13]. ImageNet is a large-scale visual recognition dataset with more than 1,000,000 training images. It is the standard benchmark for evaluating the classification performance of computer vision models. ImageNet has 1000 classes, and each example is a center-cropped 224×224 RGB image. The only prior system for privacy-preserving machine learning that demonstrates performance at the scale of ImageNet is CrypTFlow [5].

Deep learning models. For our experimental evaluation, we measure the cost of our private training and private inference protocols on several representative CNN architectures developed for object recognition. Each of these networks can be represented as a composition of a collection of standard layers: convolution, pooling, activation, batch normalization, softmax, and fully-connected layers.

- **LeNet** [44]. LeNet was proposed by LeCun et al. for handwritten digit recognition. It is a shallow network with 2 convolutional layers, 2 average pooling layers, and 2 fully

connected layers. The network uses the hyperbolic tangent ($\tanh$) as its activation function.

- **AlexNet** [16]. AlexNet was the winner of 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC-2012) competition. It has 5 convolutional layers, 3 max pooling layers, and 2 fully connected layers for a total of 61 million parameters. AlexNet uses ReLU as its activation function.
- **VGG-16** [17]. VGG-16 is the runner-up of the ILSVRC-2014 competition. It uses 16 layers consisting of convolution, ReLU, max pooling, and fully-connected layers. VGG-16 has a total of 138 million parameters.
- **ResNet** [14]. ResNet is the winner of ILSVRC-2015 competition. It introduces skip-connections that addresses the vanishing gradient problem when training deep neural network models. ResNet consists of convolution, max pooling, average pooling, batch normalization, and fully connected layers. Since their inception, the ResNet family of models have enjoyed wide adoption in the computer vision community. We evaluate the performance of ResNet-50, ResNet-101, and ResNet-152 on ImageNet. These networks respectively have 23, 44, and 60 million parameters and 50, 101, and 152 layers.

Architecture adjustments. We use the standard architecture of each of these networks, except with the following modifications:

- **AlexNet and VGG-16 on small datasets.** Since AlexNet and VGG-16 were designed for ImageNet, they are not directly compatible with smaller inputs (i.e., those from CIFAR-10 or Tiny ImageNet). Thus, when using AlexNet or VGG-16 with smaller inputs, we have to modify the network architecture. For AlexNet, we drop the final max pooling layer for CIFAR-10, and adjust the number of neurons in the fully-connected classification layers to 256-256-10 and 1024-1024-200 for CIFAR-10 and Tiny ImageNet, respectively. For VGG-16, we adjust the number of neurons in the fully-connected classification layers to 256-256-10 and 512-512-200 for CIFAR-10 and Tiny ImageNet, respectively.[4] When evaluating AlexNet on ImageNet, we use the original architecture [16]. In the case of VGG-16, we add a 2x2 average pooling layer to reduce the input dimension of the first fully connected layer from 18432 to 4608; this is due to memory limitations on the GPU. When we compare our system to the FALCON system on these models and datasets, we make the same adaptations. We provide the full specification of the AlexNet and VGG-16 model architectures we use in Appendix A.
- **Activation functions.** All networks we consider except LeNet use the ReLU function as the activation function. In contrast, LeNet uses the hyperbolic tangent function $\tanh$ as the underlying activation function. Since CRYPTGPU does not support evaluating the $\tanh$ function and modern networks primarily use ReLU as their activation function, we replace $\tanh$ with ReLU in our experiments with LeNet.
- **Average pooling.** Pooling is a standard way to downsample the outputs of the convolutional layers in a CNN. Specifically, a pooling layer accumulates the output of the convolutional layers by replacing each (small) window of the feature map (from the convolutional layer) with the average of the values (i.e., average pooling) or the max of the values (i.e., max pooling). Earlier networks such as AlexNet and VGG-16 used max pooling throughout, while more recent deep networks such as the ResNets primarily use average pooling (with a *single* max pooling layer at the beginning). While the choice of pooling function does not make a significant difference in the computational costs of plaintext training, this is not the case in private training. The difference is due to the fact that average pooling is a *linear* operation while max pooling is a highly *non-linear* operation. To reduce the computational overhead of our system, we replace all the max pooling layers in the above networks with average pooling. This reduces the complexity at the cryptographic level and allows us to take better advantage of GPU parallelism.

  We show in Section IV-B that in existing systems, the pooling layer is *not* the bottleneck, and the performance improvements of our protocol relative to past works is *not* due to our substitution of average pooling in place of max pooling. We additionally show in Section IV-D that using average pooling in place of max pooling does not significantly affect the accuracy of the models we consider.

Protocol instantiation. We instantiate our protocols from Section III using the following parameter settings:

- **Fixed-point precision**. We consider secret-sharing schemes over the 64-bit ring $\mathbb{Z}_{2^{64}}$, and encode inputs using a fixed-point representation with $t = 20$ bits of fractional precision (i.e., an input $x \in \mathbb{R}$ is encoded as $\lfloor x \cdot 2^{20} \rceil$. In Section IV-D, we analyze the effect the number of bits of precision has on the accuracy of our protocols.
- **Exponentiation.** We use the function f_m from Eq. (III.3) to approximate the exponential function. In this work, we take $m = 2^9 = 512$, so evaluating f_m requires $\log m = 9$ rounds of multiplication. With $t = 20$ bits of fixed-point precision, we measure the maximum error of our approximation on all inputs $x \leq 0$ to be at most $6 \cdot 10^{-4}$.
- **Division.** As described in Section III-C, we require a private division protocol to compute $[\![1/y]\!]^n$ where $y \in [1, Y]$, and Y is the number of classes in the classification problem. For all of the datasets we consider for private training, $Y \leq 200$. In our implementation, we use 13 iterations of Newton-Raphson (with $1/Y$ as the initialization). With $t = 20$ bits of fixed-point precision, we measure the maximum absolute difference between the approximate value and the true value for inputs in the interval $[1, Y]$ to be $\approx 10^{-4}$ (and $\approx 10^{-9}$ using floating-point evaluation).

B. Benchmarks for Private Training and Inference

We run our experiments on three Amazon EC2 instances optimized for *GPU computation* (`p3.2xlarge`). Each instance

[4] Previous systems like FALCON [6] made similar adjustments when evaluating AlexNet and VGG-16 on smaller datasets.

has a single NVIDIA Tesla V100 GPU with 16 GB of GPU memory. All of the instances run Ubuntu 18.4 and have 8 Intel Xeon E5-2686 v4 (2.3 GHz) CPUs and 61 GB of RAM. We consider a local area network (LAN) environment and place all three servers in the `us-east-1` (Northern Virginia) region. In this case, we measure the network bandwidth to be 1.25GB/s with an average latency of 0.2ms. For each model/dataset pair we consider in our evaluation, we measure the end-to-end protocol execution time and the total amount of communication.

Comparisons with prior work. We compare the performance of CRYPTGPU against FALCON [6] and CRYPTFLOW [5]. To our knowledge, these are the only privacy-preserving machine-learning frameworks that have demonstrated the ability to handle neural networks at the scale of AlexNet on large datasets. Since our primary focus is on the scalability of our approach and not on the performance on shallow networks (where GPUs are unlikely to shine compared to optimized CPU protocols), we focus our comparisons with FALCON and CRYPTFLOW.

- For CRYPTFLOW (which supports private inference for ResNet), we use the performance numbers reported in their paper (which also operate in a LAN environment).
- For FALCON (which supports private inference and private training for LeNet, AlexNet, and VGG-16), we collect benchmarks using their provided reference implementation [45]. We run the FALCON system on three *compute-optimized* AWS instances (`c4.8xlarge`) in the Northern Virginia region.[5] Each instance runs Ubuntu 18.4 and has 36 Xeon E5-2666 v3 (2.9 GHz) CPUs and 60 GB of RAM. We measure the network bandwidth between machines to be 1.16GB/s with an average latency of 0.2ms.

For the main benchmarks, we also measure the computational cost using PyTorch on *plaintext* data (with GPU acceleration).

Private inference. Table I summarizes the performance of CRYPTGPU's private inference protocol on the models and datasets described in Section IV-A. For shallow networks and small datasets (e.g., LeNet on MNIST or AlexNet on CIFAR), FALCON outperforms CRYPTGPU. However, as we scale to progressively larger datasets and deeper models (e.g., VGG-16 on Tiny ImageNet), then CRYPTGPU is faster (3.7× on VGG-16). The performance on small datasets is not unexpected; after all, if the computation is sufficiently simple, then the extra parallelism provided by the GPU is unlikely to benefit. Moreover, the use of more efficient cryptographic building blocks (which may not be "GPU-friendly") can allow a CPU-based approach to enjoy superior performance.

The setting where we would expect the GPU-based approach to perform well is in the setting of large datasets and deeper models. For instance, at the scale of ImageNet, CRYPTGPU is able to perform private inference over the ResNet-152 network (containing over 150 layers and over 60 million parameters) in just over 25 seconds. This is about 2.2× faster than CRYPTFLOW, which to our knowledge, is the only protocol for private inference that has demonstrated support for the ResNet family of networks on the ImageNet dataset. For the ResNet family of networks, the running time of CRYPTGPU scales linearly with the depth of the network.

Compared to plaintext inference on the GPU, there still remains a significant 1000× gap in performance. This underscores the importance of designing more GPU-friendly cryptographic primitives to bridge this gap in performance.

Batch private inference. We can also leverage GPU parallelism to process a *batch* of images. This allows us to amortize the cost of private inference. Table II shows the time and communication needed for private inference over a batch of 64 images on the CIFAR-10 dataset. Here, the amortized cost of private inference on a single image using AlexNet drops from 0.91s to 0.017s (a 53× reduction). With VGG-16, batch processing reduces the per-image cost from 2.14s to 0.18s (a 12× reduction).

Table III shows the time and communication needed for private inference on ImageNet using the ResNet networks with a batch of 8 images. Here, we see a 1.9× reduction in the amortized per-image private inference cost for each of ResNet-50, ResNet-101, and ResNet-152. The cost reduction compared to those on the CIFAR-10 dataset (Table II) is smaller. This is likely due to the smaller batch sizes in play here (8 vs. 64). Supporting larger batch sizes is possible by either using multiple GPUs or using GPUs with more available memory. Nonetheless, irrespective of the model/input size, we observe that batch private inference allows us to amortize the cost of private inference protocol. Communication in all cases scales linearly with the batch size.

Private training. We expect GPUs to have a larger advantage in the setting of private training (just like modern deep learning, training is much more challenging than inference and thus, more reliant on hardware acceleration). We measure the time needed for a *single iteration* of private backpropagation (Section III-C) on a batch size of 128 images for several dataset/model configurations and summarize our results in Table IV (together with measurements for the equivalent plaintext protocol). We only compare with FALCON because CRYPTFLOW does not support private training. We note that the public implementation of the FALCON system [45] does *not* include support for computing the cross-entropy loss function for backpropagation. However, given the gradients for the output layer, the provided implementation supports gradient computation for *intermediate* layers. Thus, our measurements for the FALCON system only includes the cost of computing the gradients for intermediate layers and not for the output layer; this provides a *lower bound* on the running time of

[5]Note that we use *different* instances for our comparison because CRYPTGPU is *GPU-based* while FALCON is *CPU-based*.

	LeNet (MNIST)		AlexNet (CIFAR)		VGG-16 (CIFAR)		AlexNet (TI)		VGG-16 (TI)	
	Time	Comm. (MB)	Time	Comm. (MB)	Time	Comm. (MB)	Time	Comm. (MB)	Time	Comm. (MB)
FALCON	0.038	2.29	0.11	4.02	1.44	40.45	0.34	16.23	8.61	161.71
CRYPTGPU	0.38	3.00	0.91	2.43	2.14	56.2	0.95	13.97	2.30	224.5
Plaintext	0.0007	—	0.0012	—	0.0024	—	0.0012	—	0.0024	—
	AlexNet (ImageNet)		**VGG (ImageNet)**		**ResNet-50 (ImageNet)**		**ResNet-101 (ImageNet)**		**ResNet-152 (ImageNet)**	
	Time	Comm. (GB)	Time	Comm. (GB)	Time	Comm. (GB)	Time	Comm. (GB)	Time	Comm. (GB)
CRYPTFLOW	—	—	—	—	25.9	6.9	40*	10.5*	60*	14.5*
CRYPTGPU	1.52	0.24	9.44	2.75	9.31	3.08	17.62	4.64	25.77	6.56
Plaintext	0.0013	—	0.0024	—	0.011	—	0.021	—	0.031	—

*Value estimated from [5, Fig. 10]

TABLE I: Running time (in seconds) and total communication of private inference for different models, datasets, and systems in a LAN setting. The "TI" dataset refers to the Tiny ImageNet dataset [15]. The plaintext measurements correspond to the cost of inference on plaintext data on the GPU (using PyTorch). Performance numbers for CRYPTFLOW are taken from [5]. Performance numbers for FALCON are obtained by running its reference implementation [45] on three compute-optimized AWS instances in a LAN environment (see Section IV-B). As discussed in Section IV-A, when testing the performance of CRYPTGPU, we replace max pooling with average pooling in all of the networks.

	$k = 1$		$k = 64$	
	Time	Comm.	Time	Comm.
AlexNet	0.91	0.002	1.09	0.16
VGG-16	2.14	0.056	11.76	3.60

TABLE II: Running time (in seconds) and total communication (in GB) for batch private inference on CIFAR-10 using a batch size of k.

	$k = 1$		$k = 8$	
	Time	Comm.	Time	Comm.
ResNet-50	9.31	3.08	42.99	24.7
ResNet-101	17.62	4.64	72.99	37.2
ResNet-152	25.77	6.56	105.20	52.5

TABLE III: Running time (in seconds) and total communication (in GB) for batch private inference on ImageNet using a batch size of k.

using FALCON for private training. Our system supports the full backpropagation training algorithm.

Our system achieves a considerable speedup over FALCON in multiple settings, especially over larger models and datasets. For instance, to train AlexNet on Tiny ImageNet, a single iteration of (private) backpropagation completes in 11.30s with CRYPTGPU and 6.9 minutes using FALCON. For context, privately training AlexNet on Tiny ImageNet (100,000 examples) would just take over a week ($\approx$ 10 days) using CRYPTGPU while it would take over a year ($\approx$ 375 days) using FALCON (assuming 100 epochs over the training set).

On the larger VGG-16 network, our system is constrained by the amount of available GPU memory. Our system currently supports a maximum batch size of 32 when training VGG-16 on CIFAR-10 and a maximum batch size of 8 when training on Tiny ImageNet. To establish a fair comparison when comparing our system against FALCON for privately training VGG-16, we apply the same batch size adjustment. As shown in Table IV, when training VGG-16, our system is $30\times$ faster when training on CIFAR-10 and $26\times$ when training on Tiny ImageNet. Reducing the memory overhead of our protocol and augmenting it with support for *multiple* GPUs (as is standard for modern deep learning) will enable better scalability. We leave this as an interesting direction for future work.

Like the setting of private inference, there still remains a large gap (roughly $2000\times$) between the costs of private training and plaintext training (on the GPU). Designing new cryptographic protocols that can take even better advantage of GPU parallelism will be important for closing this gap.

Private training breakdown. In Table V, we provide a fine-grained breakdown of the costs of processing the different layers in a single iteration of private training. Not surprisingly, the primary advantage of our GPU-based protocol compared to the CPU-based protocol of FALCON is in the computation of the linear layers. In the settings we consider, evaluation of the linear layers is between $25\times$ and $70\times$ faster with our system. The linear layers are the *primary* bottleneck in FALCON, and account for 86% to 99% of the overall computational cost. In CRYPTGPU, the computational costs are more evenly split between the linear layers and the non-linear layers.

For the pooling layers, the performance difference between CRYPTGPU and FALCON can be partially attributed to the the fact that FALCON uses *max pooling* rather than *average pooling*. As discussed in Section IV-A, average pooling is a linear function and simpler to evaluate privately. However, our measurements show that CRYPTGPU maintains a (significant) performance edge even if we exclude the cost of the pooling layers from the running time of FALCON.

Finally, for the ReLU layers, the CPU-based protocol in FALCON compares very favorably with the ReLU protocol in CRYPTGPU, and even outperforms our protocol on the

	LeNet (MNIST)		AlexNet (CIFAR-10)		VGG-16 (CIFAR-10)		AlexNet (TI)		VGG-16 (TI)	
	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.	Time	Comm.
FALCON*	14.90	0.346	62.37	0.621	360.83†	1.78†	415.67	2.35	359.60‡	1.78‡
CRYPTGPU	2.21	1.14	2.91	1.37	12.14†	7.55†	11.30	6.98	13.89‡	7.59‡
Plaintext	0.0025	—	0.0049	—	0.0089	—	0.0099	—	0.0086	—

*The provided implementation of FALCON does not support computing the gradients for the output layer, so the FALCON measurements only include the time for computing the gradients for intermediate layers. All measurements for FALCON are taken *without* batch normalization.
†Using a smaller batch size of 32 images per iteration (due to GPU memory limitations). We make the same batch size adjustment for FALCON.
‡Using a smaller batch size of 8 images per iteration (due to GPU memory limitations). We make the same batch size adjustment for FALCON.

TABLE IV: Running time (in seconds) and total communication (in GB) for a single iteration of private training with a batch size of 128 images for different models, datasets, and systems in a LAN setting. The "TI" dataset refers to the Tiny ImageNet dataset [15]. The plaintext measurements correspond to the cost of training on plaintext data on the GPU. Performance numbers for FALCON are obtained by running its reference implementation [45] on three compute-optimized AWS instances in a LAN environment (see Section IV-B). As discussed in Section IV-A, when testing the performance of CRYPTGPU, we replace max pooling with average pooling in all of the networks.

smaller models and datasets. Having a ReLU protocol that can better take advantage of GPU parallelism will likely improve the performance of our protocol. As described in Section III-B, our ReLU protocol relies on an arithmetic-to-binary share conversion, which is less GPU-friendly compared to bilinear operations. The ReLU protocol from FALCON relies on different techniques and it is interesting whether their approach can be adapted to be efficiently computed on the GPU.

Avenues for improvement. Compared to FALCON, our private training protocol is more communication-intensive. FALCON develops a number of specialized cryptographic protocols to substantially reduce the communication in their protocols. We believe it is an interesting question to study whether the protocols developed in FALCON are "GPU-friendly" and can benefit from GPU acceleration.

CRYPTGPU does not currently support batch normalization during private training, so we do not report private training benchmarks on the ResNet-family of models.[6] Developing a GPU-friendly protocol for batch normalization is an interesting avenue for further work and an important step towards supporting private training of the ResNet family of models. We are not aware of any system that currently supports private training over ResNet.

C. Microbenchmarks

To quantify the advantage of keeping all of the computation on the GPU, we compare the running time of the MPC protocols for evaluating convolutions (i.e., the linear layers) and for evaluating ReLU (i.e., the primary non-linear layer) on the CPU vs. the GPU. For convolutions, we study the effect of both the input dimension as well as the batch size. We use the same experimental setup described in Section IV-A for all of the experiments in this section.

Private convolution: GPU vs. CPU. For convolutions, we consider two types of convolutions: (1) convolutions with a large receptive field (filter size) but a relatively small number of input/output channels; and (2) convolutions with a small receptive field, but a large number of input/output channels. Convolutions of the first type are generally used in the initial layers of the CNN while filters of the second type are used in the later layers of the CNN. Note that when implementing convolutions on the CPU, we do *not* break up the 64-bit secret-shared tensor into 16-bit blocks (as we do in the GPU setting; see Section II-B). We provide the microbenchmarks in Fig. 1.

From Figs. 1a and 1c, we see that for small inputs, the computational cost of the private convolution protocol is comparable on both the GPU and the GPU. While there is only a $10\times$ speed-up for convolutions between a small $32 \times 32 \times 3$ input with a stack of 64 filters, the gap grows quickly as the input size increases; for instance, increasing the input size to that of a Tiny ImageNet instance $(64 \times 64 \times 3)$, the GPU-based protocol is nearly $40\times$ faster. Scaling to a $512 \times 512 \times 3$ image, the GPU-based protocol is $174\times$ faster than the CPU-based protocol (from 23.9s on the CPU to 0.14s on the GPU). An analogous trend holds when we consider convolutions with a large number of input/output channels: for small inputs, the running times of the CPU- and GPU-based protocols are quite comparable, but for large inputs (e.g., a $64 \times 64 \times 512$ input), the GPU-based protocol is $168\times$ faster (from 543s on the CPU to just 3.2s on the GPU).

We additionally note that for small instances, the protocol running time on the GPU is essentially constant—this is due to the parallelism. Only after the input becomes sufficiently large do we start seeing increases in the running time based on the size of the input. In contrast, the CPU running time always scales with the size of the input.

Similar speedups are present when we consider convolutions on batches of inputs (this is important for training and for

[6]Note that we can still perform private inference for a model that is trained using batch normalization. Namely, the normalization parameters are secret-shared (as part of the model) and applying batch normalization just corresponds to an affine transformation.

	Linear		Pooling		ReLU		Softmax	
	FALCON	CRYPTGPU	FALCON	CRYPTGPU	FALCON	CRYPTGPU	FALCON	CRYPTGPU
LeNet (MNIST)	13.07	0.49	1.34	0.076	0.47	1.00	—	0.53
AlexNet (CIFAR)	59.23	0.86	2.65	0.077	0.41	1.33	—	0.55
VGG-16 (CIFAR)*	355.16	6.33	2.86	0.21	5.40	4.74	—	0.53
AlexNet (TI)	402.45	5.60	10.20	0.37	1.92	4.16	—	1.04
VGG-16 (TI)†	355.84	7.61	2.87	0.32	5.37	4.73	—	0.98

*Using a smaller batch size 32 images per iteration (due to GPU memory limitations). We make the same batch size adjustment for FALCON.
†Using a smaller batch size 8 images per iteration (due to GPU memory limitations). We make the same batch size adjustment for FALCON.

TABLE V: Runtime (in seconds) of FALCON [6] and CRYPTGPU for evaluating the linear, pooling, ReLU, and softmax layers for different models and datasets during private *training*. The "linear" layers include the convolution and the fully-connected layers. The "pooling" layer refers to max pooling in FALCON, and average pooling in CRYPTGPU. The implementation of FALCON [45] does not currently support softmax evaluation (and correspondingly, gradient computation for the output layer). Performance numbers for FALCON are obtained by running its reference implementation [45] on three compute-optimized AWS instances in a LAN environment (see Section IV-B).

batch inference). For a fixed input size ($32 \times 32 \times 3$) and kernel size (11×11), we observe a $10\times$ speed-up for running the private convolution protocol on a single input using the GPU, but a $40\times$ to $60\times$ speed-up when we consider a batch of anywhere from 32 to 512 inputs. As an example, to evaluate a convolution over a batch of 512 inputs with this set of parameters, we require 11.6s on the CPU and only 0.27s on the GPU. We refer to Fig. 1b for the full comparison.

Private ReLU: GPU vs. CPU. Previous privacy-preserving ML systems like DELPHI [4] leveraged GPUs to accelerate convolutions, but still executed the non-linear steps (e.g., ReLU computations) on the CPU. Here, we argue that with a carefully-chosen set of cryptographic protocols, we can also take advantage of GPU parallelism to accelerate the non-linear computations. To illustrate this, we compare the running time of our private ReLU protocol on the CPU vs. the GPU. As described in Section III-B, private ReLU evaluation of ReLU on a large block of neurons (e.g., output by the convolutional layer) corresponds to evaluating a large number of point-wise Boolean operations on secret-shared binary tensors. Such operations naturally benefit from GPU parallelism.

We measure the time it takes to privately-evaluate ReLU on different numbers of secret-shared inputs (ranging from 50,000 to 32,000,000). The full results are shown in Fig. 2. For ReLU evaluation, we see a $16\times$ speedup when evaluating ReLU on a block of 256,000 inputs (from 2s on the CPU to 0.12s on the GPU). As we scale up to a block with 32 million inputs (250 MB of data), there is a $9\times$ speedup on the GPU, with the absolute running time dropping from 149s on the CPU to just 16.3s on the GPU.

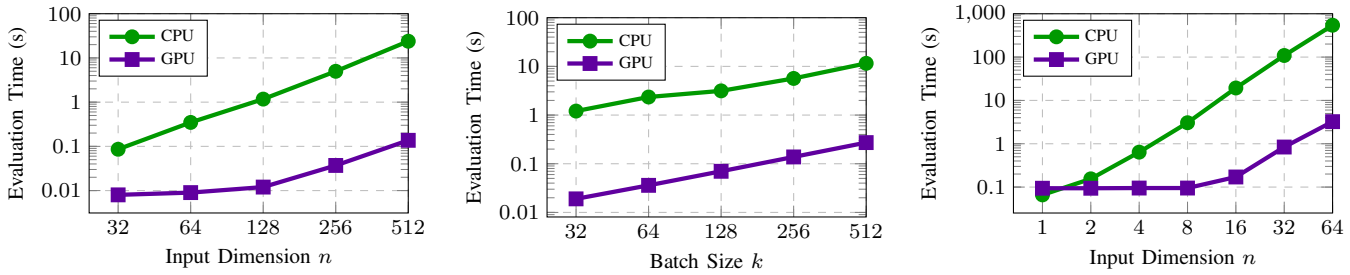
D. Accuracy of Privacy-Preserving Protocols

Several of the underlying protocols in CRYPTGPU are not exact and can introduce a small amount of error: using fixed-point encodings to approximate floating-point arithmetic, the share-truncation protocol from ABY^3, and the approximation to the softmax function. While we have chosen our parameters (e.g., the fixed-point precision) to reduce the likelihood of errors, we validate our parameter choices with an empirical analysis. In the following, we will often measure the difference between an output z_{priv} computed using CRYPTGPU with the output z_{plain} of a plaintext version of the same computation (using 64-bit floating-point values). We define the *absolute error* between z_{priv} and z_{plain} as $|z_{\text{priv}} - z_{\text{plain}}|$ and the *relative error* to be $|z_{\text{priv}} - z_{\text{plain}}|/z_{\text{plain}}$.

Fixed point precision. As discussed at the beginning of Section IV, CRYPTGPU emulates floating-point computations by encoding values using a fixed-point representation using $t = 20$ bits of fractional precision. The fixed-point computations over the integers are embedded into operations on secret-shared values over the ring $\mathbb{Z}_n$. The modulus n must be large enough to support multiplication (and more generally, convolution and matrix multiplication) of plaintext values without triggering a modular reduction. In CRYPTGPU, $n = 2^{64}$, so shares are represented by 64-bit integers.

Previous privacy-preserving protocols like FALCON [6] and DELPHI [4] use a smaller number of bits of fixed-point precision (e.g., 13 bits and 15 bits, respectively). In turn, they are able to work with arithmetic shares over a 32-bit ring as opposed to a 64-bit ring. This reduces communication (since shares are half as large) and in our model, also saves computation (recall from Section II-B that we need to split up tensors of 64-bit integers into 4 tensors of 16-bit integers in order to use existing CUDA kernels for deep learning).

Using fewer number of bits of precision reduces the accuracy of the protocol outputs, especially when scaling to deep architectures and large inputs. To analyze the effect the number of bits of fixed-point precision t has on the accuracy of the outputs of our system (i.e., the values of the output layer), we compute the average relative error between the output values output by CRYPTGPU to those computed using the plaintext inference protocol on a small example (AlexNet over CIFAR-10) as well as a large example (ResNet-50 on ImageNet). Our results are summarized in Fig. 3.



(a) Convolution on an $n \times n \times 3$ input with an 11×11 kernel, 64 output channels, 4×4 stride, and 2×2 padding.

(b) Convolution on batch of k $32 \times 32 \times 3$ inputs with an 11×11 kernel, 64 output channels, 4×4 stride, and 2×2 padding.

(c) Convolution on an $n \times n \times 512$ input with a 3×3 kernel, 512 output channels, 1×1 stride, and 1×1 padding.

Fig. 1: Comparison of total protocol execution time (in a LAN setting) for privately evaluating convolutions on the CPU and the GPU. Parameters for convolution kernels are chosen based on parameters in AlexNet [16]. The stride and padding parameters specify how the filter is applied to the input. All of the figures are log-log plots.

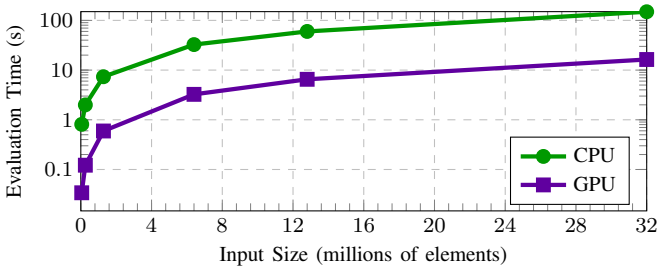


Fig. 2: Comparison of total protocol execution time (in a LAN setting) on the CPU vs. the GPU for point-wise evaluation of the private ReLU protocol on different-sized inputs.

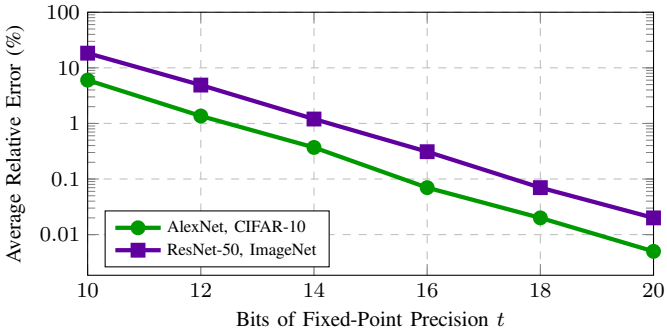
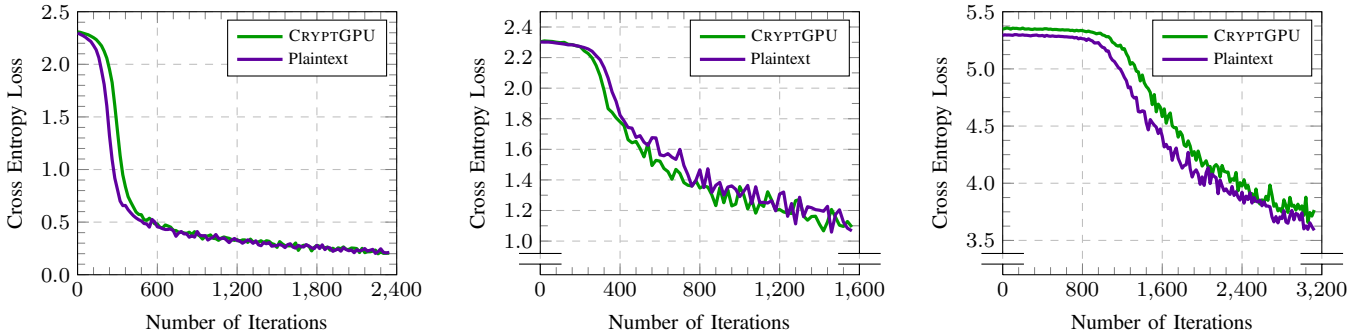


Fig. 3: Average relative error between the model outputs computed using the private inference protocol in CrypTGPU with t bits of fixed-point precision (i.e., an integer $x \in \mathbb{R}$ is represented as the nearest integer to $x \cdot 2^t$) and the output computed using *plaintext* floating-point inference. Analysis based on evaluating AlexNet on CIFAR-10 and ResNet-50 on ImageNet, and averaged over 10 randomly-chosen instances.

Fig. 3 shows that for a relatively shallow model like AlexNet on the CIFAR-10 dataset, it is sufficient to use 12 to 14 bits of fixed-point precision (e.g., the parameter setting in [6]). The relative error in this case between the outputs computed by the private inference protocol and the plaintext computation is around 1%. However, when we scale up to a model like ResNet-50 on ImageNet, the average relative error in the model outputs increases $5\times$ to almost 5%. We further remark that we are only measuring the relative error in a single forward pass over the network (inference). Larger errors would be expected in the case of private training when the protocol needs to run multiple forward and backward passes. In this work, we use $t = 20$ bits of fixed-point precision which ensures that the average relative error for private inference over ResNet-50 on ImageNet is under 0.02%. Our analysis indicates that scaling up to deeper architectures and operating over larger datasets will require a greater number of bits of precision in the underlying fixed-point representation. For instance, to keep the average relative error under 1% for ResNet-50 on ImageNet, we require at least 15 bits of fixed-point precision. As such, to prevent overflows in the arithmetic evaluation over secret-shared data for deep networks, a 32-bit ring is no longer sufficient.

Privacy-preserving inference. To evaluate the accuracy of our private inference protocol, we compare the average relative error between the outputs of our private inference protocol using ResNet-50, ResNet-101, and ResNet-152 on ImageNet and compare those against the values obtained from plaintext evaluation. We additionally compute the accuracy of the predictions (using the standard metrics of Top-1 and Top-5 accuracy—i.e., the model succeeds if the actual class of an example coincides with the most likely class predicted by the model or among the top 5 most likely classes predicted by the model). The results are summarized in Table VI. In particular, for our chosen set of parameters, we observe that the average relative error in the classifier output is at most 0.021%, and in all cases we tested (100 randomly-chosen images from the ImageNet test set), both the Top-1 accuracy and the Top-5 accuracy *exactly* match that of the plaintext model.



(a) LeNet on MNIST (trained for 5 epochs with a batch size of 128).

(b) AlexNet on CIFAR-10 (trained for 1 epoch with a batch size of 32).

(c) AlexNet on Tiny ImageNet (trained for 1 epoch on a batch size of 32).

Fig. 4: Moving average of the cross-entropy loss as a function of the number of training iterations using CrypTGPU and using a plaintext protocol for different models and datasets. In each setting, we use the same initialization and learning rate (for stochastic gradient descent) for both private and plaintext training. For LeNet, we use a random initialization. For the AlexNet experiments, we use PyTorch's default AlexNet architecture [46] for both the plaintext training and the private training experiments (which is a variant of the standard AlexNet architecture described in [16]). We use PyTorch's pre-trained initialization for AlexNet as our initialization. The moving average is computed over a window of size 20 (i.e., the value reported for iteration i is the average of the cross entropy loss on iterations $i-10,\ldots,i+9$).

	ResNet-50	ResNet-101	ResNet-152
Average Relative Error	0.015%	0.020%	0.021%
Top-1 Acc. (CrypTGPU)	78%	82%	79%
Top-1 Acc. (Plaintext)	78%	82%	79%
Top-5 Acc. (CrypTGPU)	92%	90%	93%
Top-5 Acc. (Plaintext)	92%	90%	93%

TABLE VI: Comparison of outputs of CrypTGPU's private inference protocol on ImageNet with the ResNet models with those of the plaintext algorithm (using PyTorch). The average relative error is computed between the outputs of the private inference protocol and those of the plaintext execution (on the same input). The Top-1 and Top-5 accuracies for both settings are computed based on the outputs of model inference with respect to the ground truth label. The measurements are taken over a random set of 100 examples drawn from the ImageNet validation set.

	Baseline	CrypTGPU	Plaintext
LeNet, MNIST*	10%	93.97%	93.34%
AlexNet, CIFAR-10†	10%	59.60%	59.77%
AlexNet, Tiny ImageNet‡	2%	17.82%	17.51%

*Trained for 5 epochs (2345 iterations) with a batch size of 128.
†Trained for 1 epoch (1563 iterations) with a batch size of 32.
‡Trained for 1 epoch (3125 iterations) with a batch size of 32.

TABLE VII: Validation set accuracy for different models trained using CrypTGPU and the plaintext training algorithm. For each configuration, both training approaches use the same initialization and learning rate (for stochastic gradient descent). For LeNet, we use a random initialization. For the AlexNet experiments, we use PyTorch's default AlexNet architecture [46] for both the plaintext training and the private training experiments. Here, we use PyTorch's pre-trained weights for AlexNet to speed up convergence. We also report the baseline accuracy for each configuration (i.e., accuracy of the "random-guess" algorithm). Note that training for more iterations will increase the accuracy; the intent of this comparison is to demonstrate a close similarity in model accuracies for the the model output by the private training protocol with the model output by plaintext training after a few thousand iterations of stochastic gradient descent.

Privacy-preserving training. We perform a similar set of experiments to evaluate the accuracy of our private training protocol. In Fig. 4, we plot the value of the cross-entropy loss function for a model trained using the private training protocol of CrypTGPU as well as for a model trained using the plaintext training algorithm (using the same initialization and learning rate for the underlying stochastic gradient descent optimizer). Fig. 4 shows that the value of the loss function is slightly higher initially for private training, but the overall progression closely follows that of plaintext training.

In addition to comparing the evolution of the loss function, we also compare the model accuracies (as measured on the validation set) for the models trained using CrypTGPU and using the plaintext training algorithm (again with same initialization and learning rate as above). Our results are summarized in Table VII. On all of the models/datasets we considered, the accuracy of the model output by CrypTGPU closely matches that of the plaintext evaluation. These experiments indicate that CrypTGPU efficiently and accurately supports *end-to-end* private training for models like AlexNet over moderately-large datasets like Tiny ImageNet.

Average pooling vs. max pooling. As discussed in Sec-

	Max Pooling	Average Pooling
AlexNet	76.15%	73.35%
VGG-16	82.37%	83.17%

TABLE VIII: Validation set accuracy for plaintext training of AlexNet and VGG-16 over the CIFAR-10 dataset using max pooling vs. average pooling. All networks were trained using 50 epochs using a standard stochastic gradient descent (SGD) optimizer in PyTorch.

tion IV-A, we use average pooling in place of max pooling in the models we consider. To evaluate whether the choice of pooling makes a significant difference on model performance, we use PyTorch to train the AlexNet and VGG-16 networks over the CIFAR-10 dataset where we replace all of the max pooling layers with average pooling layers. The resulting model accuracy on the CIFAR-10 test set is shown in Table VIII. In particular, we observed a 3% drop in accuracy (from 76% to 73%) for AlexNet and a 1% increase in accuracy with VGG-16 (from 82% to 83%). This indicates that using average pooling in place of max pooling does not lead to a significant degradation of model performance. We note also that in contrast to AlexNet and VGG-16 which use max pooling exclusively, the more recent ResNets use average pooling in all but the initial layer.

V. Related Work

Privacy-preserving machine learning is a special case of secure computation and can be solved via general cryptographic approaches such as secure 2-party computation (2PC) [28], secure multiparty computation [7, 8] or fully homomorphic encryption [47]. While powerful, these general approaches incur significant overhead, and much of the work in developing concretely-efficient protocols for scalable privacy-preserving machine learning have focused on more specialized approaches (that still rely on the general building blocks for designing sub-protocols). We survey some of these techniques here.

Privacy-preserving inference. Many recent works have developed specific protocols for the problem of private inference for deep learning models (c.f., [48, 1, 49, 50, 2, 51, 52, 3, 29, 53, 4, 5, 54, 30, 55, 56, 24, 57, 6] and the references therein). These works operate in a variety of different models and architectures: some works consider a 2-party setting (e.g., [1, 51, 52, 4]), others consider a 3-party (e.g., [2, 3, 6, 5, 29, 30]) or a 4-party setting (e.g., [55, 56]). Some frameworks assume that the model is held in the clear (e.g., [52, 4]) while others (including this work) support secret-shared models (e.g., [6, 5]). With the recent exceptions of Falcon [6] and CrypTFlow [5], these existing approaches only consider privacy-preserving inference using shallow neural networks (e.g., less than 10 layers) on relatively small datasets (at the scale of MNIST [11] or CIFAR [12]). Our focus in this work is designing privacy-preserving machine learning protocols that are able to support inference over modern deep learning models (which typically contain *tens of millions of parameters* and over a hundred layers) on large datasets (i.e., at the scale of ImageNet [13], one of the de facto standards for state-of-the art computer vision). As shown in Section IV-B, our system outperforms both Falcon and CrypTFlow for inference over sufficiently-large models and datasets.

Privacy-preserving training. Compared to private inference, privacy-preserving training of deep neural networks is a considerably more challenging and computationally-intensive problem and has received comparably less attention. Of the aforementioned works, only a few [1, 2, 29, 3, 55, 30, 56, 6] support privacy-preserving training. Among these systems, the only one that scales beyond MNIST/CIFAR is Falcon [6], which is the first system (to our knowledge) that supports privacy-preserving training at the scale of (Tiny) ImageNet and for models as large as AlexNet [16] and VGG-16 [17]. Our work is the first framework to leverage GPUs to demonstrate significantly better scalability to privately train deep networks over large datasets.

Privacy-preserving machine learning using GPUs. Most of the works on privacy-preserving machine learning are CPU-based and do not leverage GPU acceleration. We discuss some notable exceptions. Some works [58, 57] use GPUs to accelerate homomorphic evaluation of convolutional neural networks on MNIST. Delphi [4] uses GPUs to compute linear layers (i.e., convolutions) to support private inference; however, they still perform non-linear operations (e.g., ReLU evaluation) on the CPU and moreover, their scheme assumes the model to be public (and only the input is hidden). Our design philosophy in this work is to keep *all* of the computations on the GPU through a careful choice of "GPU-friendly" cryptographic protocols. Slalom [36] shows how to integrate a trusted computing base (e.g., Intel SGX) with GPUs to enable fast private inference of neural networks (by offloading convolutions to the GPU and performing non-linear operations within the trusted enclave). Recent works proposing scalable private training and inference protocols highlight the use of GPUs as an important way for further scalability [6, 5]. Our system is the first to support private training and inference *entirely* on the GPU.

Model stealing and inversion attacks. We note that MPC protocols can only hide the inputs to the computation (e.g., the model or the dataset) up to what can be inferred from the output. Several recent works [59, 60, 61, 62, 63] have shown how black-box access to a model (in the case of an private inference service) can allow an adversary to learn information about the model or even recover its training data. Differentially-private training algorithms [9, 10] provide one defense against certain types of these attacks. Our focus in this work is on protecting the *computation* itself and ensure that there is no *additional* leakage about the inputs other than through the output. It is an interesting question to design a private training/inference protocol that also provides robustness against specific classes of model stealing/inversion attacks.

VI. Conclusion

In this paper, we introduce CrypTGPU, a new MPC framework that implements *all* of the cryptographic operations (both linear *and* non-linear) on the GPU. CrypTGPU is built on top of PyTorch [21] and CrypTen [24] to make it easy to use for machine learning developers and researchers. Our experiments show that leveraging GPUs can significantly accelerate the private training and inference for modern deep learning and make it practical to run privacy-preserving deep learning at the scale of ImageNet and with complex networks. In addition, our systematic analysis of different cryptographic protocols provides new insights for designing "GPU-friendly" cryptographic protocols for deep learning. This will be an important step towards bridging the roughly $1000\times$ gap that still remains between private machine learning and plaintext machine learning (on the GPU).

Acknowledgments

We thank Pavel Belevich, Shubho Sengupta, and Laurens van der Maaten for their feedback on system design and providing helpful pointers. D. J. Wu is supported by NSF CNS-1917414.

References

[1] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *IEEE Symposium on Security and Privacy*, pp. 19–38, 2017.

[2] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *ACM CCS*, pp. 35–52, 2018.

[3] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-party secure computation for neural network training," *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 3, pp. 26–49, 2019.

[4] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security*, pp. 2505–2522, 2020.

[5] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow: Secure tensorflow inference," in *IEEE Symposium on Security and Privacy*, pp. 336–353, 2020.

[6] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "FALCON: honest-majority maliciously secure framework for private deep learning," *Proc. Priv. Enhancing Technol.*, vol. 2021, 2021.

[7] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *STOC*, pp. 218–229, 1987.

[8] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in *STOC*, pp. 1–10, 1988.

[9] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *ACM CCS*, pp. 1310–1321, 2015.

[10] M. Abadi, A. Chu, I. J. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *ACM CCS*, pp. 308–318, 2016.

[11] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database." http://yann.lecun.com/exdb/mnist/.

[12] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[13] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, pp. 770–778, 2016.

[15] F.-F. Li, A. Karpathy, and J. Johnson, "Tiny ImageNet visual recognition challenge," 2017.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NeurIPS*, pp. 1106–1114, 2012.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[18] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, 1989.

[19] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," 2006.

[20] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural Comput.*, vol. 22, no. 12, pp. 3207–3220, 2010.

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *NeurIPS*, pp. 8024–8035, 2019.

[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016.

[23] "Cloud tensor processing units (tpus)." https://cloud.google.com/tpu/docs/tpus.

[24] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure multi-party computation meets machine learning," in *Proceedings of the NeurIPS Workshop on Privacy-Preserving Machine Learning*, 2020.

[25] M. Ito, A. Saito, and T. Nishizeki, "Secret sharing scheme realizing general access structure," *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, vol. 72, no. 9, pp. 56–64, 1989.

[26] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *ACM CCS*, pp. 805–817, 2016.

[27] "CUDA libraries documentation." https://docs.nvidia.com/cuda-libraries/index.html.

[28] A. C. Yao, "How to generate and exchange secrets (extended abstract)," in *FOCS*, pp. 162–167, 1986.

[29] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: high throughput 3pc over rings with application to secure prediction," in *ACM CCS*, pp. 81–92, 2019.

[30] A. Patra and A. Suresh, "BLAZE: blazing fast privacy-preserving machine learning," in *NDSS*, 2020.

[31] S. Kamara, P. Mohassel, and M. Raykova, "Outsourcing multi-party computation," *IACR Cryptol. ePrint Arch.*, vol. 2011, p. 272, 2011.

[32] "cuBLAS." https://docs.nvidia.com/cuda/cublas/index.html.

[33] "cuDNN." https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html.

[34] D. Demmler, T. Schneider, and M. Zohner, "ABY - A framework for efficient mixed-protocol secure two-party computation," in *NDSS*, 2015.

[35] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *CRYPTO*, pp. 420–432, 1991.

[36] F. Tramèr and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," in *ICLR*, 2019.

[37] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *EUROCRYPT*, pp. 225–255, 2017.

[38] R. Canetti, "Security and composition of multiparty cryptographic protocols," *J. Cryptol.*, vol. 13, no. 1, pp. 143–202, 2000.

[39] O. Goldreich, *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[40] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, pp. 807–814, 2010.

[41] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[42] "PyTorch/CSPRNG." https://github.com/pytorch/csprng.

[43] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: as easy as 1, 2, 3," in *Conference on High Performance Computing Networking, Storage and Analysis, SC*, pp. 16:1–16:12, 2011.

[44] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[45] Sameer Wagh and Shruti Tople and Fabrice Benhamouda and Eyal Kushilevitz and Prateek Mittal and Tal Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning." Available at https://github.com/snwagh/falcon-public.
[46] P. Team, "Alexnet." https://pytorch.org/hub/pytorch_vision_alexnet/.
[47] C. Gentry, *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.
[48] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *ICML*, pp. 201–210, 2016.
[49] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *ACM CCS*, pp. 619–631, 2017.
[50] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "Ezpc: Programmable, efficient, and scalable secure two-party computation for machine learning." Cryptology ePrint Archive, Report 2017/1109, 2017. https://eprint.iacr.org/2017/1109.
[51] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *ACM CCS*, pp. 707–721, 2018.
[52] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *USENIX Security Symposium*, pp. 1651–1669, 2018.
[53] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: xnor-based oblivious deep neural network inference," in *USENIX Security Symposium*, pp. 1501–1518, 2019.
[54] A. P. K. Dalskov, D. Escudero, and M. Keller, "Secure evaluation of quantized neural networks," *Proc. Priv. Enhancing Technol.*, vol. 2020, no. 4, pp. 355–375, 2020.
[55] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: fast and robust framework for privacy-preserving machine learning," *Proc. Priv. Enhancing Technol.*, vol. 2020, no. 2, pp. 459–480, 2020.
[56] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4pc framework for privacy preserving machine learning," in *NDSS*, 2020.
[57] A. A. Badawi, J. Chao, J. Lin, C. F. Mun, S. J. Jie, B. H. M. Tan, X. Nan, A. M. M. Khin, and V. Chandrasekhar, "Towards the alexnet moment for homomorphic encryption: HCNN, the first homomorphic cnn on encrypted data with gpus," *IEEE Transactions on Emerging Topics in Computing*, 2020.
[58] A. A. Badawi, B. Veeravalli, C. F. Mun, and K. M. M. Aung, "High-performance FV somewhat homomorphic encryption on gpus: An implementation using CUDA," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, pp. 70–95, 2018.
[59] G. Ateniese, L. V. Mancini, A. Spognardi, A. Villani, D. Vitali, and G. Felici, "Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers," *Int. J. Secur. Networks*, vol. 10, no. 3, pp. 137–150, 2015.
[60] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *ACM CCS*, pp. 1322–1333, 2015.
[61] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *USENIX Security Symposium*, pp. 601–618, 2016.
[62] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "DeepSecure: Scalable provably-secure deep learning," in *Annual Design Automation Conference*, pp. 1–6, 2018.
[63] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High-fidelity extraction of neural network models," *CoRR*, vol. abs/1909.01838, 2019.

APPENDIX A
NETWORK ARCHITECTURE

As discussed in Section IV-A, some of the models we consider (e.g., AlexNet and VGG-16) were designed for ImageNet, and are not directly compatible with smaller datasets such as CIFAR-10 and Tiny ImageNet. As such, when training or running inference with these models on the smaller datasets, we make adjustments to their "head architecture" (i.e., the fully-connected classification layers at the top of the network). In all settings, we keep the same "base architecture" (adapted from their description in the original papers [16, 17]). We describe the base AlexNet architecture we use in Fig. 5 and the head architectures for the different datasets in Fig. 6. We describe the base VGG-16 architecture we use in Fig. 7 and the head architectures for the different datasets in Fig. 8.

Layer	Input Dimension	Description	Output Dimension
Convolution	$32 \times 32 \times 3$	11×11 kernel, 9×9 padding, 4×4 stride	$10 \times 10 \times 96$
ReLU	$10 \times 10 \times 96$	$\mathsf{ReLU}(\cdot)$ on each input	$10 \times 10 \times 96$
Average Pooling	$10 \times 10 \times 96$	3×3 kernel, 2×2 stride	$4 \times 4 \times 96$
Convolution	$4 \times 4 \times 96$	5×5 kernel, 1×1 padding, 1×1 stride	$2 \times 2 \times 256$
ReLU	$2 \times 2 \times 256$	$\mathsf{ReLU}(\cdot)$ on each input	$2 \times 2 \times 256$
Average Pooling	$2 \times 2 \times 256$	2×2 kernel, 1×1 stride	$1 \times 1 \times 256$
Convolution	$1 \times 1 \times 256$	3×3 kernel, 1×1 padding, 1×1 stride	$1 \times 1 \times 384$
ReLU	$1 \times 1 \times 384$	$\mathsf{ReLU}(\cdot)$ on each input	$1 \times 1 \times 384$
Convolution	$1 \times 1 \times 384$	3×3 kernel, 1×1 padding, 1×1 stride	$1 \times 1 \times 384$
ReLU	$1 \times 1 \times 384$	$\mathsf{ReLU}(\cdot)$ on each input	$1 \times 1 \times 384$
Convolution	$1 \times 1 \times 384$	3×3 kernel, 1×1 padding, 1×1 stride	$1 \times 1 \times 256$
ReLU	$1 \times 1 \times 256$	$\mathsf{ReLU}(\cdot)$ on each input	$1 \times 1 \times 256$

Fig. 5: AlexNet [16] base architecture on CIFAR-10. The same architecture is also used for Tiny ImageNet and ImageNet, but applied to different input dimensions ($64 \times 64 \times 3$ for Tiny ImageNet and $224 \times 224 \times 3$ for ImageNet). The head architectures (classification layers) for CIFAR-10, Tiny ImageNet, and ImageNet vary (as a function of the input size and number of output classes) and are shown in Fig. 6.

Layer	Input Dimension	Description	Output Dimension
Flatten	$1 \times 1 \times 256$	Flatten input into a single dimension	256
Fully Connected	256	256×256 matrix multiplication	256
ReLU	256	$\mathsf{ReLU}(\cdot)$ on each input	256
Fully Connected	256	256×256 matrix multiplication	256
ReLU	256	$\mathsf{ReLU}(\cdot)$ on each input	256
Fully Connected	256	256×10 matrix multiplication	10

(a) Head architecture for CIFAR-10

Layer	Input Dimension	Description	Output Dimension
Average Pooling	$4 \times 4 \times 256$	2×2 kernel, 2×2 stride	$2 \times 2 \times 256$
Flatten	$2 \times 2 \times 256$	Flatten input into a single dimension	1024
Fully Connected	1024	1024×1024 matrix multiplication	1024
ReLU	1024	$\mathsf{ReLU}(\cdot)$ on each input	1024
Fully Connected	1024	1024×1024 matrix multiplication	1024
ReLU	1024	$\mathsf{ReLU}(\cdot)$ on each input	1024
Fully Connected	1024	1024×200 matrix multiplication	200

(b) Head architecture for Tiny ImageNet

Layer	Input Dimension	Description	Output Dimension
Average Pooling	$24 \times 24 \times 256$	4×4 kernel, 4×4 stride	$6 \times 6 \times 256$
Flatten	$6 \times 6 \times 256$	Flatten input into a single dimension	9216
Fully Connected	9216	9216×4096 matrix multiplication	4096
ReLU	4096	$\mathsf{ReLU}(\cdot)$ on each input	4096
Fully Connected	4096	4096×4096 matrix multiplication	4096
ReLU	4096	$\mathsf{ReLU}(\cdot)$ on each input	4096
Fully Connected	4096	4096×1000 matrix multiplication	1000

(c) Head architecture for ImageNet

Fig. 6: Head architecture of AlexNet for CIFAR-10, Tiny ImageNet, and ImageNet.

Layer	Input Dimension	Description	Output Dimension
Convolution	$32 \times 32 \times 3$	3×3 kernel, 1×1 padding, 1×1 stride	$32 \times 32 \times 64$
ReLU	$32 \times 32 \times 64$	ReLU($\cdot$) on each input	$32 \times 32 \times 64$
Convolution	$32 \times 32 \times 64$	3×3 kernel, 1×1 padding, 1×1 stride	$32 \times 32 \times 64$
ReLU	$32 \times 32 \times 64$	ReLU($\cdot$) on each input	$32 \times 32 \times 64$
Average Pooling	$32 \times 32 \times 64$	2×2 kernel, 2×2 stride	$16 \times 16 \times 64$
Convolution	$16 \times 16 \times 64$	3×3 kernel, 1×1 padding, 1×1 stride	$16 \times 16 \times 128$
ReLU	$16 \times 16 \times 128$	ReLU($\cdot$) on each input	$16 \times 16 \times 128$
Convolution	$16 \times 16 \times 128$	3×3 kernel, 1×1 padding, 1×1 stride	$16 \times 16 \times 128$
ReLU	$16 \times 16 \times 128$	ReLU($\cdot$) on each input	$16 \times 16 \times 128$
Average Pooling	$16 \times 16 \times 128$	2×2 kernel, 2×2 stride	$8 \times 8 \times 128$
Convolution	$8 \times 8 \times 128$	3×3 kernel, 1×1 padding, 1×1 stride	$8 \times 8 \times 256$
ReLU	$8 \times 8 \times 256$	ReLU($\cdot$) on each input	$8 \times 8 \times 256$
Convolution	$8 \times 8 \times 256$	3×3 kernel, 1×1 padding, 1×1 stride	$8 \times 8 \times 256$
ReLU	$8 \times 8 \times 256$	ReLU($\cdot$) on each input	$8 \times 8 \times 256$
Convolution	$8 \times 8 \times 256$	3×3 kernel, 1×1 padding, 1×1 stride	$8 \times 8 \times 256$
ReLU	$8 \times 8 \times 256$	ReLU($\cdot$) on each input	$8 \times 8 \times 256$
Average Pooling	$8 \times 8 \times 256$	2×2 kernel, 2×2 stride	$4 \times 4 \times 256$
Convolution	$4 \times 4 \times 256$	3×3 kernel, 1×1 padding, 1×1 stride	$4 \times 4 \times 512$
ReLU	$4 \times 4 \times 512$	ReLU($\cdot$) on each input	$4 \times 4 \times 512$
Convolution	$4 \times 4 \times 512$	3×3 kernel, 1×1 padding, 1×1 stride	$4 \times 4 \times 512$
ReLU	$4 \times 4 \times 512$	ReLU($\cdot$) on each input	$4 \times 4 \times 512$
Convolution	$4 \times 4 \times 512$	3×3 kernel, 1×1 padding, 1×1 stride	$4 \times 4 \times 512$
ReLU	$4 \times 4 \times 512$	ReLU($\cdot$) on each input	$4 \times 4 \times 512$
Average Pooling	$4 \times 4 \times 512$	2×2 kernel, 2×2 stride	$2 \times 2 \times 512$
Convolution	$2 \times 2 \times 512$	3×3 kernel, 1×1 padding, 1×1 stride	$2 \times 2 \times 512$
ReLU	$2 \times 2 \times 512$	ReLU($\cdot$) on each input	$2 \times 2 \times 512$
Convolution	$2 \times 2 \times 512$	3×3 kernel, 1×1 padding, 1×1 stride	$2 \times 2 \times 512$
ReLU	$2 \times 2 \times 512$	ReLU($\cdot$) on each input	$2 \times 2 \times 512$
Convolution	$2 \times 2 \times 512$	3×3 kernel, 1×1 padding, 1×1 stride	$2 \times 2 \times 512$
ReLU	$2 \times 2 \times 512$	ReLU($\cdot$) on each input	$2 \times 2 \times 512$
Average Pooling	$2 \times 2 \times 512$	2×2 kernel, 2×2 stride	$1 \times 1 \times 512$

Fig. 7: VGG-16 [17] base architecture for CIFAR-10 inputs. The same architecture is also used for Tiny ImageNet and ImageNet, but applied to different input dimensions ($64 \times 64 \times 3$ for Tiny ImageNet and $224 \times 224 \times 3$ for ImageNet). The head architectures (classification layers) for CIFAR-10, Tiny ImageNet, and ImageNet vary (as a function of the input size and number of output classes) and are shown in Fig. 8.

Layer	Input Dimension	Description	Output Dimension
Flatten	$1 \times 1 \times 512$	Flatten input into a single dimension	512
Fully Connected	512	512×256 matrix multiplication	256
ReLU	256	$\mathsf{ReLU}(\cdot)$ on each input	256
Fully Connected	256	256×256 matrix multiplication	256
ReLU	256	$\mathsf{ReLU}(\cdot)$ on each input	256
Fully Connected	256	256×10 matrix multiplication	10

(a) Head architecture for CIFAR-10.

Layer	Input Dimension	Description	Output Dimension
Average Pooling	$2 \times 2 \times 512$	2×2 kernel, 2×2 stride	$1 \times 1 \times 512$
Flatten	$1 \times 1 \times 512$	Flatten input into a single dimension	512
Fully Connected	512	512×512 matrix multiplication	512
ReLU	512	$\mathsf{ReLU}(\cdot)$ on each input	512
Fully Connected	512	512×512 matrix multiplication	512
ReLU	512	$\mathsf{ReLU}(\cdot)$ on each input	512
Fully Connected	512	512×200 matrix multiplication	200

(b) Head architecture for Tiny ImageNet.

Layer	Input Dimension	Description	Output Dimension
Average Pooling	$6 \times 6 \times 512$	2×2 kernel, 2×2 stride	$3 \times 3 \times 512$
Flatten	$3 \times 3 \times 512$	Flatten input into a single dimension	4608
Fully Connected	4608	4608×4096 matrix	4096
ReLU	4096	$\mathsf{ReLU}(\cdot)$ on each input	4096
Fully Connected	4096	4096×4096 matrix	4096
ReLU	4096	$\mathsf{ReLU}(\cdot)$ on each input	4096
Fully Connected	4096	4096×1000 matrix	1000

(c) Head architecture for ImageNet.

Fig. 8: Head architecture of VGG-16 for CIFAR-10, Tiny ImageNet, and ImageNet.