CRYPTEN: Secure Multi-Party Computation Meets Machine Learning

Brian Knott **Shobha Venkataraman** **Awni Hannun**
Shubho Sengupta **Mark Ibrahim** **Laurens van der Maaten**
Facebook AI Research
{brianknott,shobha,awni,ssengupta,marksibrahim,lvdmaaten}@fb.com

Abstract

Secure multi-party computation (MPC) allows parties to perform computations on data while keeping that data private. This capability has great potential for machine-learning applications: it facilitates training of machine-learning models on private data sets owned by different parties, evaluation of one party's private model using another party's private data, *etc.* Although a range of studies implement machine-learning models via secure MPC, such implementations are not yet mainstream. Adoption of secure MPC is hampered by the absence of flexible software frameworks that "speak the language" of machine-learning researchers and engineers. To foster adoption of secure MPC in machine learning, we present CRYPTEN: a software framework that exposes popular secure MPC primitives via abstractions that are common in modern machine-learning frameworks, such as tensor computations, automatic differentiation, and modular neural networks. This paper describes the design of CRYPTEN and measure its performance on state-of-the-art models for text classification, speech recognition, and image classification. Our benchmarks show that CRYPTEN's GPU support and high-performance communication between (an arbitrary number of) parties allows it to perform efficient private evaluation of modern machine-learning models under a *semi-honest* threat model. For example, two parties using CRYPTEN can securely predict phonemes in speech recordings using Wav2Letter [18] faster than real-time. We hope that CRYPTEN will spur adoption of secure MPC in the machine-learning community.

1 Introduction

Secure multi-party computation (MPC; [31, 74]) allows parties to collaboratively perform computations on their combined data sets without revealing the data they possess to each other. This capability of secure MPC has the potential to unlock a variety of machine-learning applications that are currently infeasible because of data privacy concerns. For example, secure MPC can allow medical research institutions to jointly train better diagnostic models without having to share their sensitive patient data [28] or allow social scientists to analyze gender wage gap statistics without companies having to share sensitive salary data [44]. The prospect of such applications of machine learning with rigorous privacy and security guarantees has spurred a number of studies on machine learning via secure MPC [40, 43, 50, 62, 67, 71, 72]. However, at present, adoption of secure MPC in machine learning is still relatively limited considering its wide-ranging potential. One of the main obstacles to widespread adoption is that the complexity of secure MPC techniques puts them out of reach for most machine-learning researchers, who frequently lack in-depth knowledge of cryptographic techniques.

To foster the adoption of secure MPC techniques in machine learning, we present CRYPTEN: a flexible software framework that aims to make modern secure MPC techniques accessible to machine-learning researchers and developers without a background in cryptography. Specifically, CRYPTEN

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

provides a comprehensive tensor-computation library in which all computations are performed via secure MPC. CrypTen's API closely follows the API of the popular PyTorch framework for machine learning [57, 58], which makes it easy to use for machine-learning practitioners. For example, it provides automatic differentiation and a modular neural-network package. CrypTen assumes an *semi-honest* threat model [31, §2.3.2] and works for an arbitrary number of parties. To make private training and inference efficient, CrypTen off-loads computations to the GPU and uses high-performance communication libraries to implement interactions between parties.

The paper presents: (1) an overview of CrypTen's design principles; (2) a description of the design of CrypTen and of the secure MPC protocols implemented; (3) a collection of benchmark experiments using CrypTen to run private versions of state-of-the-art models for text classification, speech recognition, and image classification; and (4) a discussion of open problems and a roadmap for the further development of CrypTen. Altogether, the paper demonstrates that CrypTen's flexible, PyTorch-like API makes private inference and training of modern machine-learning models easy to implement and efficient. For example, CrypTen allows two parties to privately classify an image [27, 37] in 2-3 seconds, or to securely make phoneme predictions for 16kHz speech recordings [18] faster than real-time. We hope that CrypTen's promising performance and ease-of-use will foster the adoption of secure MPC by the machine-learning community, and pave the way for a new generation of secure and private machine-learning systems.

2 Related Work

CrypTen is part of a large body of work that develops secure MPC protocols for machine learning; see Appendix D. Most closely related to our work is CryptGPU [67], which implements an 2-out-of-3 replicated secret sharing protocol [4, 39] *on top of* CrypTen. Like CrypTen, CryptGPU provides security against *semi-honest* corruption, but it is limited to the three-party setting. CryptGPU is one of several protocols optimized for the three-party setting. For example, Falcon [72] implements a *maliciously secure* three-party MPC protocol, combining techniques from SecureNN [71] and ABY3 [50]. Falcon allows evaluation and training of convolutional networks such as AlexNet [42] and VGG [66]. Other systems that work in this setting include Astra [17], Blaze [59], and CrypTFlow [43].

There also exists a family of two-party systems that, like CrypTen, assume a semi-honest threat model. These systems include Gazelle [40], Chameleon [62], EzPC [16], MiniONN [47], SecureML [51], PySyft [64], and Delphi [49]. XONN [63] also works in the two-party setting but provides malicious security. Compared to these systems, CrypTen provides a more flexible machine-learning focused API[1] that supports reverse-mode automatic differentiation, implements a rich set of functions, and natively runs on GPUs. Moreover, CrypTen supports a wider range of use cases by working with an arbitrary number of parties, and make communication between parties efficient via communication primitives that were optimized for high-performance distributed computing.

3 Design Principles

In the development of CrypTen, we adopted the following two main design principles:

Machine-learning first API. CrypTen has a general purpose, machine-learning first API design. Most other secure MPC frameworks [36] adopt an API that stays close to the underlying MPC protocols. This hampers adoption of these frameworks in machine learning, for example, because they do not natively support tensor operations (but only scalar operations) and because they lack features that machine-learning researchers have come to expect, such as automatic differentiation. Instead, CrypTen implements the tensor-computation API of the popular PyTorch machine-learning framework [57], implements reverse-mode automatic differentiation, provides a modular neural-network package with corresponding learning routines, and supports GPU computations. We aim to allow developers to transition code from PyTorch to CrypTen by changing a single Python `import`.

Eager execution. CrypTen adopts an imperative programming model. This is different from existing MPC frameworks, which generally implement compilers for their own domain-specific languages [36]. While compiler approaches have potential performance benefits, they slow down the

[1] CrypTFlow [43] also provides such an API by integrating deeply with TensorFlow [1], but unlike CrypTen, it does not support PyTorch's eager execution model [58] or GPU support.

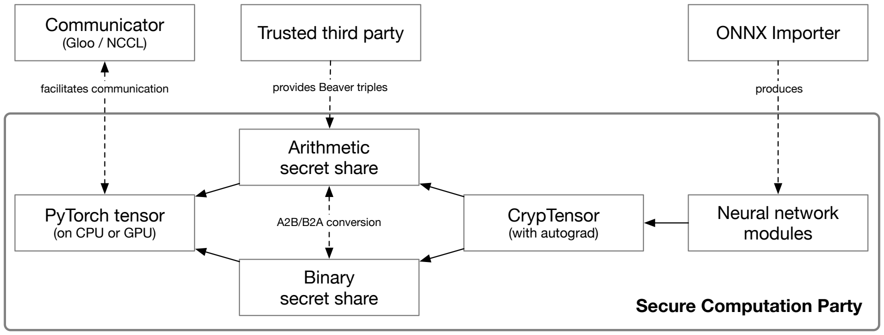

Figure 1: High-level overview of the design of CRYPTEN. See text in Section 4 for details.

development cycle, make debugging harder, and prevent users from using arbitrary host-language constructs [3]. Instead, CRYPTEN follows the recent trend in machine learning away from graph compilers [1] to frameworks that eagerly execute computations [3, 58], providing a better developer experience. Yet, CRYPTEN is performant because it implements state-of-the-art secure MPC protocols (for settings with arbitrary number of parties), because it uses PyTorch's highly optimized tensor library for most computations, because computations can be off-loaded to the GPU, and because it uses communication libraries that were optimized for high-performance distributed computing.

4 Design Overview

Figure 1 gives an overview of CRYPTEN's design. Parties perform computations using efficient PyTorch tensor operations. Because secure MPC computations are integer computations that are not natively supported on GPUs, CRYPTEN maps between integer and floating-point computations on GPUs; see Section 5.3. The multi-party computations are implemented on arithmetic and binary secret shares [23, 33]; see Section 5.1. Whereas many computations can be performed directly on arithmetic secret shares, others require conversion between arithmetic and binary secret shares (A2B) and back (B2A); see Section 5.2. Some multi-party computations require interaction between parties via a *communicator* that employs the high-performance communication primitives in Gloo [32] and NCCL [54]. Some multi-party computations require Beaver triples [7], which are supplied by a *trusted third party* (TTP).[2]

```python
import crypten, torch

# set up communication and sync random seeds:
crypten.init()

# secret share tensor:
x = torch.tensor([1.0, 2.0, 3.0])
x_enc = crypten.cryptensor(x, src=0)

# reveal secret shared tensor:
x_dec = x_enc.get_plain_text()
assert torch.all_close(x_dec, x)

# add secret shared tensors:
y = torch.tensor([2.0, 3.0, 4.0])
y_enc = crypten.cryptensor(y, src=0)
xy_enc = x_enc + y_enc
xy_dec = xy_enc.get_plain_text()
assert torch.all_close(xy_dec, x + y)
```

Figure 2: Example of secret-sharing tensors, revealing tensors, and private addition in CRYPTEN.

All secure computations are wrapped in a `CrypTensor` object that implements the PyTorch tensor API and that provides reverse-mode automatic differentiation (autograd) to enable gradient-based training of arbitrary (deep) learning models. Figure 2 illustrates `CrypTensor` creation, *i.e.*, how tensors are secret-shared and revealed, as well as a simple computation (addition). Note that each party involved in the multi-party computation executes the same code. Whenever communication between the parties is required (*e.g.*, as part of private multiplications), the communication acts as a synchronization point between the parties. The `crypten.init()` call is required once to establish the communication channel. In the example, the input tensor for the creation of the arithmetic secret share is provided party `src=0`, which indicates the rank[3] of the party that supplies the data to be secret-shared (the other parties executing this code may provide `None` as input).

[2] CRYPTEN adopts a trusted third party for generating Beaver triples for efficiency reasons, but we are planning to add TTP-free solutions based on additive homomorphic encryption [55] or oblivious transfer [41].

[3] CRYPTEN relies on MPI primitives for communication: each party knows their rank and the world size.

To enable deep-learning use cases, CrypTen allows implementing neural networks following PyTorch's API. Figure 3 shows how to create and encrypt neural networks and how to use automatic differentiation in CrypTen. The example assumes that some training `sample` and the associated `target` label are provided by the party with rank 0 (note the value of `src`). As illustrated by the example, CrypTen's API closely follows that of PyTorch. Indeed, it is possible to write a single training loop that can be used to train models using CrypTen or PyTorch without code changes. This makes it easy to adapt PyTorch code to use secure MPC for its computations, and it also makes debugging easier. The appendix presents a table listing all tensor functions that `CrypTensor` implements.

To enable interoperability with existing machine-learning platforms, neural networks can be imported into CrypTen via ONNX. Figure 4 shows how a PyTorch model is imported into CrypTen. The example illustrates how CrypTen makes private inference with a ResNet-18 easy. The example in the figure also demonstrates CrypTen's GPU support. One caveat is that all parties must use the same type of device (*i.e.*, CPU or GPU) for computations.

```
import crypten.optimizer as optimizer
import crypten.nn as nn

# create model, criterion, and optimizer:
model_enc = nn.Sequential(
    nn.Linear(sample_dim, hidden_dim),
    nn.ReLU(),
    nn.Linear(hidden_dim, num_classes),
).encrypt()
criterion = nn.CrossEntropyLoss()
optimizer = optimizer.SGD(
    model_enc.parameters(), lr=0.1, momentum=0.9,
)

# perform prediction on sample:
target_enc = crypten.cryptensor(target, src=0)
sample_enc = crypten.cryptensor(sample, src=0)
output_enc = model_enc(sample_enc)

# perform backward pass and update parameters:
model_enc.zero_grad()
loss_enc = criterion(output_enc, target_enc)
loss_enc.backward()
optimizer.step()
```

Figure 3: Example using neural networks and automatic differentiation in CrypTen.

```
import torchvision.datasets as datasets
import torchvision.models as models
import torchvision.transforms as transforms

# download and set up ImageNet dataset:
transform = transforms.ToTensor()
dataset = datasets.ImageNet(
    imagenet_folder, transform=transform,
)

# secret share pre-trained ResNet-18 on GPU:
model = models.resnet18(pretrained=True)
model_enc = crypten.nn.from_pytorch(
    model, dataset[0],
).encrypt().cuda()

# perform inference on secret-shared images:
for image in dataset:
    image_enc = crypten.cryptensor(image).cuda()
    output_enc = model_enc(image_enc)
    output = output_enc.get_plain_text()
```

Figure 4: Private inference on secret-shared images using a secret-shared ResNet-18 model on GPU.

5 Secure Computations

To facilitate secure computations, CrypTen implements arithmetic secret sharing [23, 24] and binary secret sharing [33], as well as conversions between these two types of sharing [25]. Arithmetic secret sharing is particularly well-suited for operations that are common in modern machine-learning models, such as matrix multiplications and convolutions. Binary secret sharing is required for evaluating certain other common functions, such as rectified linear units. We provide a high-level overview of CrypTen's secure computation protocol here; a detailed description is presented in the appendix.

5.1 Secret Sharing

Arithmetic secret sharing shares a scalar value $x \in \mathbb{Z}/Q\mathbb{Z}$, where $\mathbb{Z}/Q\mathbb{Z}$ denotes a ring with Q elements, across parties $p \in \mathcal{P}$. We denote the sharing of x by $[x] = \{[x]_p\}_{p \in \mathcal{P}}$, where $[x]_p \in \mathbb{Z}/Q\mathbb{Z}$ indicates party p's share of x. The shares are constructed such that their sum reconstructs the original value x, that is, $x = \sum_{p \in \mathcal{P}} [x]_p \mod Q$. To share a value x, the parties generate a pseudorandom zero-share [19] with $|\mathcal{P}|$ random numbers that sum to 0. The party that possesses the value x adds x to their share and discards x. We use a fixed-point encoding to obtain x from a floating-point value, x_R. To do so, we multiply x_R with a large scaling factor B and round to the nearest integer: $x = \lfloor B x_R \rceil$, where $B = 2^L$ for some precision of L bits. To decode a value, x, we compute $x_R \approx {}^{x}\!/_{B}$.

Binary secret sharing is a special case of arithmetic secret sharing that operates within the binary field $\mathbb{Z}/2\mathbb{Z}$. A binary secret share, $\langle x \rangle$, of a value x is formed by arithmetic secret shares of the bits of x, setting $Q = 2$. Each party $p \in \mathcal{P}$ holds a share, $\langle x \rangle_p$, such that $x = \bigoplus_{p \in \mathcal{P}} \langle x \rangle_p$ is satisfied.

Conversion from $[x]$ to $\langle x \rangle$ is implemented by having the parties create a binary secret share of their $[x]_p$ shares, and summing the resulting binary shares. Specifically, the parties create a binary secret share, $\langle [x]_p \rangle$, of all the bits in $[x]_p$. Subsequently, the parties compute $\langle x \rangle = \sum_{p \in \mathcal{P}} \langle [x]_p \rangle$ using a carry-lookahead adder in $\log_2(|\mathcal{P}|) \log_2(L)$ communication rounds [15, 22].

Conversion from $\langle x \rangle$ to $[x]$ is achieved by computing $[x] = \sum_{b=1}^{B} 2^b \left[\langle x \rangle^{(b)} \right]$, where $\langle x \rangle^{(b)}$ denotes the b-th bit of the binary share $\langle x \rangle$ and B is the total number of bits in the shared secret, $\langle x \rangle$. To create an arithmetic share of a bit, the parties use secret shares, $\left([r^{(b)}], \langle r^{(b)} \rangle \right)$, of random bits $r^{(b)}$. The random bits are provided by the TTP, but we plan to add an implementation that generates them off-line via oblivious transfer [41]. The parties use $\langle r^{(b)} \rangle$ to mask $\langle x \rangle^{(b)}$ and reveal the resulting masked bit $z^{(b)}$. Subsequently, they compute $\left[\langle x \rangle^{(b)} \right] = \left[r^{(b)} \right] + z^{(b)} - 2 \left[r^{(b)} \right] z^{(b)}$.

5.2 Secure Computation

Arithmetic and binary secret shares have homomorphic properties that can be used to implement secure computations. All computations in CrypTen are based on private addition and multiplication.

Private addition of two arithmetically secret shared values, $[z] = [x] + [y]$, is implemented by having each party p sum their shares of $[x]$ and $[y]$: each party $p \in \mathcal{P}$ computes $[z]_p = [x]_p + [y]_p$.

Private multiplication is implemented using random Beaver triples [7], $([a], [b], [c])$ with $c = ab$, that are provided by the TTP. The parties compute $[\epsilon] = [x] - [a]$ and $[\delta] = [y] - [b]$, and decrypt ϵ and δ without information leakage due to the masking. They compute the result $[x][y] = [c] + \epsilon[b] + [a]\delta + \epsilon\delta$, using trivial implementations of addition and multiplication of secret shares with public values.

Linear functions are trivially implemented as combinations of private addition and multiplication. This allows CrypTen to compute dot products, outer products, matrix products, and convolutions.

Non-linear functions are implemented using standard approximations that only require private addition and multiplication. Specifically, CrypTen evaluates exponentials using a limit approximation, logarithms using Householder iterations [38], and reciprocals using Newton-Rhapson iterations. This allows CrypTen to implement functions that are commonly used in machine-learning models, including the sigmoid, softmax, and logistic-loss functions, as well as their gradients.

Comparators are implemented using a function that evaluates $[z < 0]$ by: (1) converting $[z]$ to a binary secret-share $\langle z \rangle$; (2) computing its sign bit, $\langle b \rangle = \langle z \rangle >> (L - 1)$; and (3) converting the resulting bit to an arithmetic sharing $[b]$. This function allows CrypTen to implement arbitrary comparators. For example, it evaluates $[x < y]$ by computing $[z] = [x] - [y]$ and evaluating $[z < 0]$. Similarly, CrypTen can evaluate: (1) the sign function via $\text{sign}([x]) = 2[x > 0] - 1$; (2) the absolute value function via $|[x]| = [x] \, \text{sign}([x])$; and (3) rectified linear units via $\text{ReLU}([x]) = [x][x > 0]$. CrypTen also supports multiplexing; to do so, it evaluates $[c \, ? \, x : y] = [c][x] + (1 - [c])[y]$.

Lemma 1. *The* CrypTen *secure-computation protocol is secure against information leakage against any static passive adversary corrupting up to* $|\mathcal{P}| - 1$ *of the* $|\mathcal{P}|$ *parties involved in the computation.*

The proof of this lemma follows trivially from [9, 12, 22, 25], and is given in the appendix. We adopt a protocol that provides security under a *semi-honest* threat model because it enables a wide range of use cases of secure machine learning, whilst being more efficient than maliciously secure protocols.

5.3 Off-loading Computations to the GPU

Hardware acceleration via GPUs is a critical component for training and inference in modern machine-learning models. Akin to frameworks such as PyTorch [58] and TensorFlow [1], CrypTen can off-load computations to the GPU. On the GPU, it uses highly-optimized implementations for a range of functions that are provided by CUDA libraries such as cuBLAS [20] and cuDNN [21].

Unfortunately, these libraries are designed for computations on floating-point numbers and do not support the integer types required to perform computations on L-bit fixed-point numbers. Akin to [67], we circumvent this problem by observing that for all integers $a, b \in \mathbb{Z} \cap [-2^{26}, 2^{26}]$, we can compute the product ab using 64-bit floating-point representations and still recover the correct value over the integers. Specifically, CrypTen splits each 64-bit variable into four components, $a = a_0 + 2^{16} a_1 + 2^{32} a_2 + 2^{48} a_3$, where each a_i represents a 16-bit integer component. We compute a product ab of 64-bit integers by summing 10 pairwise products of their 16-bit components.

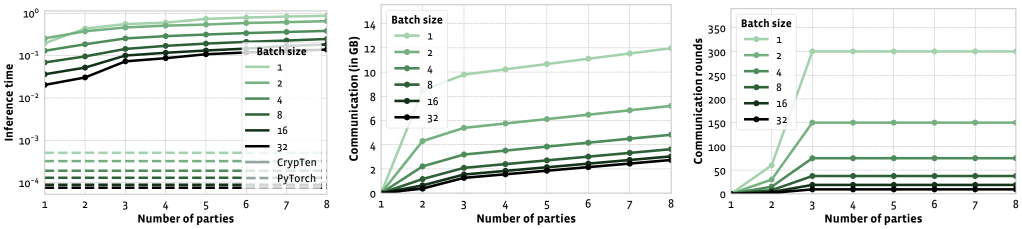

Figure 5: Benchmarks for inference with text-sentiment classification model on GPUs in CrypTen and PyTorch. **Left:** Average wall-clock time per sample (in seconds). **Middle:** Number of bytes communicated per sample, per party (in GB). **Right:** Number of communication rounds per sample.

The pairwise products of the 16-bit components are computed in parallel using highly optimized floating-point CUDA kernels. The same approach is used for matrix multiplications and convolutions. CrypTen further optimizes this approach by splitting into only 3 components of 22-bits each when possible, which reduces the number of pairwise products required to 6 (see [67, Remark II.1]).

6 Benchmarks

To measure the performance of CrypTen, we performed experiments on three tasks: (1) text classification using a linear model that learns word embeddings; (2) speech recognition using the Wav2Letter model [18]; and (3) image classification using residual networks [37] and vision transformers [27]. Because of space constraints, we focus on private inference using a secret-shared model on secret-shared data here, but our benchmark results with private training are very similar.

We performed benchmark experiments on a proprietary cluster, testing inference on both CPUs (Intel Skylake 18-core 1.6GHz) and GPUs (nVidia P100). We set the number of OpenMP threads to 1 in all benchmarks. All experiments were performed with the parties running in separate processes on a single machine. For GPU experiments, each party was assigned its own GPU. Although this setup is faster than a scenario in which each party operates its own machine,[4] we believe our benchmark results provide a good sense of CrypTen's performance. We average computation times over 30 batches, excluding the computation on the first batch as that computation may include CuDNN benchmarking. Code reproducing the results of our experiments is available on `https://crypten.ai`.

In our benchmarks, we focus on comparing (ciphertext) CrypTen computation with (plaintext) PyTorch computation. We refer the reader to [35, 67] for benchmarks that compare CrypTen to other secure MPC frameworks. Specifically, [35] finds CrypTen is 11-18× faster than PySyft [64] and approximately 3× faster than TF-Trusted [14] in MNIST classification [45] on CPU.

6.1 Text Classification

We performed text-sentiment classification experiments on the Yelp review dataset [75] using a model that consists of a linear layer operating on word embeddings. The embedding layer contains 32-dimensional embeddings of $519,820$ words, and the linear layer produces a binary output indicating the sentiment of the review. We evaluated the model on GPUs, varying the batch size and the number of parties participating. The normalized mean squared error ($\|\mathbf{x}-\mathbf{y}\|^2/\|\mathbf{x}\|^2$) between the output of the CrypTen model and that of its PyTorch counterpart was smaller than $4 \cdot 10^{-4}$ in all experiments.

Figure 5 presents the results of our experiments. The figure shows inference time *per sample* (in seconds) as a function of the number of parties involved in the computation for varying batch sizes (left); the amount of communication required per sample, *per party* (in GB); and the number of communication rounds required per sample. We include results in which the number of parties is 1: herein, we run the CrypTen protocol but involve no other parties, which implies that the single party is running the protocol on unencrypted data. One-party results allow us to bisect different sources of computational overhead: specifically, they separate overhead due to communication from overhead due to fixed-point encoding, function approximations, and (lack of) sparse-matrix operations.

[4] Communication between GPUs in two machines connected via InfiniBand has approximately 20× lower throughput than communication between two GPUs in the same machine via NVLink (25GB/s versus 600GB/s).

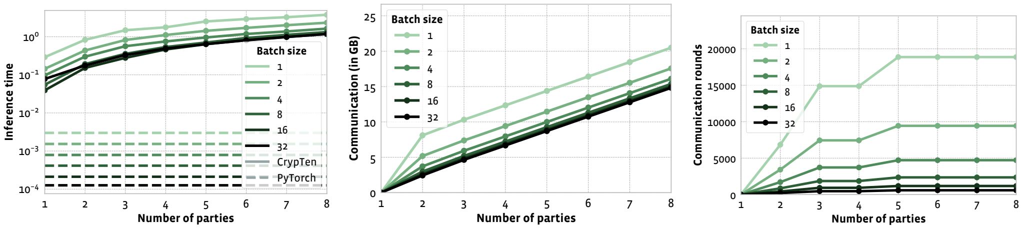

Figure 6: Benchmarks for inference with Wav2Letter model on GPUs in CrypTen and PyTorch. **Left:** Average wall-clock time per sample (in seconds). **Middle:** Number of bytes communicated per sample, per party (in GB). **Right:** Number of communication rounds per sample.

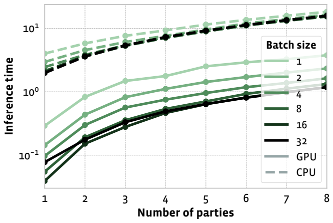

Figure 7: Wall-clock time per sample (in sec.) for Wav2Letter inference on CPUs and GPUs.

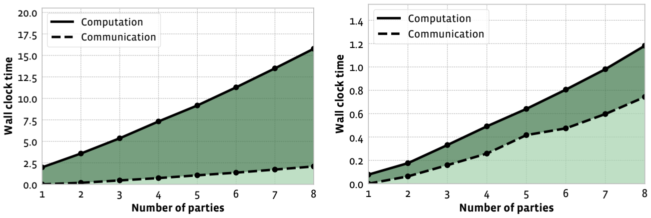

Figure 8: Average wall-clock time per sample (in seconds) for communication and computation during inference with Wav2Letter model on CPU (**left**) and GPU (**right**).

The results in Figure 5 show that CrypTen is about 2.5–3 orders of magnitude slower than PyTorch in text-sentiment classification, depending on the number of parties involved. Most computational overhead is the word embedding layer: whereas PyTorch can evaluate this layer efficiently via a sparse matrix multiplication, CrypTen cannot do sparse lookups as they would reveal information on the encrypted input. Instead, CrypTen performs a full matrix multiplication between the word-count vector and the embedding matrix. Yet, text sentiment predictions are quite fast in CrypTen: inference takes only 0.03 seconds per sample in the two-party setting with a batch size of 32.

The results also show that increasing the batch size is an effective way to reduce inference time and communication per sample. The number of communication rounds is independent of the batch size, which means communication rounds can be amortized by using larger batch sizes. The number of bytes communicated is partly amortized as well because the size of weight tensors (*e.g.*, in linear layers) does not depend on batch size. The results also show that whereas the number of communication rounds increases when moving from two-party to three-party computation, it remains constant afterwards. The larger number of communication rounds for three-party computation stems from the public division protocol, which requires additional communication rounds when more than two parties are involved to prevent wrap-around errors (see the appendix for details).

6.2 Speech Recognition

We performed speech-recognition experiments using Wav2Letter [18] on the LibriSpeech dataset [56]. The LibriSpeech dataset contains 16 kHz audio clips represented as a waveform ($16,000$ samples per second). Because the audio clips vary in length, we clip all of them to 1 second for the benchmark. Wav2Letter is a network with 13 convolutional layers using rectified linear unit (ReLU; [53]) activations.[5] The network operates directly on the waveform input, predicting one of 29 labels (26 letters plus 3 special characters). The first two layers use a filter size of 250 (with stride 160) and 48 (stride 2). The next seven layers use filter size 7, followed by two layers with filter size 32 and 1 (all with stride 1). All layers except the last two have 250 channels. The last two layers have $2,000$ channels.

The results in Figure 6 show that CrypTen is about 2.5–3 orders of magnitude slower than PyTorch depending on the number of parties involved. For Wav2Letter, the overhead is largely due to the ReLU layers in the network: evaluating a ReLU function requires a comparison, which involves a

[5] We used the reference implementation of Wav2Letter in `torchaudio`.

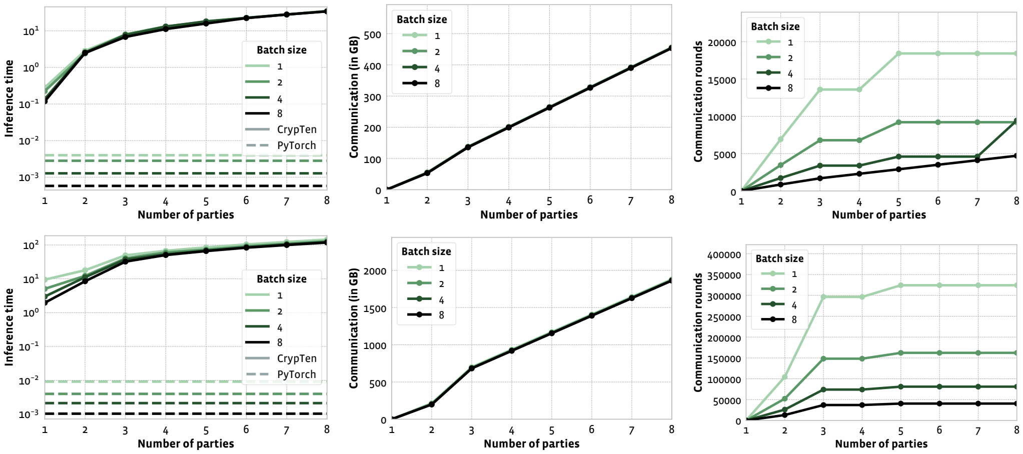

Figure 9: Benchmarks for inference with image-classification models on GPUs in CrypTen and PyTorch. **Top:** Results for ResNet-18 model. **Bottom:** Results for ViT-B/16 vision transformer. **Left:** Average wall-clock time per sample (in seconds). **Middle:** Number of bytes communicated per sample, per party (in GB). **Right:** Number of communication rounds per sample.

conversion between arithmetic and binary secret sharing and back (see the appendix). The number of communication rounds increases when the number of parties grows beyond 4: CrypTen uses a tree reduction for the summation in the comparator protocol, which implies that the number of communication rounds grows whenever the number of parties increases from 2^k to 2^{k+1}.

Figure 7 also presents results comparing Wav2Letter inference time between CPUs and GPUs. The results in the figure show that CrypTen is 1-2 orders of magnitude faster on GPUs than on CPUs. In real-world settings, this speedup can make the difference between a secure MPC use case being practical or not. Figure 8 shows how much wall-clock time is spent on communication and computation, respectively, when performing inference with Wav2Letter (using batch size 32). The results suggest that, whereas multi-party evaluation is compute-bound on CPU, it is communication-bound on GPU. On GPUs, 63% of the time is spent on communication in eight-party computation.

6.3 Image Classification

We performed image-classification experiments on the ImageNet dataset using residual networks (ResNets; [37]) and vision transformers (ViT; [27]).[6] We experimented with a ResNet-18 with 18 convolutional layers and with a ViT-B/16 model that has 12 multi-head self-attention layers with 12 heads each, operating on image patches of 16×16 pixels. Following common practice [37], we preprocess images by rescaling them to size 256×256 and taking a center crop of size 224×224.

Figure 9 presents the results of our image-classification benchmarks, which show that two parties can securely evaluate a ResNet-18 model in 2.49 seconds and a ViT-B/16 model in 8.47 seconds. A notable difference compared to the prior results is that the number of bytes communicated per sample is no longer reduced by increasing the batch size. The reason for this is that the vast majority of communication involves tensors that have the same size as intermediate activation functions: activation tensors are much larger than weight tensors in image-classification models. The amount of communication required to evaluate the ViT-B/16 model is particularly high due to the repeated evaluation of the softmax function in the attention layer of Transformers [69]. We also observe that in ResNet-18, the number of communication rounds grows faster than expected for larger batch sizes. The reason for this is that the carry-lookahead adder [22] used in the conversion from $[x]$ to $\langle x \rangle$ is very memory-intensive. When CrypTen runs out of GPU memory, it replaces the adder by an implementation that requires $O(|\mathcal{P}|)$ communication rounds (compared to $(\log_2 |\mathcal{P}|)$ for the carry-lookahead adder) but that requires less memory.

[6] We adopted the ResNet implementation from `torchvision` and the ViT implementation from `https://github.com/rwightman/pytorch-image-models`. ViT's normalized mean squared error is larger than for other models because our Gaussian error function approximation converges slowly; see Section C.2.6.

7 Conclusion and Future Work

In this paper, we have introduced and benchmarked CrypTen. We hope that CrypTen's flexible, machine-learning first API design and performance can help foster adoption of secure MPC in machine learning. We see the following directions for future research and development of CrypTen.

Numerical issues are substantially more common in CrypTen implementations of machine-learning algorithms than in their PyTorch counterparts. In particular, the fixed-point representation with L bits of precision ($L = 16$ by default) is more prone to numerical overflow or underflow than floating-point representations. Moreover, arithmetic secret shares are prone to *wrap-around* errors in which the sum of the shares $[x]_p$ exceeds the size of the ring, $Q = 2^{64}$. Wrap-around errors can be difficult to debug because they may only arise in the multi-party setting, in which no individual party can detect them. We plan to implement tools in CrypTen that assist users in debugging such numerical issues.

End-to-end privacy requires seamless integration between data-processing frameworks, such as secure SQL implementations [5], and data-modeling frameworks like CrypTen. In "plaintext" software, such frameworks are developed independently and combined via "glue code" or platforms that facilitate the construction of processing and modeling pipelines. Real-world use cases of machine learning via secure MPC require the development of a platform that makes the integration of private data processing and modeling seamless, both from an implementation and a security point-of-view.

Differential privacy mechanisms may be required in real-world applications of CrypTen in order to provide rigorous guarantees on the information leakage that inevitably occurs when the results of a private computation are publicly revealed [29]. CrypTen implements sampling algorithms for the Bernoulli, Laplace, and Gaussian distributions (see appendix), which allows for the implementation of randomized response [73], the Laplace mechanism [30], and the Gaussian mechanism [6, 29] (although care must be taken when implementing these mechanisms [13, 48]). In future work, we aim to use these mechanisms, for example, to do a secure MPC implementation of DP-SGD [2].

Threat models may vary per use case. Specifically, some use cases may require malicious security or may not provide a TTP. Possible extensions may include support for malicious security via message authentication codes [23], as well as support for Beaver triple generation via additive homomorphic encryption [55], oblivious transfer [41], or more recent methods [10] to eliminate the need for a TTP.

Model architecture design for secure MPC is another important direction for future research. Following prior work in this research area, this study has focused on implementing *existing* machine-learning models in a secure MPC framework. However, these models were designed based on computational considerations in "plaintext" implementations of the models on modern GPU or TPU hardware. The results of our benchmarks suggest that this may be suboptimal because those considerations are very different in a secure MPC environment. For example, the evaluation of softmax functions over large numbers of values requires a lot of communication in secure MPC, which makes attention layers very slow. This implies that multilayer perceptron models [68] are likely much more efficient than vision transformers [27, 69] for image classification. We hope that CrypTen's machine-learning API and ease of use will spur studies that design model architectures specifically optimized for a secure MPC environment, for example, via neural architecture search [46, 49, 76].

8 Broader Impact

Although we believe that the adoption of secure MPC in machine learning can lead to the development of AI systems that are substantially more private and secure, we note that there are also potential downsides to such adoption. In particular, because the computations in secure MPC are performed on encrypted data, it can be harder to do quality control of AI systems implemented in CrypTen. For example, it is impossible to inspect the values of intermediate activations (or even model outputs) unless all parties agree to reveal those values. This may make it harder to explain why a model makes a certain decision [26] or to detect data-poisoning attacks [8]. Indeed, there exist fundamental trade-offs between privacy and utility [61] and those trade-offs apply to CrypTen users, too.

It is also worth noting that, although the protocols implemented in CrypTen come with rigorous cryptographic guarantees, practical implementations of these protocols may be broken by other means. For example, we have no reason to assume that CrypTen would not be susceptible to side-channel attacks [65]. Hence, good data stewardship remains essential even when using secure computation.

Acknowledgments. We thank Joe Spisak, Sijun Tan, Gregory Chanan, Igor Fedan, and the PyTorch team for their support. We thank Mark Tygert, Anderson Nascimento, Amrita Roy Chowdhury, and anonymous reviewers for helpful discussions and feedback on early versions of this paper.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] M. Abadi, A. Chu, I. Goodfellow, H. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proceedings of the CCS*, pages 308–318, 2016.

[3] A. Agrawal, A. N. Modi, A. Passos, A. Lavoie, A. Agarwal, A. Shankar, I. Ganichev, J. Levenberg, M. Hong, R. Monga, and S. Cai. TensorFlow Eager: A multi-stage, python-embedded dsl for machine learning. In *arXiv:1903.01855*, 2019.

[4] 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*, pages 805–817, 2016.

[5] D. W. Archer, D. Bogdanov, L. Kamm, Y. Lindell, K. Nielsen, J. I. Pagter, N. P. Smart, and R. N. Wright. From keys to databases – real-world applications of secure multi-party computation. *Computer Journal*, 61(12):1749–1771, 2018.

[6] B. Balle and Y.-X. Wang. Improving the Gaussian mechanism for differential privacy. In *Proceedings of International Conference on Machine Learning*, 2018.

[7] D. Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[8] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In *International Conference on Machine Learning (ICML)*, pages 1467–1474, 2012.

[9] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In S. Jajodia and J. Lopez, editors, *Computer Security - ESORICS 2008*, pages 192–206. Springer Berlin Heidelberg, 2008.

[10] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Correlated pseudorandom functions from variable-density LPN. In *Cryptology ePrint Archive: Report 2020/1417*, 2020.

[11] M. Byali, H. Chaudhari, A. Patra, and A. Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. In *Privacy Enhancing Technologies Symposium*, 2020.

[12] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, pages 136–145, 2001.

[13] C. Canonne, G. Kamath, and T. Steinke. The discrete Gaussian for differential privacy. In *arXiv 2004.00010*, 2020.

[14] Cape Privacy. TF-Trusted. URL `https://github.com/capeprivacy/tf-trusted`.

[15] O. Catrina and S. De Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks*, pages 182–199. Springer, 2010.

[16] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning. In *IEEE European Symposium on Security and Privacy*, 2019.

[17] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.

[18] R. Collobert, C. Puhrsch, and G. Synnaeve. Wav2letter: An end-to-end convnet-based speech recognition system. In *arXiv:1609.03193*, 2016.

[19] R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Lecture Notes in Computer Science*, volume 3378, pages 342–362, 2005.

[20] cuBLAS. `https://developer.nvidia.com/cublas`.

[21] cuDNN. `https://developer.nvidia.com/cudnn`.

[22] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, 2005.

[23] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. `https://eprint.iacr.org/2011/535`.

[24] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

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

[26] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. In *arXiv:1702.08608*, 2017.

[27] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[28] T. Dugan and X. Zou. A survey of secure multiparty computation protocols for privacy preserving genetic tests. In *Proceedings of the International Symposium on Biomedical Imaging*, 2016.

[29] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[30] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[31] D. Evans, V. Kolesnikov, and M. Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation*. NOW Publishers, 2018.

[32] Gloo. `https://github.com/facebookincubator/gloo`.

[33] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[34] C. Guo, A. Hannun, B. Knott, L. van der Maaten, M. Tygert, and R. Zhu. Secure multiparty computations in floating-point arithmetic. In *arXiv:2001.03192*, 2020.

[35] V. Haralampieva, D. Rueckert, and J. Passerat-Palmbach. A systematic comparison of encrypted machine learning solutions for image classification. In *Proceedings of the Workshop on Privacy-Preserving Machine Learning in Practice*, pages 55–59, 2020.

[36] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. SoK: general-purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[37] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[38] A. S. Householder. *The Numerical Treatment of a Single Nonlinear Equation*. 1970.

[39] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72: 56–64, 1989.

[40] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *arXiv 1801.05507*, 2018.

[41] M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

[42] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[43] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. CrypTFlow: Secure TensorFlow inference. In *IEEE Symposium on Security and Privacy*, pages 336–353, 2020.

[44] A. Lapets, N. Volgushev, A. Bestavros, F. Jansen, and M. Varia. Secure multi-party computation for analytics deployed as a lightweight web application. Technical Report BU-CS-TR 2016-008, Boston University, 2016.

[45] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[46] H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *arXiv:1806.09055*, 2018.

[47] J. Liu, M. J. Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[48] I. Mironov. On significance of the least significant bits for differential privacy. In *Proceedings ACM Conference on Computer and Communications Security (CCS)*, pages 650–661, 2012.

[49] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security Symposium*, 2020.

[50] P. Mohassel and P. Rindal. ABY3: A mixed protocol framework for machine learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[51] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE Symposium on Security and Privacy*, 2017.

[52] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38. IEEE, 2017.

[53] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2010.

[54] NCCL. `https://developer.nvidia.com/nccl`.

[55] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[56] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. LibriSpeech: An ASR corpus based on public domain audio books. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2015.

[57] A. Paszke, S. Gross, S. Chintala, and G. Chanan. PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration, 2017.

[58] 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 *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[59] A. Patra and A. Suresh. Blaze: Blazing fast privacy preserving machine learning. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[60] R. Rachuri and A. Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *Symposium on Network and Distributed System Security*, 2019.

[61] I. S. Reed. Information theory and privacy in data banks. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, 1973.

[62] 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 *Cryptology ePrint Archive*, volume 2017/1164, 2017.

[63] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. In *USENIX Security*, 2019.

[64] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach. A generic framework for privacy preserving deep learning. In *arXiv:1811.04017*, 2018.

[65] O. Seker, S. Berndt, L. Wilke, and T. Eisenbarth. SNI-in-the-head: Protecting MPC-in-the-head protocols against side-channel analysis. In *Cryptology ePrint Archive: Report 2020/544*, 2020.

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

[67] S. Tan, B. Knott, Y. Tian, and D. J. Wu. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *arXiv 2104.10949*, 2021.

[68] I. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy. MLP-Mixer: An all-MLP architecture for vision. In *arXiv:2105.01601*, 2021.

[69] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[70] A. R. W. A. Khan, M. A. Noor. Higher-order iterative methods by using householders method for solving certain nonlinear equations. In *Mathematical Science Letters*, pages 107–120. Natural Sciences Publishing, May 2013.

[71] S. Wagh, D. Gupta, and N. Chandran. SecureNN: Efficient and private neural network training. In *Cryptology ePrint Archive*, volume 2018/442, 2018.

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

[73] S. L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

[74] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.

[75] Yelp. Yelp Review Dataset. URL `https://www.yelp.com/dataset`.

[76] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *arXiv:1611.01578*, 2016.

A Overview of Functions Implemented in CrypTen

Table 1 gives an overview of all functions currently implemented in CrypTen, together with a short description of the approach used to implement the function. Random samplers are not listed in the table. For full details on the CrypTen secure-computation protocol, we refer to Appendix C.

Function	Function name(s)	Description
Absolute	abs	Multiply value by its sign.
Addition	add, +	Each party adds their shares.
Argument of maximum	argmax	Perform pairwise comparisons or tree reduction.
Argument of minimum	argmin	Perform pairwise comparisons or tree reduction.
Average pooling	avg_pool2d	Each party computes the average pooling of its share.
Batch normalization	batchnorm	Batch normalize values using summation, division, and variance functions.
Binary AND	and, &	Compute using binary Beaver protocol.
Binary cross-entropy	binary_cross_entropy	Compute using logarithm, multiplication, and addition functions.
Binary XOR	xor, ^	Each party XORs it shares.
Clone	clone	Each party clones their share.
Comparison	>=, <=, =, ge, le, eq	To compare to 0, convert to binary secret share and inspect most significant bit.
Concatenation	cat	Each party concatenates their shares.
Convolution	conv1d, conv2d	If filter is public, each party convolves its share. If filter is private, compute using Beaver protocol.
Cosine	cos	Approximate using repeated-squaring method.
Cross-entropy	cross_entropy	Compute using softmax, logarithm, multiplication, and division functions.
Cumulative sum	cumsum	Each party computes cumulative sum of values in its share.
Division	div, /	If divisor is public, divide shares by value and correct for wrap-around errors.
Dot product	dot	Multiply all elements and sum results.
Dropout	dropout	Each party multiplies their share with dropout mask. Dropout mask is not encrypted.
Error function	erf	Approximate using Maclaurin series.
Exponent	exp	Approximate using limit approximation.
Flatten	flatten	Each party flattens their share.
Flip	flip	Each party flips their share.
Hard tangent	hardtanh	Compute using comparison, multiplication, and addition functions.
Logarithm	log	Approximate using higher-order modified Householder method.
Log-softmax	log_softmax	Compute using exponentiation, maximum, summation, and addition functions.
Matrix multiplication	matmul	If one matrix is public, each party matrix-multiplies its share. If both matrices are private, compute using Beaver protocol.
Maximum	max	Compute argmax as one-hot vector; compute dot product with input.
Max pooling	max_pool2d	Compute maximum value.
Mean	mean	Each party computes mean of its share.
Minimum	min	Compute argmin as one-hot vector; compute dot product with input.
Multiplication	mul, *	If multiplier is public, each party multiplies its share with the multiplier. If multiplier is private, use Beaver protocol.
Multiplexing	where	Multiply first value by binary mask; add second value multiplied by inverse mask.
Negation	neg	Each party negates their share.
Norm	norm	Compute using the square, sum, and square root functions.
Outer product	ger	Perform multiplication of each pair of elements.
Padding	pad	Each party pads their share.
Permute	permute	Each party permutes their share. Indexes are not encrypted.
Product	prod	Multiply all elements in the input.
Power	pow, pos_pow	For positive powers, multiply in log-domain and exponentiate. For negative powers, compute reciprocal and evaluate positive power.
Reciprocal	reciprocal	Approximate using Newton-Rhapson iterations.
ReLU	relu, relu6	Compare values with 0, and multiply values by the resulting mask.
Reshaping	reshape, view	Each party reshapes their share.
Rolling	roll	Each party rolls their share.
Scattering	scatter	Each party scatters one share into the other share. Indexes are not encrypted.
Selection	gather, index_select, narrow, take	Each party selects part of their share. Indexes are not encrypted.
Sigmoid	sigmoid	Compute using the exponential and reciprocal functions.
Sign	sign	Compare value with 0, multiply by 2, and subtract 1.
Sine	sin	Approximate using repeated-squaring method.
Softmax	softmax	Compute using exponentiation, maximum, summation, and reciprocal functions.
Square	square	Compute using Beaver protocol.
Square root	sqrt	Approximate using Newton-Rhapson iterations.
Squeezing	squeeze	Each party removes dimensions with size 1 from their share.
Stacking	stack	Each party stacks their shares.
Subtraction	sub, -	Each party subtracts their shares.
Summation	sum	Each party sums all values in its share.
Tangent	tanh	Perform linear transformation of sigmoid value of output.
Trace	trace	Each party sums all diagonal elements of their share.
Transpose	t, transpose	Each party transposes their share.
Unsqueezing	unsqueeze	Each party adds dimensions with size 1 to their share.
Variance	var	Compute using square, addition, and subtraction functions.

Table 1: Overview of all functions on tensors implemented in CrypTen.

B Security of CrypTen Functions

CrypTen provides MPC implementations of a large number of functions. However, these functions are all composed from a small set of primitives, which are listed in Table 2. CrypTen provides the security guarantee in Lemma 1. The proof for this security guarantee follows trivially from the following observations and results from prior work:

i. Operations in arithmetic secret sharing are performed in the ring $\mathbb{Z}_{2^L}$. Multiplications in this ring are proven to be secure in [9].
ii. Operations in binary secret sharing are performed using the GMW protocol [33]. AND operations in this protocol are proven to be secure in [25].
iii. Conversion from arithmetic to binary (A2B) secret shares is performed using the protocol that is proven to be secure in [22].
iv. Tensor indexing operations like concatenation, selection, reshaping, *etc.* are non-interactive, which implies an adversary cannot gain any information.
v. Security proofs for custom MPC protocols are provided in Appendix C (see Table 2 for details).
vi. All other operations are compositions of secure functions (see Appendix C for details). This implies they are secure because security is closed under composition [12].

MPC Primitive	Round Complexity	Security Proof
Arithmetic secret sharing		
Addition	0	Non-interactive
Multiplication	1	[9, Theorem 1]
Truncation	$1^\dagger$	Appendix C.1.1
Binary secret sharing		
XOR	0	Non-interactive
AND	1	[25, §III.B]
Bit-shift	0	Non-interactive
Conversions		
A2B	$\log_2(\lvert\mathcal{P}\rvert)\log_2(L)$	[22, §3]
B2A	1	Appendix C.1.2
Sampling		
Bernoulli(.5)	1	Appendix C.3.1

Table 2: Overview of the MPC primitives used in CrypTen, with their round complexity and references to the relevant security proof. Round complexity is defined as the number of sequential round-trips of communication required between parties to implement a given function, using an L-bit ring and $|\mathcal{P}|$ parties. †The number of rounds needed for truncation in the two-party setting is zero.

C Detailed Description of Secure MPC Protocols

C.1 Secret Sharing

CrypTen uses two different types of secret sharing: (1) arithmetic secret sharing [23] and (2) binary secret sharing [33]. Below, we describe the secret sharing methods for single values x but they can trivially be extended to real-valued vectors $\mathbf{x}$.

C.1.1 Arithmetic Secret Sharing

CrypTen uses arithmetic secret sharing to perform most MPC computations. In arithmetic secret sharing, a scalar value $x \in \mathbb{Z}/Q\mathbb{Z}$ (where $\mathbb{Z}/Q\mathbb{Z}$ denotes a ring with Q elements) is shared across $|\mathcal{P}|$ parties in such a way that the sum of the shares reconstructs the original value x. We denote the sharing of x by $[x] = \{[x]_p\}_{p\in\mathcal{P}}$, where $[x]_p \in \mathbb{Z}/Q\mathbb{Z}$ indicates party p's share of x. The representation has the property that $\sum_{p\in\mathcal{P}}[x]_p \mod Q = x$. We use a fixed-point encoding to obtain

Algorithm 1: Private computation of the wrap count for an arithmetically shared value.

```
Input: Arithmetic secret shared value $[x]$,
Secret shared random value $[r]$ and its wrap count $[\theta_x]$.
Compute: $[z] \leftarrow [x] + [r]$
for $p \in \mathcal{P}$ do
    Party $p$ computes: $[\beta_{xr}]_p \leftarrow ([x]_p + [r]_p - [z]_p)/Q$.
end for
Construct: $[\beta_{xr}] = \{[\beta_{xr}]_p\}_{p \in \mathcal{P}}$
Decrypt: $z \leftarrow$ reveal($[z]$)
Compute during decryption: $\theta_z \leftarrow (\sum_p [z]_p - z)/Q$.
Compute: $[\eta_{xr}] \leftarrow z < [r]$
Compute: $[\theta_x] \leftarrow \theta_z + [\beta_{xr}] - [\theta_r] - [\eta_{xr}]$
```

x from a floating-point value x_R. To do so, we multiply x_R with a large scaling factor B and round to the nearest integer: $x = \lfloor Bx_R \rceil$, where $B = 2^L$ for some precision parameter, L. To decode a value, x, we compute $x_R \approx x/B$. Encoding real-valued numbers this way incurs a precision loss that is inversely proportional to L. Since we scale by a factor B to encode numbers, we must scale down by a factor B after every multiplication. We do this using the truncation protocol described below.

Addition. The addition of two secret-shared values, $[z] = [x] + [y]$, can be trivially implemented by having each party p sum their shares of $[x]$ and $[y]$: each party $p \in \mathcal{P}$ computes $[z]_p \leftarrow [x]_p + [y]_p$.

Multiplication. To facilitate multiplication of two secret shared values, the parties use random Beaver triples [7], generated in an offline preprocessing phase. A Beaver triple of secret shared values $([a], [b], [c])$ satisfies the property $c = ab$. The parties use the Beaver triple to compute $[\epsilon] = [x] - [a]$ and $[\delta] = [y] - [b]$ and decrypt ϵ and δ. This does not leak information if a and b were drawn uniformly at random from the ring $\mathbb{Z}/Q\mathbb{Z}$. The product $[x][y]$ can now be evaluated by computing $[c] + \epsilon[b] + [a]\delta + \epsilon\delta$, where ϵ and δ requires a round of communication among all parties. It is straightforward to confirm that the result of the private multiplication is correct:

$$\begin{aligned}[c] + \epsilon[b] + [a]\delta + \epsilon\delta &= [a][b] + [x][b] - [a][b] + [y][a] - [b][a] + ([x] - [a])([y] - [b]) \\ &= [x][y].\end{aligned}$$

Because this result holds for any linear function, $f(\cdot)$, of two variables for which the triple (a, b, c) satisfies $c = f(a, b)$, we use the same procedure to perform matrix multiplication and convolution.

Square. To compute the square $[x^2]$, the parties use a Beaver pair $([a], [b])$ such that $b = a^2$. The parties compute $[\epsilon] = [x] - [a]$, decrypt ϵ, and obtain the result via $[x^2] = [b] + 2\epsilon[a] + \epsilon^2$.

Truncation. A simple method to divide an arithmetically shared value, $[x]$, by a public value, ℓ, would divide the share of each party by ℓ. However, such a method can produce incorrect results when the sum of shares "wraps around" the ring size, Q. Defining θ_x to be the number of wraps such that $x = \sum_{p \in \mathcal{P}} [x]_p - \theta_x Q$, indeed, we observe that:

$$\frac{x}{\ell} = \sum_{p \in \mathcal{P}} \frac{[x]_p}{\ell} - \frac{\theta_x}{\ell} Q \neq \sum_{p \in \mathcal{P}} \frac{[x]_p}{\ell} - \theta_x Q.$$

Therefore, the simple division method fails when $\theta_x \neq 0$, which happens with probability $^x/_Q$ in the two-party case. Many MPC implementations specialize to the $|\mathcal{P}| = 2$-party case and assume this probability is negligible [52, 62, 71]. However, when $|\mathcal{P}| > 2$ the probability of failure grows rapidly and we must account for the number of wraps, θ_x. We do so by privately computing a secret share of the number of wraps in x, $[\theta_x]$. To this end, we define three auxiliary variables:

- θ_x represents the number of wraps produced by the shares of a secret shared variable $[x]$, such that $x = \sum_p [x]_p - \theta_x Q$, where Q is the ring size.
- β_{xr} represents the differential wraps produced between each party's shares of two secret shared variables, $[x]$ and $[r]$, such that $[x]_i + [r]_i \mod Q = [x]_i + [r]_i - [\beta_{xr}]_i Q$.
- η_{xr} represents the wraps produced by two plaintext variables, x and r, such that $x + r \mod Q = x + r - \eta_{xr} Q$.

We use these variable in Algorithm 1 to compute $[\theta_x]$. This approach is inspired by Algorithm 4 of [71], but extends to an arbitrary number of parties. The correctness of this algorithm can be shown through the following reduction:

$$\begin{aligned}
z &= x + r - \eta_{xr} Q \\
\textstyle\sum_p [z]_p - \theta_z Q &= (\textstyle\sum_p [x]_p - \theta_x Q) + (\textstyle\sum_p [r]_p - \theta_r Q) - \eta_{xr} Q \\
\textstyle\sum_p [z]_p - \theta_z Q &= (\textstyle\sum_p [x]_p + [r]_p) - (\theta_x + \theta_r + \eta_{xr}) Q \\
\textstyle\sum_p [z]_p - \theta_z Q &= (\textstyle\sum_p [z]_p - [\beta_{xr}]_p Q) - (\theta_x + \theta_r + \eta_{xr}) Q \\
\textstyle\sum_p [z]_p - \theta_z Q &= (\textstyle\sum_p [z]_p) - (\beta_{xr} + \theta_x + \theta_r + \eta_{xr}) Q \\
\theta_x &= \theta_z + \beta_{xr} - \theta_r - \eta_{xr}.
\end{aligned}$$

We then use $[\theta_x]$ to correct the value of the division by ℓ:

$$\frac{x}{\ell} = [y] - [\theta_x] \frac{Q}{\ell} \quad \text{where} \quad [y] = \left\{ \frac{[x]_p}{\ell} \right\}_{p \in \mathcal{P}}.$$

In practice, it can be difficult to compute $[\eta_{xr}]$ in Algorithm 1. However, we note that η_{xr} has a fixed probability of being non-zero, irrespective of the number of parties. Indeed, regardless of the number of parties, we have $P(\eta_{xr} \neq 0) = {}^{x}\!/_{Q}$. In practice, we can therefore skip the computation of $[\eta_{xr}]$ and simply set $\eta_{xr} = 0$. This implies that incorrect results can be produced by our algorithm with small probability. For example, when we encode a real value $\hat{x}$ using a fixed-point encoding $x = \lfloor B\hat{x} \rceil$, truncation will produce an error with probability $P(\eta_{xr} \neq 0) = {}^{\lfloor B\hat{x} \rceil}\!/_{Q}$. This probability can be reduced by increasing Q or reducing the precision parameter, B.

Security proof. One can show the security of Algorithm 1 by noting that the only information gained by an adversary is the revealed shares of $[z]$, which are indistinguishable from white uniform random noise because shares of $[r]$ are chosen to be uniformly random.

C.1.2 Binary Secret Sharing

Binary secret sharing is a special case of arithmetic secret sharing that operates within the binary field $\mathbb{Z}/2\mathbb{Z}$. In binary secret sharing, a sharing $\langle x \rangle$ of a value x is generated as a set of arithmetic secret shares of the bits of x within the binary field. Each party $p \in \mathcal{P}$ holds a share $\langle x \rangle_p$ that satisfies $x = \bigoplus_{p \in \mathcal{P}} \langle x \rangle_p$. Because addition and multiplication modulo 2 are equivalent to binary XOR and AND operations, we can use bitwise operations on integer types to vectorize these operations.

Note that XOR and AND operations form a basis for the set of Turing-complete operations (via circuits). However each sequential AND gate requires a round of communication, which makes all but very simple circuits very inefficient to evaluate via binary secret sharing. In CrypTen, we only use binary secret sharing to implement comparators.

Bitwise XOR. Similar to addition in arithmetic secret sharing, a binary XOR of two binary secret-shared values, $\langle z \rangle = \langle x \rangle + \langle y \rangle$ can be trivially implemented by having each party XOR their shares of $\langle x \rangle$ and $\langle y \rangle$. That is, each party $p \in \mathcal{P}$ computes $\langle z \rangle_p \leftarrow \langle x \rangle_p \oplus \langle y \rangle_p$.

Bitwise AND. Since the bitwise AND operation is equivalent multiplication mod 2, we can utilize the same method we use to multiply arithmetic secret shared values. To facilitate bitwise AND of two binary secret-shared values, the parties use random triples generated in an offline preprocessing phase. The generated triple $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ satisfies the property $c = a \otimes b$. The parties then compute $\langle \epsilon \rangle = \langle x \rangle \oplus \langle a \rangle$ and $\langle \delta \rangle = \langle y \rangle \oplus \langle b \rangle$ and decrypt ϵ and δ. This does not leak information since a and b contain bits drawn uniformly at random. $\langle x \rangle \otimes \langle y \rangle$ can now be evaluated by computing $\langle c \rangle \oplus (\epsilon \otimes \langle b \rangle) \oplus (\langle a \rangle \otimes \delta) \oplus (\epsilon \otimes \delta)$. Correctness follows from the same logic as multiplication in arithmetic secret sharing. We note that revealing ϵ and δ requires a round of communication among all parties in this protocol.

Logical shifts. Because each bit of a binary secret-shared value is an independent secret-shared bit, logical shifts can be performed trivially. To shift the bits of a binary secret-shared value $\langle x \rangle$ by a constant k, each party can compute the shift locally on its share, $\langle y \rangle_p = \langle x \rangle_p >> k$.

C.1.3 Converting Between Secret-Sharing Types

Many machine-learning models require both functions that are easier to compute on arithmetic secret shares (*e.g.*, matrix multiplication) and functions that are easier to implement via circuits on binary

Algorithm 2: Private single bit conversion from binary to arithmetic sharing.

```
Input: Binary secret shared bit $\langle b\rangle$; random bit in both arithmetic and binary sharing $[r]$, $\langle r\rangle$.
Compute: $\langle z\rangle \leftarrow \langle b\rangle \oplus \langle r\rangle$.
Decrypt: $z \leftarrow \text{reveal}(\langle z\rangle)$.
Compute: $[b] \leftarrow [r] + z - 2[r]z$.
```

secret shares (*e.g.*, argmax). Therefore, CrypTen uses both types of secret sharing and converts between the two types as needed using the techniques proposed in [25].

From $[x]$ to $\langle x\rangle$: To convert from an arithmetic share $[x]$ to a binary share $\langle x\rangle$, each party first secretly shares its arithmetic share with the other parties and then performs addition of the resulting shares. The parties construct binary secret shared values $\langle y_p\rangle$ where each y_p represents one of the arithmetic secret shares $y_p = [x]_p$. This process is repeated for each party $p \in \mathcal{P}$ to create binary secret shares of all $|\mathcal{P}|$ arithmetic shares $[x]_p$. Subsequently, the parties compute $\langle x\rangle = \sum_{p\in\mathcal{P}} \langle y_p\rangle$. To compute the sum, a carry-lookahead adder circuit can be evaluated in $\log_2(|\mathcal{P}|)\log_2(L)$ rounds [15, 22]. In practice, the carry-lookahead adder circuit is quite memory-intensive. When CrypTen runs out of GPU memory, we adopt an alternative adder circuit that requires substantially less memory but performs $|\mathcal{P}|\log_2(L)$ communication rounds to perform the summation.

From $\langle x\rangle$ to $[x]$: To convert from a binary share $\langle x\rangle$ to an arithmetic share $[x]$, the parties compute $[x] = \sum_{b=1}^{B} 2^b \left[\langle x\rangle^{(b)}\right]$, where $\langle x\rangle^{(b)}$ denotes the b-th bit of the binary share $\langle x\rangle$ and B is the total number of bits in the shared secret. To create the arithmetic share of a bit, the parties use b pairs of secret-shared bits $([r], \langle r\rangle)$ generated offline. Herein, $[r]$ and $\langle r\rangle$ represent arithmetic and binary secret-shares of the same bit value r. Parties then use Algorithm 2 to generate $\left[\langle x\rangle^{(b)}\right]$ from $\langle x\rangle^{(b)}$. This process can be performed for each bit in parallel, reducing the number of communication rounds required for the conversion process to one.

Security proof. One can show the security of Algorithm 2 by noting that the only information gained by an adversary is the revealed shares of $\langle z\rangle$, which are indistinguishable from white Bernoulli random noise because shares of $\langle r\rangle$ are chosen to be uniformly random.

C.1.4 Logic-based Operations

Many applications require implementations of logic-based operators to make branching decisions and compute piece-wise functions.

Comparisons. To compare two secret-shared values $[x]$ and $[y]$, we can produce $[x < y]$ by computing their difference $[z] = [x] - [y]$ and comparing the result to zero: $[z < 0]$. We compute $[z < 0]$ by first converting $[z]$ to a binary secret-share $\langle z\rangle$, computing its sign bit using a right shift $\langle b\rangle = \langle z\rangle >> (L-1)$, and converting the resulting bit to an arithmetic sharing $[b]$. Because we are using an integer encoding, the most significant bit of z represents its sign. It is possible to compare $[x < y]$ directly using a less-than circuit, but this requires converting an extra value to binary secret sharing and incurring another $\log_2 L$ rounds of communication to compute the less-than circuit.

We can use the ability to compute $[x < y]$ to compute all other comparators on $[x]$ and $[y]$:

$$\begin{aligned}
[x > y] &= [y < x] \\
[x \geq y] &= 1 - [x < y] \\
[x \leq y] &= 1 - [y < x] \\
[x = y] &= [x \leq y] - [x < y] \\
[x \neq y] &= 1 - [x = y].
\end{aligned}$$

We optimize evaluation of the is-equal operator by computing $[x \leq y]$ and $[x < y]$ in parallel.

Multiplexing. Multiplexing is a very valuable tool for computing conditional and piece-wise functions. To multiplex between two values $[x]$ and $[y]$ based on a condition c, we must first evaluate c to a a binary value $[c] \in \{[0], [1]\}$. We can then compute $[c\ ?\ x : y] = [c][x] + (1 - [c])[y]$. This allows us to evaluate if-statements using CrypTen, where $[x]$ is the result when the if-statement is executed, and $[y]$ is the result otherwise. However, unlike if-statements, both results must be evaluated, meaning we cannot use tree-based or dynamic programming techniques to optimize algorithm runtimes.

Sign, absolute value, and ReLU. Several important functions can be computed using the multiplexing technique. We can compute $\text{sign}([x]) = 2[x > 0] - 1$. We can then use this to compute $|[x]| = [x]\,\text{sign}([x])$. Similarly we can compute the ReLU function by noting $\text{ReLU}([x]) = [x][x > 0]$.

Argmax and maximum. CRYPTEN supports two methods for computing maximums $[\max x]$. Both methods first compute a one-hot argmax mask that contains a one at the index containing a maximal element $[y] = \text{argmax}([x])$. A maximum can then be obtained by taking the sum $[\max x] = \sum_i [y_i][x_i]$ where the sum is taken along the dimension over which the maximum is being computed. By default, the argmax is computed using a tree-reduction algorithm, though configurations are available to use pairwise comparisons depending on network bandwidth / latency.

The *tree-reduction* algorithm computes the argmax by partitioning the input into two halves, then comparing the elements of each half. This reduces the size of the input by half in each round, requiring $O(\log_2 N)$ rounds to complete the argmax. This method requires order $O(\log_2 N)$ communication rounds, $O(N^2)$ communication bits, and $O(N)$ computation complexity.

The *pairwise* method generates a matrix $[A]$ whose rows are constructed by the pairwise differences of every pair of elements $\forall i \neq j : [A_{ij}] = [x_i - x_j]$. We then evaluate all comparisons simultaneously by computing $[A \geq 0]$. All maximal elements will correspond to columns whose elements are all greater than 0, so we can compute the argmax mask $[m]$ by taking the sum over all columns of $[A]$. However, if more than one maximal element exists, this will result in a mask $[m]$ that is not one-hot. To make this one-hot we take a cumulative sum $[c]$ of $[m]$ and return $[c < 2][m]$ to return the index of the first maximal element. This method requires $O(1)$ communication rounds, $O(N^2)$ communication bits, and $O(N^2)$ computation complexity. In theory, because of constant-round communication, this method should be more efficient than the tree-reduction algorithm when the network latency is high.

Argmin and minimum. To compute minimums and argmins, we compute our argmax mask with a negated input: $[\text{argmin}\, x] = [\text{argmax}(-x)]$.

C.2 Mathematical Approximations

Many functions are very expensive to compute exactly using only addition, multiplication, truncation, and comparisons. CRYPTEN uses numerical approximations to compute these functions, optimizing for accuracy, domain size, and efficiency when computed on secret shares. Each of these approximations has a specific domain over which the approximation converges well. One can modify the domain of convergence for certain functions using function-specific identities. For example, $\forall a \in \mathbb{R}$:

$$\begin{aligned} \ln(x) &= \ln(ax) - \ln(a) \\ x^{-1} &= a(ax)^{-1} \\ e^x &= e^{x-a}e^a. \end{aligned}$$

CRYPTEN also offers configurable parameters for protocol-specific optimizations, for example, custom initializations that improve convergence for iterative methods in a pre-specified input domain.

C.2.1 Exponential, Sine, and Cosine

There are many well-known polynomial approximations for the exponential function, for example, the Taylor series, $e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n$. However, because exponentials grow much faster than polynomials, the degree of the polynomial we would need to approximate the exponential function increases exponentially as the domain increases. Therefore, we instead use the limit approximation, which allows us to do repeated squaring very efficiently:

$$e^x = \lim_{n \to \infty} \left(1 + \frac{x}{2^n}\right)^{2^n}.$$

CRYPTEN can also use the repeated-squaring method to compute complex exponentials efficiently, which enables the computation of the sine and cosine functions:

$$\begin{aligned} \cos x &= \Re(e^{ix}) \\ \sin x &= \Im(e^{ix}). \end{aligned}$$

C.2.2 Reciprocal

CrypTen uses Newton-Raphson iterations to compute the reciprocal function. This method uses an initial guess, y_0, for the reciprocal and repeats the following update:

$$y_{n+1} = y_n(2 - xy_n).$$

This will converge to $\lim_{n \to \infty} y_n = \frac{1}{x}$ quadratically as long as the initial guess y_0 meets the Newton-Raphson convergence criterion, which is $0 < y_0 < \frac{2}{x}$ for the above. By default, CrypTen uses:

$$y_0(x) = 3e^{0.5 - x} + 0.003,$$

to initialize the approximation, which provides convergence on a large domain. This function was found by inspection and can be replaced by a user-defined value using CrypTen's configuration API. Because this method only converges for positive values of x, we compute the reciprocal using the identity $\frac{1}{x} = \frac{\operatorname{sgn} x}{|x|}$. (Note that square matrix inverses and Moore-Penrose inverses can be found using similar techniques given input matrices with singular values that meet the convergence criterion.)

C.2.3 Square Root and Normalization

CrypTen uses Newton-Raphson iterations to compute square roots. However, the Newton-Raphson update formula for square roots, $y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})$ is quite inefficient to compute on secret shares. Instead, we use the much more efficient Newton-Raphson update formula for inverse square root:

$$y_{n+1} = \frac{1}{2} y_n (3 - x y_n^2).$$

We then multiply by the input x to obtain the square root: $\sqrt{x} = (x^{-0.5})x$. We can also use the inverse square root function to efficiently normalize values via: $\frac{x}{\|x\|} = x \left(\sum_i x_i^2\right)^{-1/2}$.

C.2.4 Logarithm and Exponents

To compute logarithms, CrypTen uses higher-order iterative methods to achieve better convergence. The following update formula can be found using high-order modified Householder methods on $\ln(x)$ [70] or by manipulating the Taylor series expansion of $\ln(1 - x)$:

$$h_n = 1 - xe^{-y_n}$$
$$y_{n+1} = y_n - \sum_{k=1}^{\infty} \frac{1}{k} h_n^k.$$

Note that at each step $\ln x = y_n + \ln(1 - h_n)$, but we can only approximate $\ln(1 - h_n)$ using a truncated Taylor Series approximation. For this method, the order of the Householder method (*i.e.*, the polynomial degree in the second equation) will determine the speed of convergence. Since the convergence rate per iteration increases proportionally to the degree of the polynomial, whereas an exponential must be computed for each iteration, it is more computationally efficient to use high-degree polynomials instead of doing many iterations. By default, CrypTen uses a polynomial of degree 8, the initialization $y_0 = \frac{x}{120} - 20e^{-2x-1} + 3$, and 3 iterations. This provides effective convergence on the domain $[10^{-4}, 10^2]$.

Using the logarithm and exponential functions, we can also compute arbitrary public or private exponents on positive inputs x using the equation $x^y = e^{y \ln(x)}$.

C.2.5 Sigmoid and Hyperbolic Tangent

We have explored several methods for computing logistic functions in MPC, including direct computation, rational approximations, and Chebyshev polynomial approximations [34]. We have found that direct computation is the most efficient when it is combined with some specific optimizations. Specifically, CrypTen uses the exponential and reciprocal functions to compute:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

We optimize this computation by noting that the range of the sigmoid function is $[0, 1]$, and the range for the positive half of its domain is $[0.5, 1]$. Therefore, when we compute the reciprocal using the method described in Section C.2.2, we compute $\sigma(|x|)$ using an initialized value of 0.75 for the Newton-Raphson iterations to improve convergence. We extend the result to the full domain by noting $\sigma(-x) = 1 - \sigma(x)$. We compute the hyperbolic tangent function via $\tanh(x) = 2\sigma(2x) - 1$.

C.2.6 Gaussian Error Function

We use a Maclaurin series to approximate the Gaussian error function $\text{erf}(x) = {}^{2}/_{\sqrt{\pi}} \int_0^x e^{-x^2} dx$. The resulting approximation is given by: $\text{erf}(x) \approx \frac{2}{\sqrt{\pi}} \sum_{k=0}^{K} \frac{(-1)^k x^{2k+1}}{k!(2k+1)}$, where K is the number of terms in the approximation (we set $K = 8$ by default). Although the approximation works reasonably well in practice, we note that it is known to have poor convergence when $x > 1$ (see OEIS A007680).

C.3 Random Sampling

Several applications of privacy-preserving computations require secret-shared generation of random numbers such that no party can gain any information about the value of realizations. We use the following methods for generating secret shares of random samples from several popular distributions.

C.3.1 Uniform Sampling

Due to quantization introduced by our encoding with scale 2^L, we can only produce discrete uniform random variables with 2^L possible values. To do so, we produce samples $[u] \sim Uniform(0, 1)$ by generating L bits as Rademacher variates. To generate these bits, each party randomly generates its own binary secret-share with the same distribution locally. The XOR sum of independently distributed Rademacher variates, $u = \oplus_{p \in \mathcal{P}} \langle u \rangle_p$, is itself a Rademacher variate and is uncorrelated with any of the input bits.

Security proof. One can show the security of this sampler by noting that no adversary could gain any information about the sampled bit from its own binary share of the bit, because the XOR sum of independently distributed Rademacher variates is uncorrelated with any of the input bits. The bits are then converted to an arithmetic share $[u]$ using Algorithm 2, which is itself secure.

C.3.2 Bernoulli Sampling

To compute a Bernoulli random variable with arbitrary mean $[b] \sim Bern(p)$, we first generate a uniform random variable $[u] \sim Uniform(0, 1)$ and compute $[b] = [u > p]$. Note that due to quantization in $[u]$, the true probability parameter of the Bernoulli random sample is quantized to the nearest multiple of 2^{-L}, as would have happened if p was encoded using the fixed-point encoder.

C.3.3 Gaussian Sampling

Gaussian random samples $[x] \sim \mathcal{N}(\mu, \sigma^2)$ can be computed using the Box-Muller transform. Given a pair of independent uniformly distributed random variables $([u_1], [u_2])$, two independent Gaussian random variables $([x_1], [x_2])$ from $\mathcal{N}(0, 1)$ can be generated by computing:

$$[x_1] = \sqrt{-2 \ln[u_1]} \cos(2\pi[u_2])$$
$$[x_2] = \sqrt{-2 \ln[u_1]} \sin(2\pi[u_2]).$$

Since the range of the uniform inputs is $[0, 1]$, we optimize our numerical approximations for better performance on this domain. To obtain samples $[y] \sim \mathcal{N}(\mu, \sigma^2)$, we compute $[y] = \sigma[x] + \mu$.

C.3.4 Exponential and Laplace sampling

Exponential random variables $[x] \sim Exp(\lambda)$ can be computed using the inverse CDF method. Given a uniform random sample $[u] \sim U[0, 1]$, an exponential random variable is generated via:

$$[x] = -\lambda^{-1} \ln([u]).$$

Again, we optimize the logarithm for the domain [0, 1].

A Laplace distributed random sample $[y] \sim Lap(\mu, k)$ can be generated from an exponential random sample, $[x] \sim Exp(k^{-1})$, and a Rademacher variate, $[b]$, by evaluating $[y] = (2[b] - 1)[x]$.

C.3.5 Weighted Random Sampling

To produce a weighted random sample of inputs $[x_i]$ with weights given by $[w_i]$, we first generate a uniform random sample in $([0, \sum_i [w_i])$ by drawing a uniform sample, $[u]$, and evaluating $[r] = [u]\,[\sum_i w_i]$. Care should be taken to avoid precision issues caused by generating $[u]$ in fixed-point with finite precision. We then compute the cumulative sum values $[c_i]$ of the weights $[w_i]$, and compare those values to our random value $[m_i] = [c_i > r]$. This produces a mask vector whose entries are all zero below some index j and all one above index j. To convert this mask vector into a one-hot vector, we append a zero in front of the $[m_i]$-values and compute $[o_i] = [m_i] - [m_{i+1}]$. Finally, we obtain the selected sample from the inputs $[x_i]$ by multiplying the samples with the one-hot vector and summing: $[y] = \sum_i [x_i][o_i]$.

D Comparison with Secure MPC Frameworks for Machine Learning

Table 3 presents a comparison of CrypTen with other secure MPC frameworks for machine learning. For each framework, the table shows whether the framework supports maliciously secure threat models, can generate Beaver triples (if needed) without requiring a trusted third party, supports GPU computations, supports model training, supports general purpose function evaluation, and implements automatic differentiation (autograd). We define a secure MPC framework for machine learning to be general-purpose if it supports at least the following functions: linear functions, convolutions, rectified linear units (ReLU), max-pooling, and batch normalization.[7]

Framework	**Malicious security**	**Triple generation**	**Supports GPUs**	**Supports training**	**General purpose**[†]	**Supports autograd**
Two parties						
Chameleon [62]	✗	✗	✗	✗	✗	✗
Delphi [49]	✗	✔	✗	✗	✗	✗
EzPC [16]	✗	✔	✗	✗	✗	✗
Gazelle [40]	✗	✔	✗	✗	✗	✗
MiniONN [47]	✗	✔	✗	✗	✗	✗
PySyft [64]	✗	✔	✔	✗	✗	✗
SecureML [51]	✗	✔	✗	✔	✗	✗
XONN [63]	✔	N/A	✗	✗	✗	✗
Three parties						
ABY3 [50]	✗	N/A	✗	✔	✗	✗
Astra [17]	✗	✔	✗	✔	✗	✗
Blaze [59]	✗	✔	✗	✔	✗	✗
CrypTFlow [43]	✗	N/A	✗	✗	✔	✗
CryptGPU[‡] [67]	✗	✗	✔	✔	✔	✔
Falcon [72]	✔	N/A	✗	✔	✔	✗
SecureNN [71]	✗	N/A	✗	✔	✗	✗
Four parties						
FLASH [11]	✔	N/A	✗	✔	✗	✗
Trident [60]	✔	N/A	✗	✔	✗	✗
Arbitrary number of parties						
CrypTen (ours)	✗	✗[§]	✔	✔	✔	✔

Table 3: Overview of secure MPC frameworks for machine learning and their properties. [†]We define a framework to be general-purpose if it supports all of the following layers: linear, convolution, rectified linear unit (ReLU), max-pooling, and batch normalization. [‡]We note that CryptGPU was developed *on top of* CrypTen, hence, it inherits many features from CrypTen. [§]Future versions of CrypTen will support Beaver triple generation without requiring a trusted third party.

[7] CrypTen supports a variety of functions beyond these five functions, but we focus on these five in our comparison as they are the main building blocks of many deep network architectures.